Document number:	P1231R0	
Audience:	WG 21, LEWG, EWG	
Date:	2018-10-08	
Reply to:	JC van Winkel	(jcvw@google.com)
	Christopher Di Bella	(cjdb.ns@gmail.com)

Proposal for Study Group: C++ Education

Introduction

C++ is a popular language in industrial software engineering, but this popularity is not reflected in education. Most computer science curricula do not acknowledge programming using C++. This is problematic for companies who see more and more 'C++ illiterate' candidates in their hiring pipelines.

This paper proposes to create a Study Group to help improve education in C++ in academia, consulting, on-site, in-house training, online tutorials, or otherwise, to establish guidelines for teaching C++.

Status

Few schools and universities have a C++ curriculum. This means that new hires in companies using C++ must teach their employees C++ (or hand them a book and throw them in at the deep end). It would be advantageous for companies and the community to trust new hires to know how to write software using C++ from the point of hiring.

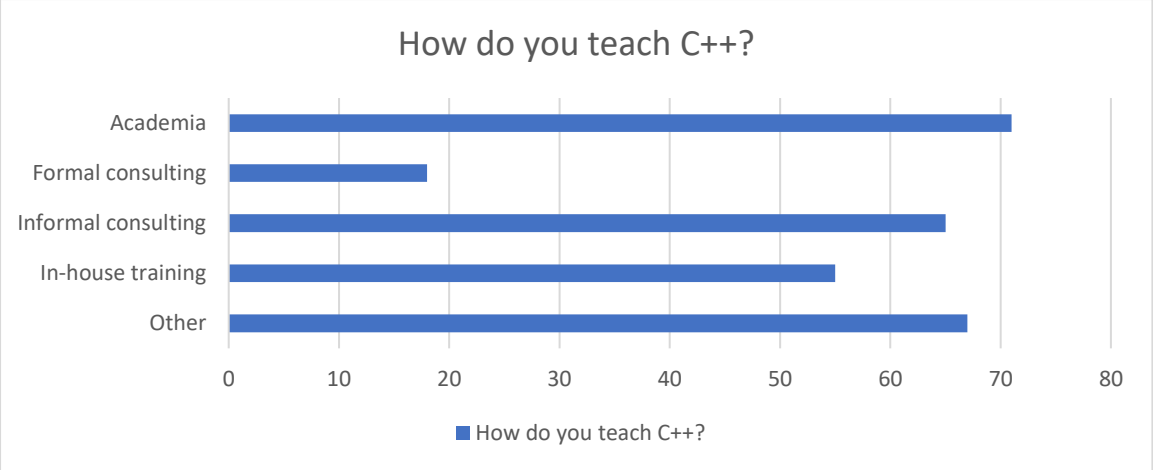
If students do get C++, they oftentimes get small projects to implement that are dismissed the moment they have finished the course. But when they start working in companies with large existing code bases, they rarely work on small, self-contained projects from scratch. Most often they are required to work on large existing codebases, and the ability to read other people's code is an important skill for any hired engineer.

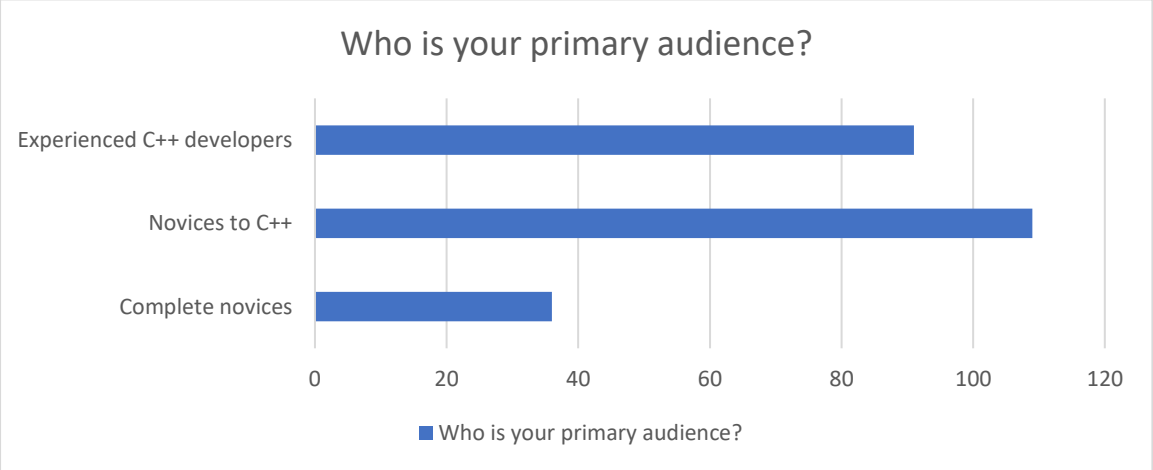
We also see that the way people teach C++ is sometimes still a "C then C++" style instead of using C++ idioms from the start. Bjarne's Programming -- Principles and Practice Using C++ (Swan book) shows that this is not necessary; and in a CppCon 2015 talk, Kate Gregory discusses how it is harmful to start teaching C++ by starting with C (Gregory, 2015). In a survey we conducted, we see 29% of the C++ educators do this (fully or partly). Of these instructors, only 36% have read Design and Evolution of C++, and hence have some historic perspective on why certain things in C++ are the way they are.

Teaching C++ survey

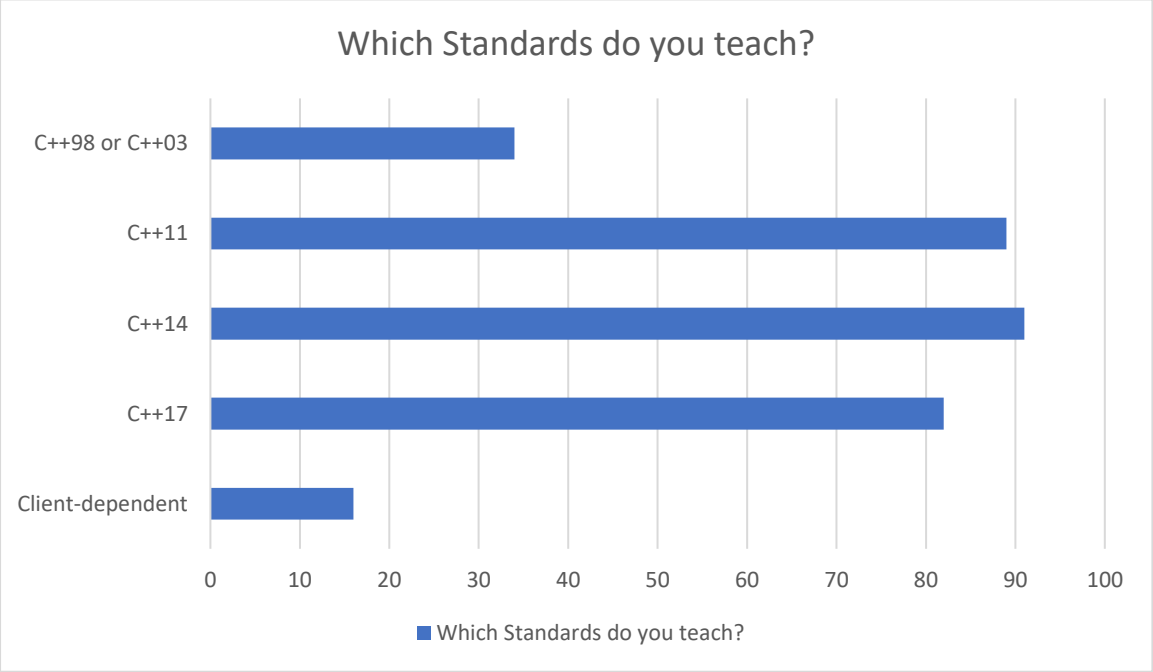
In preparation for the CppCon 2018 talk titled '*How to Teach C++ and Influence a Generation*' and for this proposal, we surveyed C++ educators to determine how C++ is currently taught. At the time of writing, approximately 150 parties responded.

C++ is mostly taught through academia, followed by informal teaching and internal training, with primary audiences majorly being novices to C++ and experienced C++ developers.

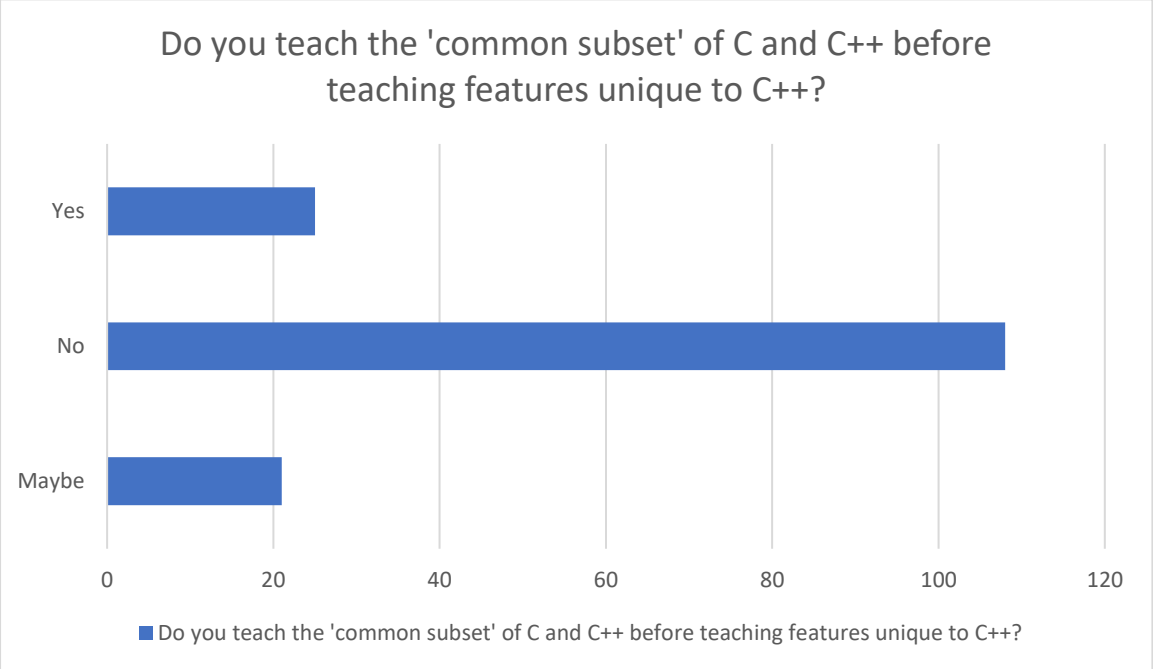




All International Standards appear to be well-represented in education, with C++14 being taught most, and C++98 and C++03 being taught least.



One concern that the authors have is that a portion of unknown size of educators advocate for teaching some subset of C++ that intersects with C, before approaching the parts of C++ that are appropriate for teaching first (e.g. `std::vector<T>` vs `T[]` and `std::string` vs `char const*`).



Goals

Our goal is to improve the quality of education of C++ received by software developers[1], so that when writing software using C++, they can correctly leverage the programming language and its ecosystem to write correct, maintainable, and performing software.

The major formal educational facilities include academia, professional consulting, internal training, and on-site training. To achieve this goal, we see the necessity for several curriculum guidelines for various levels of expertise and application domains. When teaching, it is crucial to know and present appropriate material to the audience. We believe the group should address guidelines tailored to several levels of expertise. We recommend exploring the following archetypes[2] for potential curriculum guidelines:

- Complete programming novice
- Experienced programmer; novice to C++[3]
- Programmers adept in some domains related to C++
- Experts in some domains related to C++

The guidelines may (or may not) have specialised sections for application domains, such as heterogeneous programming, AI, finance, gaming, embedded applications, etc.

---

[1] The term software developer includes people with formal education in any of computer science, software engineering, and computer engineering, as well as those without (e.g. self-taught, took an elective in a different degree, high school student, etc.). A software developer may be a paid role or a non-paid role.

[2] This does not preclude, nor does it suggest audiences or age groups. This should be a separate discussion point throughout the evening session or even the study group.

[3] This refers to programmers who are already experienced with some programming language *other than* C++ (e.g. C, Objective C, Java, Python, C#, Rust, D, Swift, etc.).

Not only the C++ language itself is important but also the environment it runs in can be in scope. That could include the toolchains, testing, support libraries and so on.

We also want to foster a culture where every new proposal submitted to LWG/CWG is accompanied by tutorial material on how, why, and when to use said new feature. The 'Tony Table' idiom employed by the committee can be considered a starting point for educational guides. It is also good evidence for cultures being established in WG21.

As C++ progresses, we need to ensure that the established curriculum guidelines remain relevant to the active International Standard. For example, a curriculum guideline published after C++20 is released would consider concepts and lazy range adaptors, and when C++23 is published, the guidelines should be updated to consider the content that is released with said International Standard (potential candidates may or may not include eager range adaptors and reflection).

Non-goals

This paper does not aim to suggest that an education study group confront the following issues:

- Creating course materials - we aim to have guidelines, so people can create their own curricula, following best practices set in the guidelines, but using their own judgement for deciding how they ought to embed the curriculum in their specific environment (e.g. by having relevant examples and exercises or pointing to book support pages like those for the Swan book).
- Source-style issues including, but not limited to east `const` vs west `const`, bracing, tabs vs spaces, or any other material that a formatting tool can address. The appropriate point of discussion for this topic should be around integrating formatting tools into guidelines, so that this immediately becomes a personal, stylistic, non-issue.

Tasks

To assemble a set of curriculum guidelines, we solicit advice from teachers in high school, universities, consultants, and people involved in teaching in companies.

Where and when to meet

Similarly to SG14, an education study group could potentially meet via telecon, at WG21 meetings, and if sizeable enough, at conferences such as CppCon. We respect that participants are busy individuals; to ensure that participants have enough time to action work, we propose that the study group meet once per month.

Study group chair

The authors would like to open the chair to a fair vote, and welcome interested parties to volunteer for chairing the study group. The authors are willing to jointly chair the group.

Exit criteria

The authors believe that there should not be any exit criteria, as the C++ release cycle is three-yearly, with the occasional Technical Specification or Technical Report publication. As such, we firmly believe that any education group should persist, so that the state of education is maintained alongside the current release of C++.

Recommendations

The authors would like to make the following recommendations for the following talking points, should they be of interest to the study group.

Exercises

Exercise sets and projects are an important part of learning to program: they allow for experimentation and feedback. Although it is a non-goal for the proposed study group to provide explicit exercises, we encourage each curriculum guide to provide material and examples that help educators develop their own exercises or provide access to resources with pre-made exercise sets. We recommend that curricula include laboratory-style exercises for practice and reinforcement, and projects for comprehension.

It is our belief that real-world projects are scarcely covered in course materials. Course projects that exist in a vacuum do not provide appreciation for how industrial projects exist. It is not appropriate for a first or second computer science course to expose novices to large-scale projects, but it is important for students to learn how to eventually read existing code. We would like to request that the study group perform active research into determining when it becomes appropriate for a course to substitute one course-only project with a large-scale open-source project that requires understanding and patching[4].

Preparation

Preparing a course requires significant effort on the teacher's part, both at the organisational level (i.e. for the whole course on Beginner's C++), and at the individual class level (i.e. planning for next week's specific class on object lifetimes). This section focuses on the former, rather than the latter, because it is critical that an encouraging environment is provided to all students to thrive, and because planning for an individual class should, at most, include revision of topics and working out how to address topics.

Philosophy of C++

When learning and teaching, it is important to convey – but not necessarily directly expose – the philosophy of C++. Understanding the design decisions behind aspects of C++ can be garnered in the following ways[5]:

1. Read '*Design and Evolution of C++*'.
2. Read WG 21 proposals that include significant motivation.
3. Read the WG 21 minutes for a proposal.
4. Attend WG 21 meetings.
5. Discuss the design of a feature with its authors.

Just as P0939 recommends that WG 21 proposal authors read '*Design and Evolution of C++*' before submitting a contributing to advancing C++, we recommend that educators read at least the first half of '*Design and Evolution of C++*', Chapters 1 and 22 of '*Programming -- Principles and Practice Using*

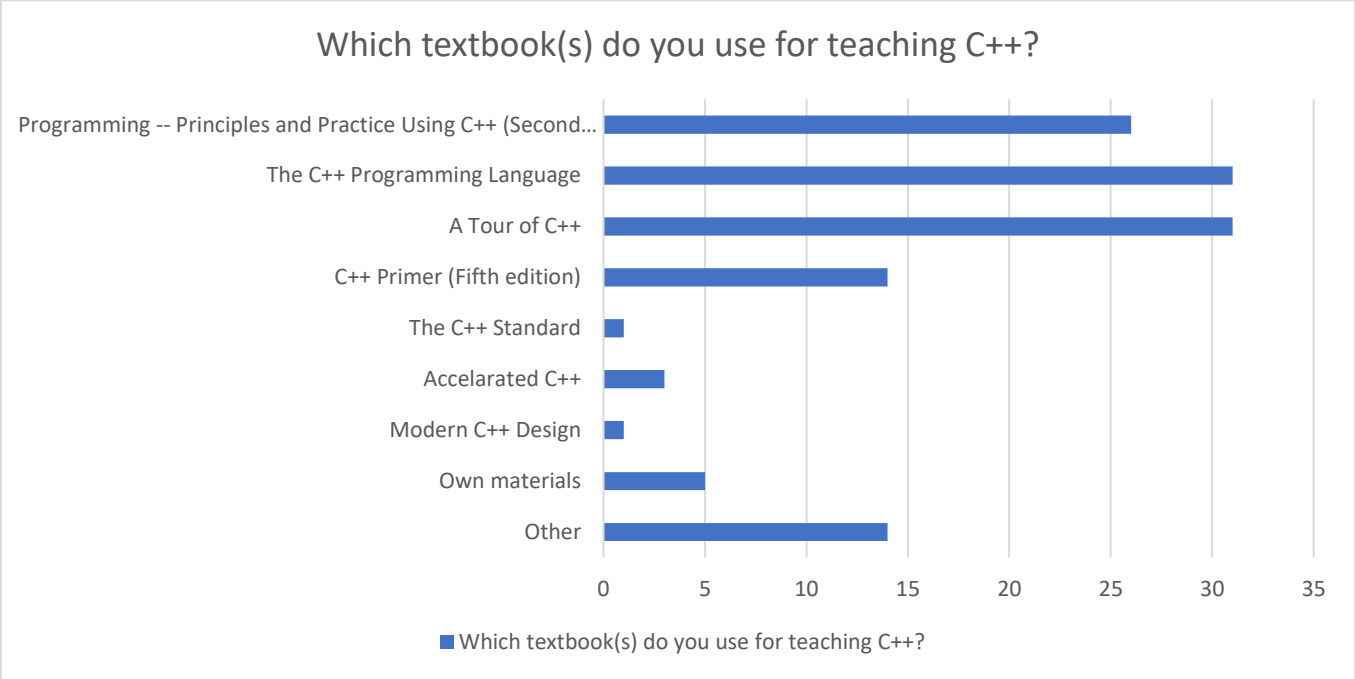
[4] It has been noted that helping students to appreciate what 'good code' and 'bad style' are would be a boon to student outcomes (see below). Additionally, there are no universal tools in any ecosystem, and it isn't possible to provide a census. Informing teachers of different tools, similar to Jason Turner's C++ Best Practices GitHub repository would be a boon for all teachers.

[5] Of these options, only the first is readily accessible to most people: many WG 21 proposals go through several revisions, and are often targeted at members of WG 21, who are the primary audience. Reading WG 21 minutes is reserved only to those who are members of national bodies or have attended a committee meeting, and attending a WG 21 meeting is even less accessible, as it requires being allowed to attend week-long meetings that are often exorbitantly expensive. Finally, while discussing design features with individuals might be accessible over email, but this requires a person to track down the author, send them an email, get a response, and engage in dialogue. This involves a lot of overhead, and requires that all teachers are willing (or have the time) to do this, and also requires that all paper authors reciprocate: this is an unfair request, and is deemed inaccessible.

C++ (Second edition)', and the introduction to the C++ Core Guidelines, prior to planning any class structure or material. This will provide teachers with the necessary background to understand and critique the design and motivations behind C++ features.

Auditing preferred resources

Many resources used to teach C++ are either outdated or are of poor quality, and are – probably unintentionally -- harmful for those wanting to learn C++; these resources should all be actively avoided. There is a peer-reviewed guide on StackOverflow that essentially provides whitelisted educational C++ resources. The current survey found that '*The C++ Programming Language (Fourth edition)*' and '*A Tour of C++*' are the most popular formal resources for teaching C++.



Given that '*Accelerated C++*' and '*Modern C++ Design*' are books published in the early 2000s without revised updates, there is reason to believe that – at a superficial level – the courses using these textbooks are restricted to teaching C++98 idioms at best. This is not a mark against the authors' quality, but the style of programming that C++ programmers employ has changed over the past two decades since these texts were first published. Another teacher seems to use the C++ Standard as their teaching 'textbook', which is ill-advised on many levels, as the Standard acts as a contract between the compiler and the programmer: not as a resource for teaching C++. While consulting the Standard to produce one's materials is recommended, using the Standard as a teaching device is scarcely recommended, unless the course is directed at people needing to learn to read the Standard.

In the authors' experience, many people will be fond of the resources that they learnt from, unless they perform an objective audit of books to assess the quality of material. An audit is a lengthy process and requires cross-sectional analysis of many resources to determine if a book should be recommended or avoided. We would like to request that the study group charter time for analysing and determining which formal resources should be formally recommended by WG 21. One method previously used to help evaluate resources includes ACCU book reviews. It would be beneficial for the study group to consider the ACCU book review model for a broader range of resources, including videos, blogs, and so on.

Providing an ecosystem

Software development is never solely about writing code, and almost always includes reliance on tools such as package management, build systems, debuggers, sanitisers, profilers, version control, linters, code formatters, test frameworks, benchmarking, and third-party libraries.

Writing software is never as simple as getting it to compile. To quote Stepanov and Rose, “no one writes good code the first time” (Stepanov & Rose, 2015); we need to test our code to ensure that it is correct, and we need to benchmark it if we care about performance (Carruth, 2017).

Other tools, such as build systems, package management, debuggers, version control, and third-party are all imperative parts of software engineering, and we recommend that teachers look to passively or actively introduce students to these tools. A teacher may like to demonstrate their chosen toolset through an IDE or a Docker container to reduce the number of steps required to get set up.

Learning objectives, outcomes, and ‘ASSBATs’

A complete curriculum and its components should have clear learning objectives. For each part of the curriculum, the design should contain a handful of ‘ASSBAT’ items (A Student Should Be Able To[6]). These ASSBATs describe what the student must have learned in the terms of action statements. For example: after this module, a student should be able to "write a function that correctly uses value and reference parameters". An example of a detailed curriculum for senior high school students can be found here.

Appropriate order of teaching

About thirty percent of the surveyed answered that they might first teach the subset of C++ that intersects with C before teaching features unique to C++. Prior to the delivery of ‘*How to Teach C++ and Influence a Generation*’, an informal survey of books, tutorials, and videos was conducted, and revealed that a large population of the resources still teach C++ by first looking at low-level features that are found in C (for example, pointers, raw loops, and explicit memory management).

While the authors strongly commend the efforts made by the seventy percent of teachers teaching C++ from a high, top-down level, we would like to further expand this, so that as many educators as possible are teaching C++ as C++: not as C-plus-more.

Examples of books that do this include ‘*Programming – Principles and Practice Using C++*’ and ‘*A Tour of C++*’. An example of an online class achieving the same goal is ‘*C++ Fundamentals including C++17*’. We advise that guidelines produced by the group encourage curricula that empower students to write good C++ programs as soon as possible and not scare them away by promoting more complicated constructs early (some if which don't even have to be taught at all).

Conclusion

Despite being widely used, C++ is underrepresented in education, especially when contrasted with other programming languages such as Java and Python. We aim to start an effort to improve both the quantity and quality of education concerning programming using C++. We have given some recommendations and requests regarding the preparation of teachers before teaching, setting clear learning objectives, and stimulating the teaching of high-level C++, first by setting guidelines for teaching C++.

[6] See http://hosting.uaa.alaska.edu/afbeb/SymposiumVI/Materials/39_Carpenter_Donald_Materials.pdf.

We wish to spark a discussion in an evening session to discuss what a study group should do to improve the situation regarding teaching C++.

Works Cited

Carruth, C. (2017). Going Nowhere Faster. Bellevue: CppCon. Retrieved from https://youtu.be/2EWejmkKlxs

Gregory, K. (2015). Stop Teaching C. Bellevue: CppCon. Retrieved from https://youtu.be/YnWhqhNdYyk

Stepanov, A., & Rose, D. (2015). *From Mathematics to Generic Programming.* Crawfordsville: Addison-Wesley.