A Large-Scale Analysis of C++ Lambdas and Their Capture Lists in Opensource Repositories

Luiz Romário Santana Rios
Universidade Federal do Rio de Janeiro
Rio de Janeiro-RJ, Brazil
luizromario@ufrj.br

Hugo Musso Gualandi
Universidade Federal do Rio de Janeiro
Rio de Janeiro-RJ, Brazil
hugomg@ic.ufrj.br

Abstract

In C++, lambda expressions offer fine control over variable capture: variables can be captured wholesale or individually, and they may be captured by value or by reference. This study is a large-scale analysis of C++ lambda variable capture patterns, aggregating data from over 1600 open-source repositories, since the release of lambdas in C++11. We are the first to measure the frequency of lambda expressions per lines of code, as opposed to total number of lambdas, and are the first to measure the frequency of different variable capture strategies. We confirm that lambda use in C++ is on a growing trend and it is in widespread use: over 70% of projects use lambdas. However, we observed that adoption of lambdas was a slow process; it was only in late 2019 that adoption reached 50%. We also observe substantial variety in variable capture strategies. Out of the repositories that used lambdas, 38.3% favored capturing no variables, 23.5% favored explicit capture, 26.1% favored automatic capture by reference, 8.6% favored capturing only the "this" pointer, and only 3.6% favored automatic capture by value. Overall, implicit capture was twice as popular as explicit capture lists.

Keywords

Programming paradigms and styles, Programming language design and implementation, Run-time environments

1 Introduction

Lambda expressions have been a cornerstone of functional programming for as long as they have been around. Unmanaged languages, those without a garbage collector, have a particular challenge when desiging lambdas: how to manage the lifetime of their closure objects. In C++, which added lambdas in its C++11 standard [12], the programmer can use a *capture list* syntax to control how the external variables referenced in the lambda are stored into the closure [2]. For example, in Listing 1 the bracket-delimited list is the lambda's capture-list, which says that `sum` is captured by reference and `factor` is captured by value.

```
std::vector<double> v;
double sum = 0;
int factor = 2;
std::for_each(
    v.begin(), v.end(),
    [&sum, factor]{ sum += factor; });
```

Listing 1: A C++ lambda with a capture list. Here, the closure stores a reference to `sum` and a copy of the value of `factor`.

Besides the already mentioned explicit listing of captured variables, C++ lambdas can also automatically capture variables that are used inside the body of the lambda. Adding a `&` symbol to the capture list (e.g. `[&, foo]`) will make all variables other than the ones already in the capture list be captured by reference; likewise, adding a `=` symbol to the capture list (e.g. `[=, &bar]`) will automatically capture variables by value (i.e. by copying their value into the closure object). It is also possible to capture the **`this`** pointer, in which case all member variables of the class will be implicitly available inside the body of the lambda.

Since the introduction of C++ lambdas, some studies analyzed lambda usage in C++ code. Uesbeck et al. [13] analyzed the impact of lambdas on developer productivity on various coding tasks and found that lambdas did not seem to lead to significant improvements, and even decreased productivity for novice developers. But studies analyzing real in-production code projects seem to suggest a growing trend of lambda adoption. Lucas et al. [6] analyzed the adoption of modern C++ features in the KDE community[1] and found growing trend in adoption of those features, including lambdas. On the other hand, Kim and Ho [3] also found a growing trend in the use of lambdas with a few select opensource C++ projects, but not a large overall adoption. Hokka et al. [1] analyzed 500 C++ repositories and found that 175 of those used lambdas, with a total count of 24764 lambdas.

In summary, these studies point to C++ lambdas being a source of reduced productivity with limited adoption on one hand, but a growing adoption trend in at least one community on the other. However, the first study is limited to coding exercises instead of real code; the last two analyse a wide range of projects, but don't compare them regarding lambda usage, nor try to find usage patterns; and none of the studies pay attention to how capture lists are used. These works give us good insights about the usage of lambdas in C++, but they don't dive deep enough.

Knowing this, we aimed to understand, in numbers, how lambdas and their capture lists are used in real C++ code. This goal led us to the following research questions:

RQ1 What is the distribution of lambda usage in C++ projects?
RQ2 Which lambda capture strategies are most common?
RQ3 Have lambdas become more commonly used over time?
RQ4 Have lambda capture strategies changed over time?

To answer those questions, we decided to download a large selection of opensource projects with C++ code in them and, for each project, measure its lambda usage throughout history—since the introduction of lambdas in C++ until now.

[1] https://kde.org, accessed in May 30th, 2026

In this article we show the results of the metrics we extracted from this repository selection. Section 2 describes our criteria for repository selection and what measurements we took from the repositories, describing the software used to do so; Section 3 analyzes the measurements we took with some statistics and graphs; Section 4 discusses the results and the limitations of the study; Section 5 talks about related works; and Section 6 concludes the text.

2 Methodology

For this study, we needed a large and diverse selection of real and popular C++ projects. Analyzing real projects gives us insight on what features are used by developers when they need to solve problems. There has to be some basic guarantee of quality as well, so that we can avoid measuring throwaway code as much as possible. We had to decide where to download these projects from and how to select them. We also had to decide how to measure and classify lambda usage and how traverse the history of the projects. In this section, we will present and motivate the choices we took for conducting our study. Our scripts were all written in Ruby, except where stated otherwise, and they are available as artifacts (see Artifact Availability).

2.1 Project Selection

Manually selecting projects would be unfeasible, so we needed some sort of repository of public projects. The most obvious choice is GitHub[2] so we went with it. However, to add a little more variety, we added some more projects from GitLab's flagship server[3].

We wanted to ensure no personal bias was involved in the choices of projects for the study; on the other hand, we wanted to avoid irrelevant projects (single-user projects, throwaway repositories, etc). So we decided to fetch all projects automatically by querying the APIs of the repositories with custom-written scripts.

For GitLab, we selected all C++ projects with more than 10 stars. GitLab projects are much less starred than GitHub projects, so this threshold excludes most C++ projects. The resulting query returned 690 projects.

For GitHub, we queried for all C++ projects in descending order. GitHub's API can retrieve at most 100 projects per query and, after some consecutive queries, it denies further requests. In our case, before GitHub started deyning our requests, we retrieved 1000 projects. We found that number sufficient.

That amounts to a total of 1691 fetched project repo URLs. However, we didn't extract the metrics of all of those projects, mainly because a few of them have the same name (in which case we discard the least popular duplicates), but also because a couple of the repositories had issues that prevented them from being properly cloned. Overall, 1665 projects were analyzed.

All repositories of the projects were downloaded between the 26th and the 31st of July 2026.

2.2 Counting and Classifying Lambda Usage

Since we are analyzing the projects histories, we are taking the same measurements for several commits in the repository. The following process is repeated for each repository, at each measured commit.

We start by counting every lambda in the project's C++ code. To do that, first we find all source files with the most common C++ filename extensions: `h`, `hh`, `hpp`, `hxx`, `cpp`, `cxx`, `cc`, `C`, and `tpp`. Then, we wrote a tree-sitter[4]-based parser in Rust to find all lambda expressions in all source files of a given directory. The query we used to find the lambdas was:

```
(lambda_expression captures:
  (lambda_capture_specifier) @cap)
```

Theoretically, just `(lambda_capture_specifier) @cap` would work, but we found that the more complete expression is more reliable and generates less false-positive matches.

Then we group theses matches according to a classification based on how the capture list is used. In order to simplify the number of categories and the filtering of the results, we decided to use the simplest possible capture list configurations as categories and everything remaining as one more category. The following are the chosen capture strategy categories based on our needs:

(1) Auto-capture by reference and nothing else (i.e. `[&]`)
(2) **`this`** capture and nothing else (i.e. `[this]`)
(3) Auto-capture by value and nothing else (i.e. `[=]`)
(4) Empty capture list (i.e. `[]`)
(5) Explicit capture—i.e. anything that doesn't fall into any of the previous categories (e.g. `[&, this, foo]`)

Preliminary measurements showed that this categorization produces decent results: all categories have relevant occurrence numbers.

We use regular expressions to classify the matches, one for each of the first four categories: `\[&\]`, `\[this\]`, `\[=\]`, and `\[\]`, respectively. What remains is grouped in the fifth category.

We could have created a separate tree-sitter expression for each capture strategy category for better accuracy, but we chose this approach for simplicity—only one tree-sitter query expression and four small regexes are needed, instead of five full blown tree-sitter query expressions.

After finding the lambda usages, we count each category of lambda that is used as well as the C++ linecount of the project—using a tool called tokei[5] for fast counting.

Finally, we store the results. First, we generate a separate file listing all the matches that were found, grouped by category, for later validation; then, we append the following metrics at the end of a summary CSV file:

- The SHA hash of the commit
- The date of the commit
- Total lambda count
- Auto-ref lambda count
- **`this`** lambda count
- Auto-val lambda count
- Empty capture lambda count
- Explicit capture lambda count
- Total C++ line count

[2] https://github.com
[3] https://gitlab.com
[4] https://tree-sitter.github.io/, accessed in May 30th, 2026
[5] https://github.com/xampprocky/tokei

Selecting which commits to analyze. This process is repeated backwards in the git history, starting from the most recent commit. Given the sheer scale of the study, taking these measurements for each commit in the repository is not viable. We attempted to develop a diff-based solution for these metrics, but some preliminary measurements showed that the metrics drifted too much from the real values to be useful. So we decided to pick one commit per git history month and fully calculate the metrics at each one. This proved to be sufficient, as we're studying a 15-year period (i.e. 180 months).

2.3 Metrics and Statistics

After extracting the raw lambda usage data from the repositories, we have to normalize our measurements so that we can compare projects of different code sizes and account for the change in code size within the same project when analyzing its history. The main metric we analyze throughout this study is *lambdas per ten thousand lines* (lams/10kloc). We use this metric in statistics, in graphs, and to compare different C++ codebases. We use this unit of measurement for the total lambda count and also for each of the five categories.

Choosing to measure lams/10kloc was a simple matter of readability. Our goal was to measure the proportion of lambdas in relation to code size, but if we chose lamdas per line as our unit, the resulting numbers would be too small. Our unit just takes the original ratio and multiplies it by ten thousand. Any other multiplying factor would have reached a similar result, but preliminary measurements indicated that most projects would be in the order of magnitude of 10 to 100 lams/10kloc, which we found to be suitable numbers for better readability.

Our first analysis compares the metrics of the current version of each of the projects at the time of writing. Focusing on a single point in time at first gives us a basis for the later history analysis and it allows for a more detailed anaylsis. We calculated the lams/10kloc distribution for the total number of projects and for each category and grouped projects by most used capture strategy.

Next, we needed some statistics for all projects over time. One problem we had to solve is that, even though we took one commit per month in each repository, the commits we picked for any given month were not from the same date for all repositories. So, for example, if we wanted to plot the average lams/10kloc for all repositories, we wouldn't be able to just iterate over date because certainly each date would only feature the measurements for a few of the projects. To solve this, we merged all the data from the repositories over history into a single matrix where the first column contains the dates of all of the commits in all repositories, sorted by date, and each one of the other columns contains the metrics of one of the projects. To fill the blanks—that is, to provide entries for the dates where a given project might not have a commit—we look for the closest past commit and take the value from there.

We chose to take our values from the the cumulative distribution curve of lams/10kloc for all projects—for the total lambda count as well as the categories. For example: to analyze the evolution of lams/10kloc over time, one of the values we took from the data, for each point in time, was the value of lams/10kloc with a cumulative distribution of 75%—that is, the maximum value of lams/10kloc after you exclude the 25% top lams/10kloc. By analyzing the cumulative distribution, we are able to find value ranges where many different percentages of projects sit within.

All graphs in this paper were generated with a Python script using matplotlib[6].

2.4 Counting File Duplicates

The presence of duplicate files across repositories can pose a problem for large-scale analysis of opensource projects. To estimate the impact of file duplication, we examined file hashes for all C++ files in the most recent commit of all repositories. We compute the total number of unique hashes among all C++ files as well as the number of unique hashes among C++ files that contain lambdas.

3 Results

We analyzed a total 1665 repositories, ranging from less than a dozen lines of code to millions. In this section, we begin by comparing them in their most recent revision as of the time of writing. Then we show the evolution of lambda usage over time.

3.1 Current Metrics

In their most recent revisions, we found a large variability of lams/10kloc in the projects. Of the 1665 repositories, 403 (24.2%) feature *no* lambda use (i.e. exactly 0 lams/10kloc). The maximum lams/10kloc overall is 41 thousand. However, the vast majority of projects have a lams/10kloc that is, at the very most, around the hundreds (Table 1 and Figure 1).

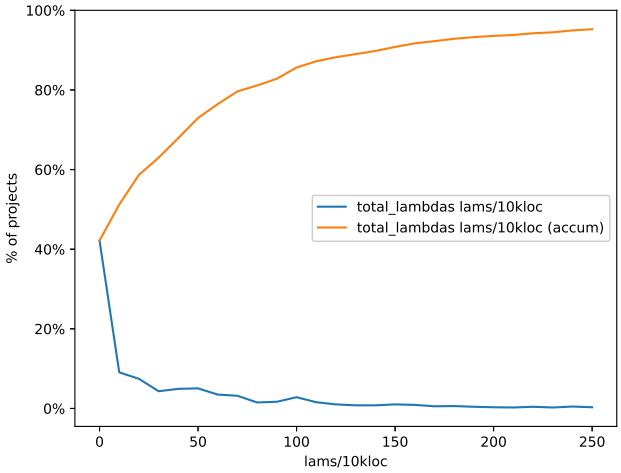

Figure 1: Distribution of projects by total lams/10kloc. The horizontal axis is lams/10kloc and the vertical axis is the percentage of projects.

Values outside the 0-1000 lams/10kloc range seem to be outliers and are likely distorted by a low linecount. For example, the aforementioned 41 thousand value comes from a project with only 30 lines of C++ code. One evidence of this is that, if we remove all projects with less than ten thousand lines of C++ code, we're left with 1020 projects, and the overwhelming majority of them have

[6] https://matplotlib.org/, accessed in June 1st, 2026

Table 1: Table of maximum lams/10kloc for a given percentage of projects with the least lambdas (with the amount of projects in braces).

% of projects	Overall	Over 10,000 lines
75%	65.9 (1250)	68.5 (766)
90%	151.3 (1500)	125.4 (919)
95%	252.2 (1583)	181.9 (970)
99%	1048.2 (1649)	483.4 (1011)

no more than 182 lams/10kloc (Table 1). We can now answer our first research question:

RQ1 What is the distribution of lambda usage in C++ projects?
Answer 24% of all studied projects use no lambdas. Among the projects that do use lambdas, the overwhelming majority of them (95%) have no more than 182-252 lams/10kloc (depending on code size).

Table 2 shows the maximum lams/10kloc for some given percentages of projects. The data indicates that the most proeminent categories of capture strategies are automatic capture by reference, explicit capture, and no capture, with the first two being essentially tied and the last one fairly ahead of them; capturing **`this`** is not as commonly used, but it has its uses; by far, automatic capture by value is the least used—the majority of projects barely even use it.

Table 2: lams/10kloc with 75-99% cumulative distribution per category

	`[]`	`[&]`	`[&foo, bar]`	**`[this]`**	`[=]`
75%	18.7	11.8	12.7	3.3	0.6
90%	45.9	41.1	37.9	15.3	3.4
95%	98.2	70.1	62.6	27.5	10.1
99%	399.2	284.8	274.7	117.0	63.8

However, even though the explicit capture is one of the most popular capture strategies, implicit capture (i.e. the other categories combined) is the dominant capture strategy, with around 70% of all lambda usages.

This result is confirmed if we group the projects by most popular capture strategy (Table 3). Again, not capturing anything, auto-capturing by reference and explicitly capturing are the most popular strategies, with no capturing taking the lead by a good margin. Also noteworthy is that not using lambdas at all has about the same popularity as the most popular capture strategies.

We can now answer our second research question.

RQ2 Which lambda capture strategies are most common?
Answer Not capturing anything (`[]`), automatic capture by reference (`[&]`), and explicit capture (`[&foo, bar]`) are the most used capture strategies, being preferred, respectively, by 38.3%, 26.1%, and 23.5% of all projects that use lambdas. All implicit capture strategies combined dominate explicit capture by a large margin (76.6% vs 23.5% preference). Automatic capture by value is by far the least preferred strategy: it is only the most used in 3.6% of all projects.

Table 3: Projects grouped by most used capture strategy (percentage of projects does not account for projects not using lambdas).

Most used capture strategy	# of projs	% of projs
Empty capture list (`[]`)	483	38.3%
Not using lambdas	403	–
Auto-capture by reference (`[&]`)	329	26.1%
Explicit capture (ex.: `[&foo, bar]`)	296	23.5%
Capture **`this`** (**`[this]`**)	108	8.6%
Auto-capture by value (`[=]`)	46	3.6%

3.2 Lambdas Across History

To visualize how the distribution of projects by lambda usage evolves over time, we measured the maximum lams/10kloc over time for 30%, 40%, 50%, 60%, 75%, 90%, 95%, and 99% of the projects (Figure 2, not all lines are shown). These measurements revealed some important facts about lambda adoption.

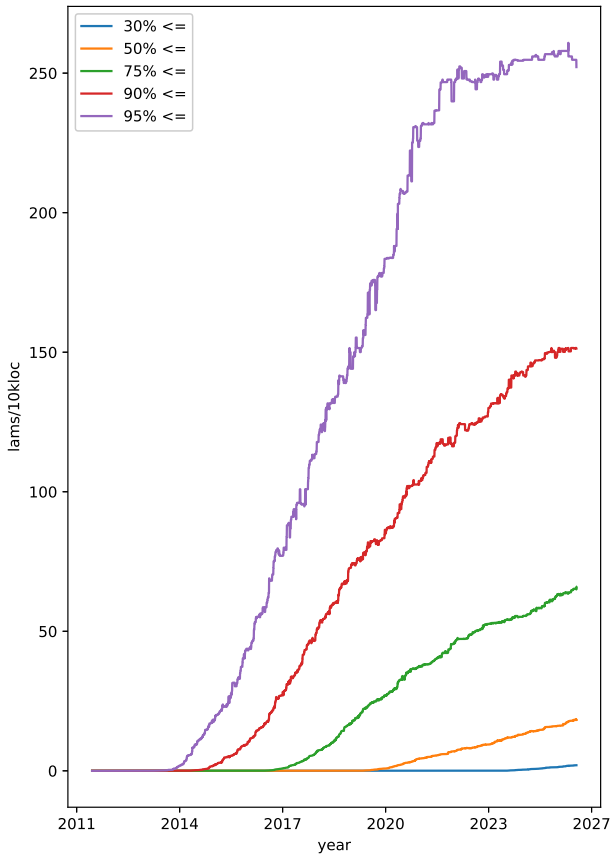

Figure 2: Maximum lams/10kloc for 30%, 50%, 75%, 90% (above), and 95% (below) of projects with the least lams/10kloc.

It took a very long time for lambdas to be widely adopted. 75% of the projects did not use lambdas at all before late 2016, nearly five

and a half years after the initial inclusion of lambdas in C++. Even then, the initial adoption after this point was extremely limited, not even surpassing 1 lam/10kloc. A little over year later, in early 2018, the maximum lams/10kloc for 75% of the projects was still at around 3 lams/10kloc. It was only a year later after that, in January of 2019 (7 years after the initial release of C++11), that these projects reached a 10 lams/10kloc maximum. Even if we include projects that started using lambdas the earliest, 90% of projects never used lambdas earlier than July of 2014 and 95% no earlier than May of 2013. The point in time where more than half of the measured projects used lambdas was only in late 2019, eight and a half years after C++11, and only in mid-2024, nearly thirteen years after the release of C++, we could confidently claim there was widespread adoption of lambdas, with 70% of all projects adopting lambdas in some amount.

Adoption growth. Across all projects, in all distribution levels, after the point in time where lambdas are initially adopted, we see an steady upwards trend of lams/10kloc. The upwards trend remains visible until the time of writing for all distribution levels with no noticeable deceleration for all distributions up until 60%. For 75% of the projects, a very slight deceleration is noticeable from around 2023 onwards. Looking at the 90% data, the deceleration is much more noticeable, especially around 2024, when the 140 maximum lams/10kloc mark is crossed, though there's not a clear plateau. However, on the 95% data, there's a clear plateau starting at around 2022, when the 250 maximum lams/10kloc mark is crossed.

Capture strategy across history. There does not seem to be any overall change in capture strategy across the board (Figure 3). In every level, the same capture preferences we showed in Subsection 3.1 remained largely the same from the moment lambdas started getting adopted right until the time of writing. Moreover, it looks like the most popular strategies are also the first ones to be adopted, meaning that at no point in time we see one capture strategy meaningfully overtaking the other.

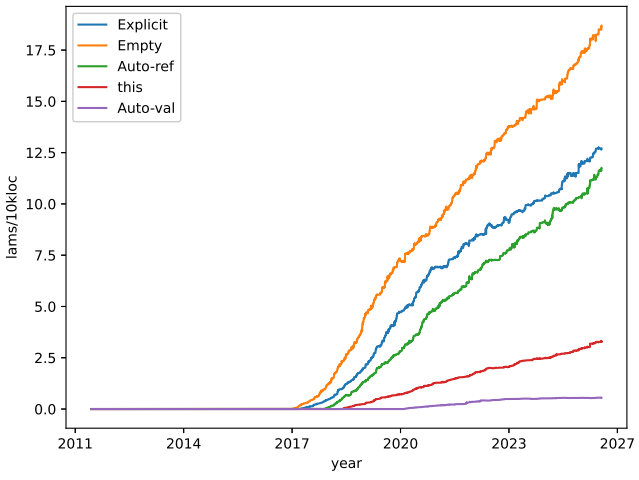

Figure 3: Maximum lams/10kloc of 75% of the projects over time for all capture strategy categories.

With these results, we can now answer our final research questions:

RQ3 Have lambdas become more commonly used over time?
Answer Even though widespread adoption of lambdas took around a decade to happen, it did happen and lambda usage and adoption have been on a growing trend ever since. Some of the data suggest a deceleration might have started, but only for projects with a very high lams/10kloc count.

RQ4 Have lambda capture strategies changed over time?
Answer The most common strategies are still the same.

3.3 File Duplicates

We analyzed file duplication in aggregate, without separating it by repository. Our dataset has 2,555,092 C++ files in total, which have 1,893,692 unique hashes. This means that 661,400 (26%) of those files are duplicates. We observe that many duplicates are small files. For example: the empty file appears 2013 times and there are 326 occurrences of a file containing a single "1" character.

If we restrict our analysis to C++ files that contain lambdas, there are 213,609 files in total, which have 188,807 unique hashes. This means that there are 24,802 duplicates (12%).

4 Further Discussion

The results show that, on one hand, lambda adoption is steadily increasing and we can now claim that there is widespread adoption of lambdas across opensource C++ projects; on the other hand, it took a around a decade for lambdas to be widely adopted and a large portion of them still don't use any lambdas.

The late adoption of lambdas in C++ could be partially explained by the lack of compiler support at the time of release. For example, both GCC and Clang released their first fully C++11-compatible releases in mid-2013, almost two years later than C++11 . Moreover, operating systems and companies might take years to adopt new compiler versions after they are released. However, these facts alone can't explain why, up until 9 years after the release of C++11 and 7 years after the first compilers to support C++11 were released, most C++ projects still did not adopt lambdas at all. This is a question we are not able to answer here, but most likely there are other factors that need to be studied.

There is a well-defined (albeit wide) range of lams/10kloc that most projects currently sit within—which is from 0 to around 200 lams/10kloc. Given the growing trend in lams/10kloc, one could imagine that this upper limit may go up as well. However, the deceleration in the growing trend seen at Section 3.2 could indicate that, even if the upper trend keeps going steady in the future for most projects, projects may still firmly within the 200 to 250 maximum. This could suggest that, after a certain amount of lams/10kloc is crossed, there's a tendency of projects to stabilize the usage of lambdas in relation to the code.

We noticed that all implicit capture strategies combined are preferred over explicit capture. We cannot know for sure what leads to this preference, but notice that, in other languages that feature lambdas and have garbage collection, external variables are always implicitly "captured"—i.e. the closure is implicitly constructed. This suggests that the current usage patterns of capture lists in C++ tends to mirror lambdas in functional languages.

4.1 Study Limitations

The way we selected repositories, without any curation, means that a few undesirable selections, which could be filtered out by a human, remain. For example: our selection might include repeated repositories that go by different names. One glaring example of this is that our selection includes the Chromium web browser, but it also includes a project called chromium-sdk, which is chromium-based and is nearly identical to Chromium. This similarity is evidenced by their metrics: they both have around 18 million lines of C++ code, 32.3 lams/10kloc in total, 13.6 lams/10kloc with no capture, etc. This code repetition might distort the overall final results.

A common practice that might also cause code repetition between repositories is vendoring. Some projects choose to manage some of their dependencies by vendoring them, i.e. including the code of the dependencies entirely inside the repository in a subdirectory dedicated to dependency code (usually called `dependencies`, `third_party`, `contrib` or other similar names). This poses a problem to the current study, because the code of the dependencies can sometimes be the majority of the code in the repository. An example of this problem is the FreeBSD project. This project has 43,1 lams/10kloc, a decent amount of lambda uses for a 3.5-million-line codebase. However, looking at the actual instances of lambda usage that were found, they all come from vendored LLVM code.

The problem of code duplication has been studied by Lopes et al. [5], who encountered up to 73% of file duplication in their large-scale analysis of GitHub repositories. This figure is much higher than the one we encountered: 26% of copied files overall and 12% of copied files that use lambdas. It is possible that the difference is due to the amount of repositories that were analyzed. They analyzed 364 thousand repositories—more than one hundred times more than we did. Their dataset contains a longer tail of unpopular repositories and it is possible that, because of this, it could contain more repositories that are forks and clones.

In any case, our measurements do include duplicate files, which we did not filter out. However, this duplication likely has a smaller impact than the one encountered in the dataset of Lopes et al. [5].

There's also some ambiguity in determining what is C++ code and what is C code, given the nature of C++. For instance, the `.h` extension is widely used for headers with C++ code, even when the `.hpp` extension, which is exclusive to C++ headers, exists. This stems from the fact that C++ is nearly a superset of C and most C code can be compiled as C++ code. This caused an interesting situation with the GCC project. Looking at GCC's lams/10kloc over time, a massive drop can be noticed around 2021 (Figure 4). This happened when several `.c` source files changed extensions to `.cpp`[7], bumping the total C++ linecount of the project from 700 thousand lines to 2.3 million lines. No actual change in the code happened at that point, it was still the original C code without any C++ functionality. However, a simple filename change caused a dramatic change in the metrics.

[7]Commit hash: `5c69acb32329d49e58c26fa41ae74229a52b9106`

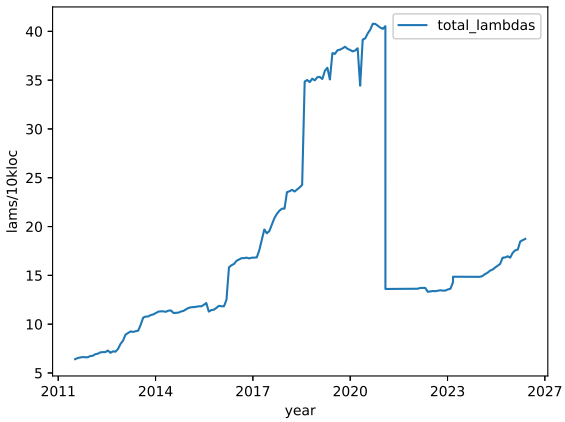

Figure 4: GCC's lams/10kloc over time.

5 Related Work

Due to growing demand for functional programming patterns among programmers, they have been increasingly adopted in imperative languages [10], which requires adapting those functional features to the design of the imperative languages.

Uesbeck et al. [13] studied how lambdas impact the productivity of developers. The way they conducted the study was giving a few programming tasks to be completed by C++ developers from various skill levels, ranging from university freshmen to professionals, and compared the amount of compiler errors during development when using lambdas versus when using iterator-based solutions. Kim and Ho [3] studied the adoption of lambdas and templates in C++ codebases over time and found that templates were much more widely adopted than lambdas. Hokka et al. [1], who measured lambda usage in 2021, found that, by that point, lambda usage, while relevant, was still not widespread, which corroborates our findings. Finally, the 2024 study by Lucas et al. [6] found a trend in the widespread adoption of modern C++ features in the KDE community (including lambdas) right around the time period where we identified lambdas first got widespread adoption overall.

Rust is another language with lambdas, but no garbage collection, so it faces the same challenges about closure management that C++ does—though it approaches the problem in a different way (this distinction is briefly explored by Rios and Ierusalimschy [8]). Wolff et al. [14, ch 5.3] made a preliminary analysis of 200 Rust projects and found that 90% of lambda usages do not have mutable captured state. Levy et al. [4] found that Rust lambdas pose limitations for embedded applications—namely, the language's ownership model is overly restrictive and prevents otherwise safe uses of lambdas for asynchronous programming.

Some other large-scale studies about lambdas in opensource projects have been found. Nielebock et al. [7] analyzed the use of lambdas for concurrent object-oriented code in almost 3 thousand opensource projects written in C++, C#, and Java. Sangle and Muvva [11] classified 19 different kinds of lambda usage in 760 Python projects.

6 Conclusion

In this study, we measured lambda usage in C++. We analyzed 1665 opensource C++ projects, extracted lambda usage metrics from all of them across their git history, and employed a new metric (lams/10kloc) that allows differently sized projects to be compared. Most (95%) projects have at most 250 lams/10kloc.

Our analysis identified the most popular lambda capture strategies: the empty capture list (`[]`, 38.3%), automatic capture by reference (`[&]`, 26.1%), and explicit capture (e.g. `[&foo, bar]`, 23.5%). Most (76.6%) projects prefer to capture variables implicitly.

We also measured the adoption of lambdas over time. On one hand, it took some time for lambdas to start getting widespread adoption among C++ projects, and, on the other hand, lambda adoption has been steadily growing. Also, over time, we found out that the popularity of capture strategies has remained largely the same since the beginning of lambdas in C++ until now.

A limitation of this study is that we measured, but not excluded, duplicate files. According to our analysis, out dataset contains 26% of duplicate C++ files and 12% of duplicate C++ files with lambdas.

Artifact Availability

All data used in the article, as well as all scripts used to generate them, are available at the Open Science Framework platform [9].

Acknowledgements

Funded by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) – Project Code 403601/2023-1. The machines to run experiments were kindly provided by Instituto Tecgraf/PUC-Rio and LC3 Lab at UFRJ.

References

[1] Heidi Hokka, Felix Dobslaw, and Jonathan Bengtsson. 2021. Linking Developer Experience to Coding Style in Open-Source Repositories. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 516–520. doi:10.1109/SANER50967.2021.00057

[2] Jaakko Järvi and John Freeman. 2010. C++ lambda expressions and closures. *Science of Computer Programming* 75 (2010), 762–772. https://doi.org/10.1016/j.scico.2009.04.003

[3] Donghoon Kim and Loc Ho. 2020. The Reaction of Open Source Projects to C++ Templates and Lambdas: An Empirical Replication Study. In *International Conference on Software Engineering and Knowledge Engineering*. https://api.semanticscholar.org/CorpusID:221193864

[4] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems* (Monterey, California) *(PLOS '15)*. Association for Computing Machinery, New York, NY, USA, 21–26. doi:10.1145/2818302.2818306

[5] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages. doi:10.1145/3133908

[6] Walter Lucas, Fausto Carvalho, Rafael Campos Nunes, Rodrigo Bonifácio, João Saraiva, and Paola Accioly. 2024. Embracing modern C++ features: An empirical assessment on the KDE community. *Journal of Software: Evolution and Process* 36, 5 (2024), e2605. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2605 doi:10.1002/smr.2605

[7] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. 2019. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empirical Software Engineering* 24, 1 (01 Feb 2019), 103–138. doi:10.1007/s10664-018-9622-9

[8] Luiz Rios and Roberto Ierusalimschy. 2025. Approaching closures in unmanaged languages: a comparison between C++, Rust, and Swift. In *Anais do XXIX Simpósio Brasileiro de Linguagens de Programação* (Recife/PE). SBC, Recife, PE, Brasil, 97–99. doi:10.5753/sblp.2025.13162

[9] Luiz Romário Santana Rios and Hugo Musso Gualandi. 2026. A Large-Scale Analysis of C++ Lambdas and Their Capture Lists in Opensource Repositories: Artifacts for the SBLP 2026 Article. https://doi.org/10.17605/OSF.IO/A27VE. doi:10.17605/OSF.IO/A27VE

[10] Luiz Romário Santana Rios and Roberto Ierusalimschy. 2019. *A survey of function values in imperative programming languages.* Master's thesis. Pontifícia Universidade Católica do Rio de Janeiro. doi:10.17771/PUCRio.acad.47283

[11] Shubham Sangle and Sandeep Muvva. 2019. On the use of lambda expressions in 760 open source Python projects. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1232–1234. doi:10.1145/3338906.3342499

[12] Bjarne Stroustrup. Last edited in 2016. C++11 - the new ISO C++ standard. https://www.stroustrup.com/C++11FAQ.html Accessed in June 2nd, 2026.

[13] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 760–771. doi:10.1145/2884781.2884849

[14] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 145 (Oct. 2021), 29 pages. doi:10.1145/3485522