How do developers use C++ libraries? An empirical study

Di Wu Lin Chen * Yuming Zhou Baowen Xu

State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China

nju.wudi@gmail.com lchen@nju.edu.cn zhouyuming@nju.edu.cn bwxu@nju.edu.cn

***Abstract*—C++ libraries provide an abundance of reusable components for writing high-quality programs and are thus widely adopted by software developers. However, to date there is little work investigating how these libraries are actually used in real software. In this paper, we perform an empirical study to investigate the adoption of C++ standard libraries in open-source applications, with the goal to provide actionable information for developers to help them employ libraries more efficiently. To this end, we analyze 379 historical revisions of 30 applications, containing 149 million lines of C++ code, to conduct the experiment. The experimental results show that: (1) three standard libraries (i.e. Containers Library, Utilities Library, and Strings Library) are significantly more often used than other libraries; (2) the new libraries of C++11 (i.e. Regular Expressions Library, Atomic Operations Library, and Thread Support Library) are significantly less often used than the formerly-established libraries; (3) the deprecated library constructs (i.e. auto pointers, function objects, and array I/O operations) are not used at a declining frequency; and (4) applications with a larger size do not adopt libraries more frequently. Based on these results, we propose four suggestions, which could help developers learn and use C++ libraries in an efficient way.**

Keywords- Programming Language, C++, Library, Empirical Study

I. INTRODUCTION

C++ libraries are pervasively used in software development, as they enable developers to write high-quality programs by employing reusable components rather than implementing all code from scratch [4]. To date, various libraries have been provided to help solve problems of different domains. Among these libraries, the Standard C++ Library is the most renowned, since it provides a large set of standardized components that are shipped with identical behavior by every C++ implementation [5]. According to C++11 [3], the latest[1] C++ specification, the Standard C++ Library is constituted by 11 sub-libraries, including 3 new libraries introduced in C++11 and 8 old libraries established in C++98 [1] and C++03 [2]. For brevity, people generally call these sub-libraries as “standard libraries”.

In recent years, many researchers have been devoting to improve the performance of standard libraries. However, few studies focus on how these libraries are actually adopted in real software. This lack of knowledge may bring troubles to software developers, since they do not know which standard libraries are the most commonly used and need their attention to be paid on, whether they have made full use of new standard libraries, and whether deprecated library constructs are less frequently used.

In this paper, we perform an empirical study to investigate the adoption of C++ standard libraries in open-source applications, with the goal to provide actionable information for developers to help them use libraries more efficiently. To be specific, we propose the following four research questions: (1) RQ1: Which libraries are the most often used? (2) RQ2: Are the new libraries of C++11 used as often as the formerly-established libraries? (3) RQ3: Are the deprecated library constructs used at a declining frequency after C++11 was published? and (4) RQ4: Do applications with a larger size adopt libraries more frequently? The purpose of RQ1 investigates whether there exist a few libraries that are more often used than others. If the most commonly used libraries are found, we may suggest developers, especially the new comers of open-source projects, to focus on understanding and using these libraries. The purpose of RQ2 investigates whether the new libraries of C++11 have been widely used. If the answer is “Yes”, we will have empirical evidence to support that the new features of C++11 have been widely adopted in developing real software. Otherwise, we may advise developers to pay special attention on using applicable new library constructs instead of writing their own code of similar functionality. The purpose of RQ3 investigates whether the deprecated library constructs are gradually infrequently used. Due that auto pointers, function objects, and array I/O operations can be replaced by other advanced features, they have been deprecated since C++11. By investigating RQ3, we can understand whether developers have realized to reduce using these outdated library constructs. The purpose of RQ4 investigates the correlation between system size and the frequency of library use. In previous studies on Java and C# libraries [9, 10], researchers found that applications with different sizes adopt libraries differently. The empirical result for RQ4 can be used to answer whether this conclusion is also applicable to the use of C++ libraries.

In order to answer these research questions, we analyze 379 historical revisions of 30 applications, containing 149 million lines of C++ code, to conduct the experiment. The experimental results show that: (1) three standard libraries (i.e. Containers Library, Utilities Library, and Strings Library) are significantly more often used than other libraries; (2) the new libraries of C++11 (i.e. Regular Expressions Library, Atomic Operations Library, and Thread Support Library) are significantly less often used than the formerly-established libraries; (3) the deprecated library constructs (i.e. auto pointers, function objects, and array I/O operations) are not used at a declining frequency; and (4) applications with a larger size do not adopt libraries more frequently. Based on these results, we propose four suggestions, which could help developers learn and use C++ libraries in an efficient way.

* Corresponding author: Lin Chen; Email: lchen@nju.edu.cn

[1] C++14 was recently approved, but its official specification has not been released. Thus, we still serve C++11 as the latest standard of C++ in this paper.

(DOI reference number: 10.18293/SEKE2015-009)

The rest of the paper is organized as follows. Section II introduces the C++ Standard Library. Section III describes the studied applications, data collection procedure, and data analysis methods. Section IV reports the experimental results, the implications, and the threats to validity of our study. Section V discusses related work. Section VI concludes the paper and outlines the direction for future work.

II. An Overview of the C++ Standard Library

The C++ Standard Library is a general name for the standardized built-in classes, functions, and macros in C++. The whole standard library is constituted by 11 sub-libraries, which are generally called "standard libraries". Before C++11, 8 elementary standard libraries were supported. To differentiate them from new libraries of C++11, we call these libraries as "**formerly-established libraries**". These libraries basically consist of Containers, Iterators, Algorithms, Utilities, Strings, Numerics, Input/Output, and Localizations. The first three libraries together with function objects in the Utilities library constitute STL (the Standard Template Library), which provides generic classes and functions to create and operate common data structures like vectors, queues, and stacks. The other five libraries are specific to language support (as well as general-purpose utilities support), string processing, scientific computation, I/O management, and internationalization support, respectively. Since C++11, three **new libraries** have been introduced. They are Regular Expressions Library, Atomic Operations Library, and Thread Support Library. The first new library is used to perform pattern matching for strings. The other two new libraries are specific to concurrent programming, equipped with low-level (atomics-based) and high-level (thread and task-based) concurrency facilities, respectively. Moreover, three formerly-established library constructs (i.e. auto pointers, function objects, and array I/O operations) are deprecated in C++11. They are no longer supported either due to the low efficiency or due to the advanced replacers.

III. Experimental Setup

In this section, we first introduce the open-source applications used for investigating our research questions. Then, we report the data collection procedure. Finally, we describe the data analysis methods.

A. Studied Applications

To investigate the proposed research questions, we analyze 30 open-source applications, whose source code is obtained by using *svn* and *git clone* tools. These applications are selected for the following reasons: (1) they cover different application domains listed on *http://sourceforge.net*, thus making the empirical results not skewed to a specific kind of applications; (2) they have a big difference in code size, thus making the result for RQ4 sufficiently reliable; and (3) they are developed as ongoing projects, thus making the experimental data up-to-date. The detailed information of the 30 applications is shown in Table I. As we can see from Table I, these applications cover 10 software domains. Moreover, they vary in age (2 to 16 years) and code size (9 to 4731 KSLOC). For these applications, we use their latest revisions by the end of 2014 to investigate RQ1, RQ2, and RQ4 and use their historical revisions to investigate RQ3. In our experiment, the historical revisions are regularly selected as the last revisions in each season after September 2011, the release time of C++11. We do not investigate all historical revisions because the code repositories contain many dump revisions, which may pose a threat to the accuracy of our experimental data. For some applications (i.e. PN, SwiftSearch, HTEditor, and ConEmu), only a few revisions are studied. This is either due to their late establishing time or due to the long time intervals between adjacent revisions.

TABLE I. Open-Source Applications in the Study

Project	Age	C++ KSLOC of latest revision	# Studied revisions	Total C++ KSLOC[1]	Category
VLC	16	135.825	14	1855.855	Audio & Video
LameXP	5	21.449	14	315.346	
MPC-HC	9	521.267	14	9581.495	
MuPDF	11	16.137	14	140.816	Business & Enterprise
Qucs	12	125.600	13	2323.950	
LibreOffice	5	4730.718	14	68554.599	
LeechCraft	8	325.526	14	3819.669	Commu-nications
MirandaNG	3	1087.472	12	11155.677	
KopeteIMClient	13	348.629	14	4009.079	
TortoiseGit	7	457.517	14	5153.035	Develop-ment
PN	13	155.059	7	1084.903	
KDevelop	16	108.759	14	1388.152	
Warzone2100	10	186.721	12	2233.427	Games
Pentobi	4	30.818	14	375.543	
SuperTuxKart	8	369.355	14	3476.652	
Blender	13	600.906	14	6852.224	Graphics
LuminanceHDR	13	38.828	13	452.887	
FreeCAD	4	1185.593	14	15992.063	
GoldenDict	6	75.008	14	748.038	Home & Education
Kiwix	8	63.863	14	1060.158	
SUMO	13	132.400	14	1651.322	
rr	4	18.245	14	66.944	Science & Engi-neering
Trimph4php	3	80.211	10	562.450	
RStudio	2	127.943	14	1300.345	
KmyMoney	3	146.252	14	1993.554	Security & Utilities
SwiftSearch	3	8.556	6	43.196	
HTEditor	13	95.517	7	712.315	
ConsoleZ	8	65.056	14	840.411	System Adimini-stration
NVDA	9	11.291	14	139.735	
ConEmu	2	203.471	5	955.516	

B. Data Collection

We collect the experimental data by using "Understand" [17], a tool that automatically analyzes the source code of applications without manual configuration. To be specific, the data is collected by the following steps. At the first step, we obtain C++ files by using the "C++ Strict" option provided by "Understand" and build an Understand database for each studied application. At the second step, we process Understand databases to identify the use sites of standard library constructs, including library classes, library functions, and library macros. Since all standard library constructs are marked with the "std::" namespace, they can be easily detected by running a Perl script which invokes Understand APIs. At

[1] "KSLOC" means "thousand source lines of code (excluding comments)". Generally speaking, it is equal to "KLOC" ("thousand lines of code").

the third step, we compare the names of practically used constructs with the names of actual standard library constructs. We do this in order to filter out those fake standard library constructs used by developers. At the fourth step, we divide all examined standard library constructs into the new library group and the formerly-established library group. At the final step, we calculate the KSLOC value and the number of C++ files for each application by looking up the metrics reported by Understand. With these five steps, we can obtain the experimental data set, which consists of: (1) the number of use for each standard library (both formerly-established and new libraries); (2) the number of use for deprecated library constructs; (3) the number of use for standard libraries in each application and in its historical revisions; and (4) the KSLOC value and the number of files in each application.

C. *Data Analysis*

In order to answer RQ1, RQ2, and RQ3, we apply the Wilcoxon signed-rank test to examine whether two groups of data have a significant difference. More specifically, for RQ1, we compare the percentages of use for the 11 standard libraries in pair-wise. Here, the percentage is calculated as the number of use for a specific library divided by the total number of use for all libraries. If a few libraries exceed other libraries in the percentage of use at the significance level of 0.05, we will accept them as the most commonly used standard libraries. Otherwise, we will conclude that there is not an outstanding library that is more often used than others. For RQ2, we compare the percentage of use for each new library with the percentage of use for each formerly-established library. If new libraries show a significant difference (significance level = 0.05) from the formerly-established libraries in the percentage of use, we will conclude that the new libraries and formerly-established libraries are not equally commonly used. Otherwise, we will fail to reject the hypothesis that "new libraries are as often used as formerly-established libraries". For RQ3, we compare the densities of use for the deprecated library constructs in each season after C++11 was published. The densities are calculated both at line level (number of use for deprecated library constructs per KSLOC) and at file level (number of use for deprecated library constructs per file). Here, we use the density instead of the raw number of library construct use in order to avoid the impact brought by the change of system size. The answer to RQ3 will be "Yes" if the density value in one season (for instance, Dec. 2014) is significantly lower than the density value in the former season (for instance, Sep. 2014). Otherwise, we will fail to conclude that the deprecated library constructs are used at a declining frequency after C++11 was officially released. After performing each Wilcoxon signed-rank test, we further apply the Cliff's δ, which is used for median comparison, to examine whether the magnitude of difference is important from the viewpoint of practical application [6]. By convention, the magnitude of the difference is either trivial ($|\delta| < 0.147$), small (0.147-0.33), medium (0.33-0.474), or large (> 0.474) [7].

In order to answer RQ4, we use the Spearman's rank correlation analysis to examine whether the size of applications is significantly positively correlated to the frequency of library use. In previous studies [9, 10], researchers found that applications with different sizes adopt libraries differently. More specifically, larger applications tend to have more library uses. However, the raw number of library use cannot effectively reflect the frequency of library use in different applications, because larger applications usually have more functionalities and not surprisingly have more library uses. In order to remove the impact of different system size, here we use the density to replace the raw number of library use. More specifically, we first calculate the density of library use for each application. The densities are calculated both at line level (number of library use per KSLOC) and at file level (number of library use per file). Then, we calculate the Spearman's coefficient (rho) of the correlation. In particular, the p-value is employed to examine whether the correlation is significant at the significance level of 0.05. If the calculated p-value is less than 0.05, we will conclude that applications with a larger size adopt libraries more frequently. Otherwise, we will have a conclusion that the size of application does not significantly positively correlates to the frequency of library use.

IV. Results and Implications

In this section, we report in detail the experimental results and discuss their implications.

A. *RQ1: Which libraries are the most often used?*

We employ the result from the Wilcoxon signed-rank analysis for the percentage of library use to answer RQ1. In particular, we apply Figure I to describe the percentage of use for each library. In this figure, each boxplot shows the median (the horizontal line within the box), the 25th and 75th percentiles (the lower and upper sides of the box), and the mean value (the small red rectangle inside the box). By observing Figure I, we can see that the percentages of use for three libraries (i.e. Containers, Utilities, and Strings) are obviously larger than the percentages of use for other libraries (i.e. Iterators, Algorithms, Numerics, I/O, Localizations, Regular Expressions, Atomic Operations, and Thread Support), indicating that these three standard libraries are the most commonly used by developers. The data listed in Table II confirms our observation from Figure I. This table displays the result from the Wilcoxon signed-rank analysis for the pair-wise comparisons between three standard libraries (the first row) and the other eight standard libraries (the first column). In particular, we report the significance (p-value) and the magnitude (Cliff's δ) of the difference, respectively. To be specific, for the Containers Library, it significantly outperforms other eight libraries in the percentage of use (all p-values < 0.001). Moreover, the effect sizes are large in terms of Cliff's δ ($0.804 \leq |\delta| \leq 0.966$). The Utilities Library, as expected, shows a similar result, and the effect sizes are considerably large ($0.931 \leq |\delta| \leq 0.973$). For the Strings Library, its percentage of use is also significantly larger than the other eight libraries, with seven p-values less than 0.001 and one p-value equaling to 0.017. Moreover, the effect sizes are either small ($\delta = 0.329$), moderate ($\delta = 0.393$), or large ($0.482 \leq |\delta| \leq 0.862$). To summarize, the core observation from Table II is that three new libraries significantly outperform the other eight libraries in the percentage of use and the magnitude of difference is relatively large. Therefore, we have the

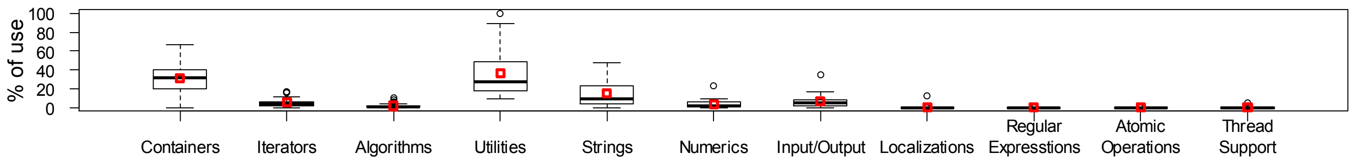


Figure I. Boxplot showing the percentage of use for standard libraries

TABLE II. RESULTS OF WILCOXON SIGNED-RANK ANALYSIS FOR RQ1

	Containers		Utilities		Strings	
	p	δ	p	δ	p	δ
Iterators	<0.001	0.820	<0.001	0.949	<0.001	0.393
Algorithms	<0.001	0.911	<0.001	0.966	<0.001	0.642
Numerics	<0.001	0.859	<0.001	0.973	<0.001	0.482
I/O	<0.001	0.804	<0.001	0.931	0.017	0.329
Localizations	<0.001	0.947	<0.001	0.963	<0.001	0.784
Regular exp.	<0.001	0.963	<0.001	0.967	<0.001	0.853
Atomic op.	<0.001	0.966	<0.001	0.967	<0.001	0.862
Thread sup.	<0.001	0.959	<0.001	0.967	<0.001	0.838

* All the p-values are BH-adjusted

following conclusion for RQ1: **three standard libraries (i.e. Containers Library, Utilities Library, and Strings Library) are significantly more often used than the other libraries**.

In order to find out which library constructs play a key role in Containers, Utilities, and Strings, we further pick out the most commonly used library constructs on ground of their number of use. All library constructs are divided into three groups, namely library classes, library functions, and library macros. According to the obtained result, library functions (73.95%) are more often used than library classes (7.89%) and library macros (18.16%). One possible explanation for this is that library functions are generally used as APIs and they are widely applied to operate elementary data structures (for instance, bitsets, shared pointers, maps, etc). Also, we find that many library classes are implemented as templates, especially the STL templates (for instance, map, set, list, and vector) and the Utilities templates (for instance, tuple, pair, bitset, numeric_limits, shared_ptr, and auto_ptr). This indicates that library templates play an important role in creating the basic data structures, which is in line with our previous findings about the utilization of templates [16]. For library macros, we find that the most commonly-used macros are inclusive members of Utilities. This result is not surprising, because an important role of the Utilities Library is to provide language support with built-in macros like UINT8_MAX, INT16_MAX, EXIT_SUCCESS, etc.

Implication. From the empirical results for RQ1, we advise developers, especially the new comers of open-source projects, to be proficient with the usage of Containers, Utilities, and Strings. Since these standard libraries are the most often used in real software development, adopting them effectively is beneficial to increase the efficiency of programming.

B. RQ2: Are the new libraries of C++11 used as often as the formerly-established libraries?

We employ the result from the Wilcoxon signed-rank analysis for the percentage of new library use to answer RQ2. Here, we exclude the experimental data provided by the applications which were established before C++11 was released. We do this mainly because these applications have already existed before the delivery of new libraries, thus investigating their use of new libraries may pose a threat to the result for RQ2. To eliminate this negative impact, we only employ the data of new library use in the applications which were established after the delivery of C++11. Table III shows the results for the pair-wise comparisons between the adoption of new libraries (the first row) and the adoption of formerly-established libraries (the first column). In particular, we report the significance (p-value) and the magnitude (Cliff's δ) of the difference, respectively. To be specific, for the Regular Expressions Library, its percentage of use is significantly different from the percentage of use for seven formerly-established libraries (p-values $\leq$ 0.016). Moreover, the effect sizes are large in terms of Cliff's δ ($0.877 \leq |\delta| \leq 1$). The only exception is the Localizations Library, which does not show a significant difference from Regular Expressions (p-value = 0.281). For the other two new libraries (i.e. Atomic Operations and Thread Support), they show a similar result as the Regular Expressions Library. From this reasoning, we conclude that new libraries and formerly-established libraries are differently used. Actually, **the new libraries of C++11 are much less often used than the formerly-established libraries**.

TABLE III. RESULTS OF WILCOXON SIGNED-RANK ANALYSIS FOR RQ2

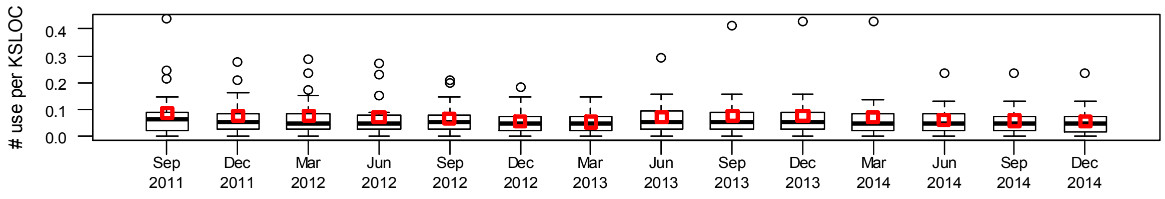
	Regular exp.		Atomic op.		Thread sup.	
	p	δ	p	δ	p	δ
Containers	0.010	-1.000	0.010	-1.000	0.010	-1.000
Iterators	0.016	-0.877	0.016	-0.889	0.034	-0.827
Algorithms	0.016	-0.877	0.016	-0.889	0.019	-0.802
Utilities	0.010	-1.000	0.010	-1.000	0.010	-1.000
Strings	0.016	-0.877	0.016	-0.889	0.019	-0.877
Numerics	0.010	-1.000	0.010	-1.000	0.010	-0.926
I/O	0.016	-0.877	0.016	-0.889	0.019	-0.877
Localizations	0.281	-0.333	0.100	-0.444	0.419	-0.309

* All p-values are BH-adjusted; p-values > 0.05 are shown in grey background.

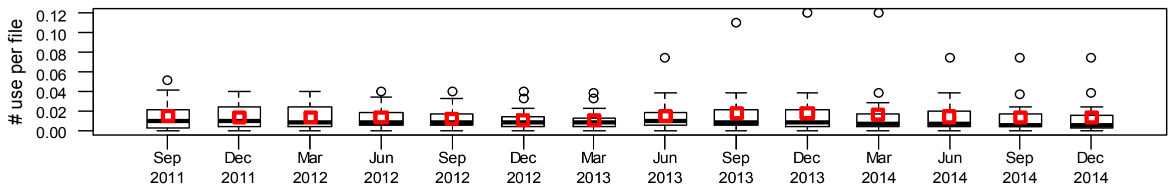
Implication. The result for RQ2 is opposed to our initial expectation that new libraries and formerly-established libraries should be equally used. One possible explanation for this is that most developers are still not familiar with the usage of new libraries, as the new libraries are been a part of the C++ standard for only three years. For this reason, we highly recommend developers to pay special attention on learning the usage of new libraries (i.e. Regular Expressions Library, Atomic Operations Library, and Thread Support Library) and employ them when they need to write string matching or concurrent programs.

C. RQ3: Are the deprecated library constructs used at a declining frequency after C++11 was published?

We employ the result from the Wilcoxon signed-rank analysis for the density of use for deprecated library constructs



(a) Density of use for the deprecated library constructs (at line level)



(b) Density of use for the deprecated library constructs (at file level)

Figure II. Density of use for the deprecated library constructs

to answer RQ3. In particular, we use Figure II to describe the density values both at line level (number of use per KSLOC) and at file level (number of use per file). In Figure II, each boxplot shows the median (the horizontal line within the box), the 25th and 75th percentiles (the lower and upper sides of the box), and the mean value (the small red rectangle inside the box). By observing the two subfigures, we do not see an obvious declining trend for the density values from Sep. 2011 to Dec. 2014, indicating that the deprecated library constructs are not decreasingly frequently used after C++11 was released. The data listed in Table IV confirms our observation from Figure II. In Table IV, we show the Wilcoxon signed-rank analysis results for comparing two adjacent seasons since September 2011. Of the 13 comparison results listed in the "Line level" group, we totally find 6 significant results (p-values < 0.05), whose effect sizes are either trivial or small in terms of Cliff's δ ($0.008 \le |\delta| \le 0.163$). By observing the "File level" column, however, we only have 3 significant results, whose effect sizes are relatively negligible ($0.044 \le |\delta| \le 0.108$). To summarize, the core observation from Table IV is that the density of use for the deprecated library constructs does not significantly decrease from late 2011 to the end of 2014. From this reasoning, we draw the conclusion for RQ3 as **the deprecated library constructs are not used at a declining frequency after C++11 was published**.

TABLE IV. RESULTS OF WILCOXON SIGNED-RANK ANALYSIS FOR RQ3

Groups for comparison	Line level		File level	
	p	δ	p	δ
Dec.2011 vs. Sep.2011	0.060	-0.006	0.133	-0.008
Mar.2012 vs. Dec.2011	0.358	-0.039	0.529	0.003
Jun.2012 vs. Mar.2012	0.032	-0.025	0.087	0.008
Sep.2012 vs. Jun.2012	0.060	-0.017	0.116	-0.019
Dec.2012 vs. Sep.2012	0.005	-0.163	0.007	-0.108
Mar.2013 vs. Dec.2012	0.032	-0.047	0.031	-0.044
Jun.2013 vs. Mar.2013	0.157	0.015	0.446	0.119
Sep.2013 vs. Jun.2013	0.377	0.019	0.534	0.014
Dec.2013 vs. Sep.2013	0.083	-0.055	0.345	-0.033
Mar.2014 vs. Dec.2013	0.074	-0.080	0.095	-0.069
Jun.2014 vs. Mar.2014	0.039	0.008	0.097	-0.003
Sep.2014 vs. Jun.2014	0.032	-0.080	0.031	-0.080
Dec.2014 vs. Sep.2014	0.013	-0.050	0.087	-0.025

* All p-values are BH-adjusted; p-values > 0.05 are shown in grey background.

Implication. One possible explanation for RQ3 is that most developers do not realize that several long-lived library constructs (i.e. auto pointers, function objects, and array I/O operations) have been deprecated since C++11. For this reason, we advise developers to keep an eye on the changes in the new C++ standards and update their code accordingly. In particular, we wish developers to remove the uses of the deprecated library constructs, because these constructs will completely stop to be supported since C++17 [18], the next major revision of the C++ programming language.

D. RQ4: Do applications with a larger size adopt libraries more frequently?

In order to answer RQ4, we use the Spearman's rank correlation analysis described in Section III.C to examine the correlation between the size of application and the density of library use. Here, we calculate application size both as KSLOC (line-level size) and as the number of files (file-level size), with the purpose to investigate RQ4 from different perspectives and obtain a consistent result. To be specific, the result of Spearman's rank correlation analysis at line level shows that application's KSLOC does not significantly correlate to the density of library use (number of library use per KSLOC) (p-value = 0.896). A similar result is reported by the Spearman's rank correlation analysis at file level, which shows that the number of files and the density of library use (number of library use per file) are not significantly correlated (p-value = 0.799). From this reasoning, we conclude that the size of application is not significantly correlated to the density of library use. In other words, **applications with a larger size do not adopt libraries more frequently**.

Implication. According to Robillard and DeLine [8], library users can efficiently understand an API if they are provided with examples to demonstrate "best practices" for using the API. Thus, it would be valuable work to explore real examples of library use in open-source applications. Since the conclusion for RQ4 indicates that applications of different size do not adopt libraries at different frequency, we suggest new comers of open-source projects to learn API usage examples by reading the source code of small applications. This can help them obtain better learning effect by avoiding understanding the complex source code of large applications.

E. Threats to Validity

There are four possible threats to validity in this study. The threat to the construct validity is the correctness of library use sites reported by "Understand". Since many studies have produced reliable empirical results by using "Understand" [17], the data in our study can also be considered as acceptable. The threat to the internal validity is that we do not exclude new library constructs from the formerly-established libraries. But according to our empirical data, the new library constructs only account for a relatively small proportion of the use (1.23%) for formerly-established library use. For this reason, our empirical results are still reliable. The first threat to the external validity is that we only use open-source applications to conduct the experiment. The empirical results may not be applicable to industrial applications, as different ways of software development probably make a difference in the adoption of libraries. The second threat to the external validity is that we only investigate standard libraries. The third-party libraries are not included mainly because they are generally considered not as widely used as standard libraries.

V. RELATED WORK

Due to page limitation, here we only discuss a few studies most related to our work. In recent years, more and more researchers have started to investigate the adoption of software libraries in an empirical way. Torres et al. [9] were among the first to study the usage of Java concurrency libraries and they found a list of commonly-used concurrency library constructs. Also, they concluded that medium to large-sized applications tend to use more concurrency constructs. However, this conclusion was drawn by simply comparing the raw number of library use among small applications (1-20KLOC), medium applications (20-100KLOC), and large applications (>100KLOC). By comparison, we use the Spearman's rank correlation analysis method to test the relationship between the size of application and the frequency of library use, which can produce a more reliable result. Another related study was an empirical investigation on C# parallel libraries performed by Okur and Dig [10], who showed that applications with different sizes have different adoption trends. However, they only compare the raw number of library use among different applications instead of investigating the frequency of library use. For this reason, this finding is limited to some extent. Before this study, we have already performed an empirical investigation on the adoption of C++ templates [16], which showed that STL predominates the overall use of library templates. Compared with our previous work, this paper investigates the adoption of C++ libraries at a higher level by focusing on the whole C++ Standard Library, not limited to library templates. The other related work includes the investigation on MPI open-source applications [11], the study on Java library use trend [12], the research on Java API popularity [13], the assessment on third-party libraries [14], and the exploration on third-party component reuse [15].

VI. CONCLUSION AND FUTURE WORK

In this paper, we conduct a study on the adoption of C++ libraries in real applications. The whole study is performed by investigating four research questions regarding the most often used libraries, the difference between the use of the new libraries and the use of the formerly-established libraries, the trend of adopting deprecated library constructs, and the relationship between the size of application and the frequency of library use. By employing inferential statistics, we get reasonable results for the proposed research questions. Based on the empirical results, we give four actionable suggestions, which could help developers, especially the new comers of open-source projects, learn and use libraries efficiently. In the future work, we will investigate more research questions and perform an empirical study on more applications to understand the adoption of C++ libraries in depth.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61170071, 61432001, 91418202, 61472175, 61472178), the National Natural Science Foundation of Jiangsu Province (BK20130014), and the program B for Outstanding PhD candidate of Nanjing University.

REFERENCES

[1] ISO/IEC. Information Technology—Programming Languages—C++. ISO/IEC 14882-1998. 1998.

[2] ISO/IEC. Information Technology—Programming Languages—C++, Second Edition. ISO/IEC 14882-2003. 2003.

[3] ISO/IEC. Information Technology—Programming Languages—C++, Third Edition. ISO/IEC 14882-2011. 2011.

[4] N. Josuttis. The C++ Standard Library: A Tutorial and Reference - Second Edition. Addison-Wesley, 2012.

[5] B. Stroustrup. The C++ programming language – Fourth Edition. Addison-Wesley, 2013.

[6] E. Arisholm, L. Briand, B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1), 2010: 2-17.

[7] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys? In: *Annual Meeting of the Florida Association of Institutional Research*, 2006: 1-3.

[8] M. P. Robillard, R. DeLine. A field study of API learning obstacles. *Emp. Soft. Eng.*, 16(6), 2011: 703-732.

[9] W. Torres, G. Pinto, B. Fernandes, J. P. Oliveira, F. Ximenes, F. Castor. Are Java programmers transitioning to multicore? A large scale study of java FLOSS. *SPLASH*, 2011: 123-128.

[10] S. Okur, D. Dig. How do developers use parallel libraries? *FSE*, 2012: Article No. 54.

[11] C. Marinescu. An empirical investigation on MPI open source applications. *EASE*, 2014: Article No. 20.

[12] Y. M. Mileva, V. Dallmeier, M. Burger, A. Zeller. Mining trends of library usage. *IWPSE-Evol*, 2009: 57-62.

[13] Y. M. Mileva, V. Dallmeier, A. Zeller. Mining API popularity. *TAIC PART*, 2010: 173-180.

[14] S. Blom, J. Kiniry, M. Huisman. A structured approach to assess third-party library usage. *ICECCS*, 2013: 212-221.

[15] W. Schwittek, S. Eicker. A study on third party component reuse in Java enterprise open source software. *CBSE*, 2013: 75-80.

[16] D. Wu, L. Chen, Y. Zhou, B. Xu. An empirical study on the adoption of C++ templates: library templates versus user defined templates. *SEKE*, 2014: 144-149.

[17] SciTools Understand. https://scitools.com/.

[18] C++17. http://en.wikipedia.org/wiki/C%2B%2B17.