Teaching Concepts TS Online

P0821R0
By: Robert Douglas
Reply-To: rwdougla@gmail.com
Date: 10/16/2017

Background

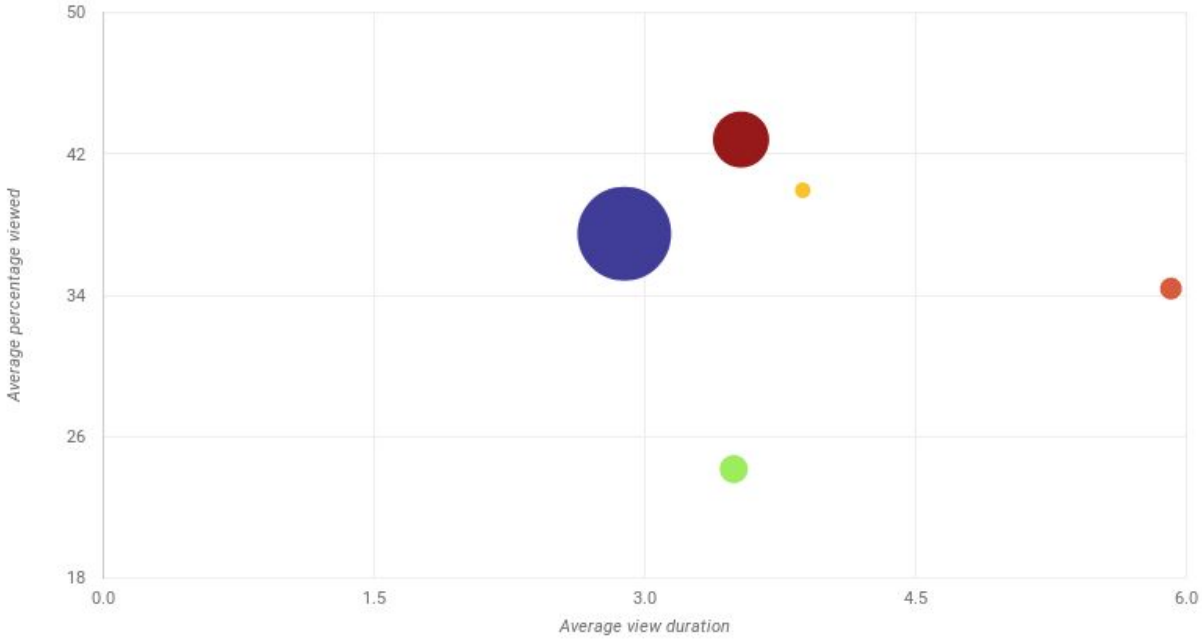
Motivation

With the inclusion of a portion of the ConceptsTS to the C++ Working Draft in Toronto, it came to my attention that there was still insufficient trusted feedback on concepts from the community. This appeared to be creating an abundance of FUD and this was impacting the ability of the committee to work together.

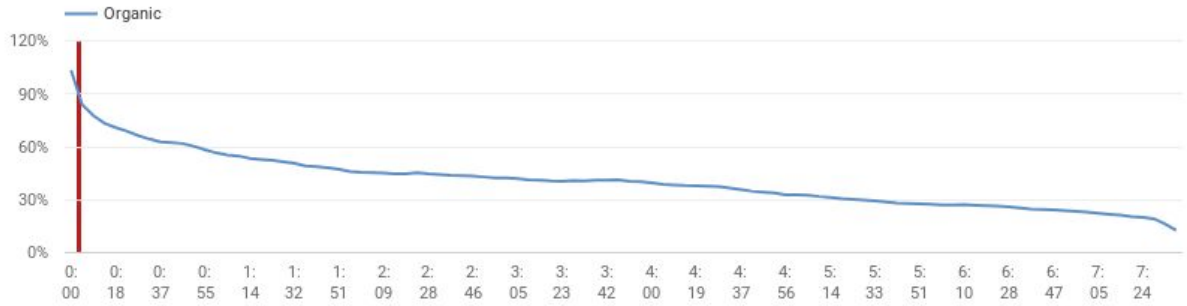
Prior to the Toronto meeting, I had begun a YouTube channel, creating videos teaching C++-related topics. I saw an opportunity to actually try teaching as much of the ConceptsTS as I could/should, and ask the viewers to submit some code samples, afterward, using concepts. The intent of this exercise, as it was made clear to the viewers, was to gather feedback for WG21 to consider about whether the feature set pulled in for Concepts was the correct set. This paper presents the results of this experiment.

About Teaching via Online Videos

To better understand the methodology, it's worthwhile talking about some of the issues with online videos.
Average viewer retention rate for any of my previous videos at the time was 3:16.
In the following graph, Blue is a video about typename, red is on std::optional, yellow on Type Erasure, green on CTAD, and orange on std::variant. Bubble size is total time spent watching them.



This is a breakdown of what percentage of the viewers are still watching the “Why Typename?” video by time into the video.



With this knowledge, keeping the video short and succinct is key, and so the amount of information that can be conveyed in that time is limited. Also, one cannot expect even half their viewers to watch the entire video.

Methodology

I created a single video at first, viewable here: https://youtu.be/xsSYPD0v5Mg
The point of this video was to teach the following:

1. Definition of *requires-expression.*
2. Definition of *requires-clause*.
3. Syntax for variable *concept*s
4. Usage of concept-id or *requires-expression* in *requires-clause* (“S1” in the results)
5. Usage of concept-id in template parameter list (*constrained-parameter*) (“S2” in the results)
6. Usage of concept-id as a placeholder in function signature (*constrained-type-specifier*) (“S3” in the results)

With the video I asked for submissions from viewers to:

1. Reimplement std::min using concepts
2. Reimplement std::for_each using concepts
3. Email their results to me along with answers to the following survey:
    a. Is C++ your preferred language?
    b. In years, what would you say is your experience in C++?
    c. Are you interested in using Concepts in your code in the future?

With these submissions, I encouraged viewers to also consult material elsewhere, or ask questions they had, within the comment section. 4 viewers submitted questions. It is not clear whether any of them also submitted responses, but I believe none did.

After several weeks of results trickling in, I released a second video:
https://youtu.be/yHhFC_zR5rY
This video intended to more thoroughly teach the grammar of the *requires-expression*. It also asked again for the same feedback. I kept track of whether each submission was before or after the second video was posted.

Recognized Limitations

1) The number of submissions, though well more than anything else I’ve seen to date, is still lower than desirable.
2) This experiment is not, on its own, a full representative sample of the C++ community, though the results show a wide variance in the demographics, at least in relation to experience levels.
3) Being an online resource, the viewer has few immediate resources to consult.
4) Though the viewer has access to a compiler through godbolt.org, with a link setup for the proper GCC version and compiler flags, most viewers were not going to develop this

alongside a proper testing environment, or really any environment that they could run the code.

5) The presented challenge takes place outside of any long-standing code base. It was intentional to use a well-understood toy example.
6) The videos did not teach 'best-practice', but rather, just the syntax and grammar. It was up to the viewer to infer what they should do.

Results

Organization

The following terms are used throughout the results
Syntax 1 (S1): Usage of concept-id or *requires-expression* in *requires-clause*
Syntax 2 (S2): Usage of concept-id as a *constrained-parameter* (template parameter list)
Syntax 3 (S3): Usage of concept-id as a *constrained-type-specifier* (placeholder in function signature)

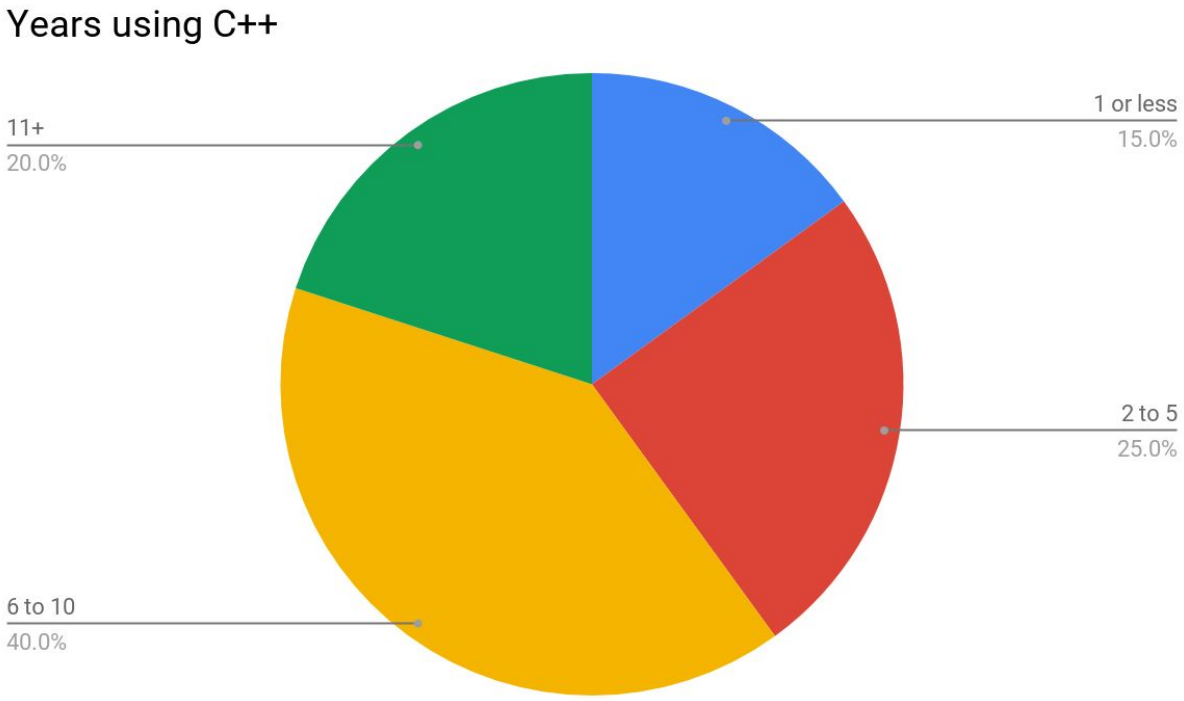
Respondents

Of the 1700 views for the first video, and 800 for the second, I received 20 submissions in total. According to data pulled from godbolt.org's servers, the associated link was hit about 100 times. It is unclear exactly how many of those were unique individuals, though.
Of these respondents, I have only ever met 3, personally, one of those I know simply from a local users-group I attend. To my knowledge, there are no regularly-attending committee members represented in this result set, but there is one paper-author I am aware of. I coached no one, individually, on this. 2 of the responses were submitted after the second video was published.
The data set is available at https://github.com/everythingcpp/ConceptsResponses. The summary tables are included in the Appendix. Respondents have been anonymized to a numeric identifier. This identifier is referenced in this paper, where needed.

Notable Stats

85% of respondents called C++ their preferred language

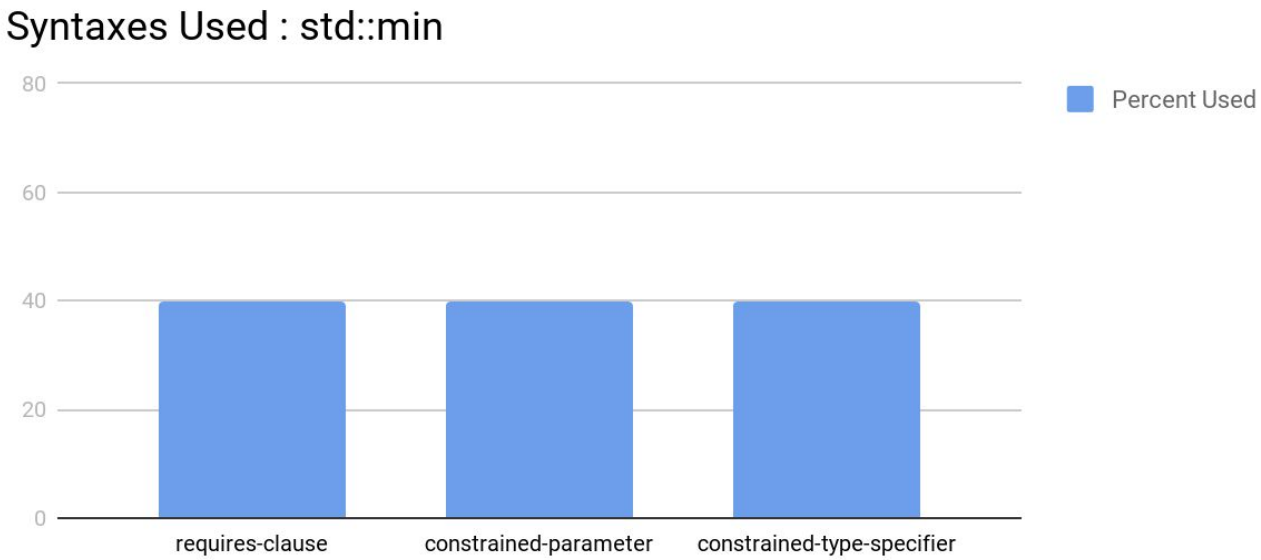
The following pie chart breaks down the experience levels of the respondents.



90% of respondents plan to use C++ concepts in the future

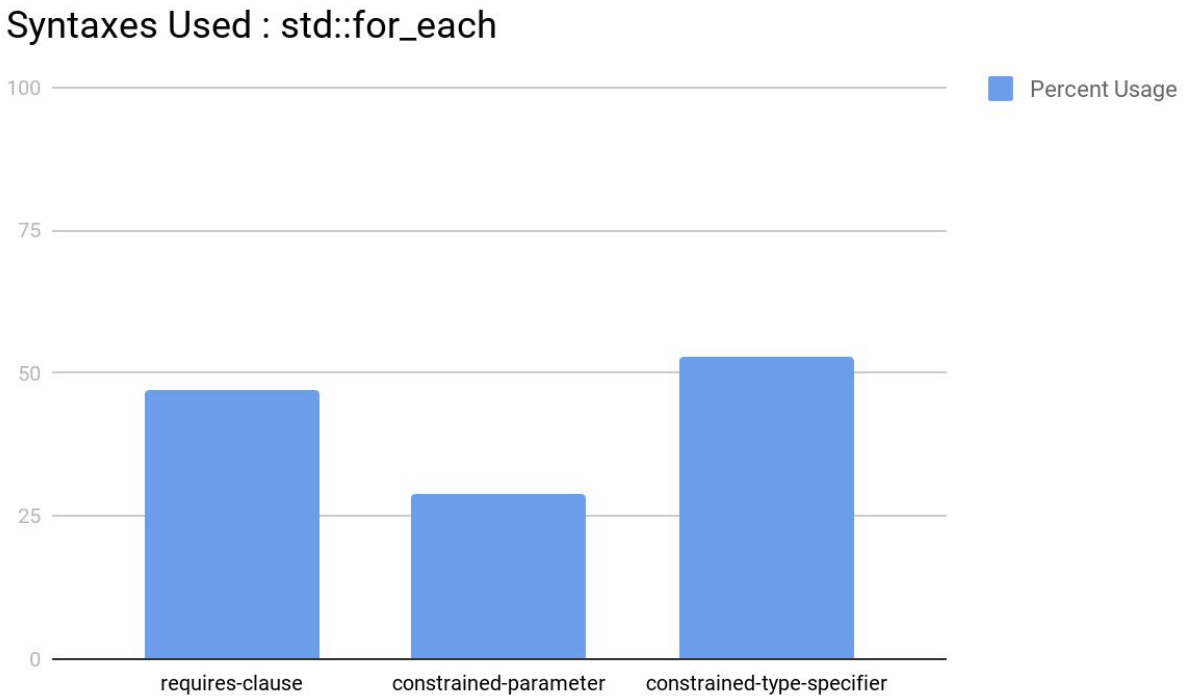
Minimum number of concepts created: 1
Maximum number of concepts created: **56**
Percentage of submissions with compiling solutions for both, whether correct or not: 16

Reimplementing std::min



4 Respondents used 2 syntaxes.

Reimplementing std::for_each



3 respondents used 2 syntaxes. There were 4 at one time, but one resubmitted and eliminated the combined use of the *requires-clause* and *constrained-parameter*, in favor of just using *constrained-type-specifier*.

Additional Notes

- Mixing syntaxes within the same function signature was not just done, it was surprisingly common. I did not teach that this was possible. People just did it.
- 2 submissions used *template-introduction* syntax, which I did not present.
- One respondent using *template-introduction* syntax read cppreference on concepts fully, prior to their submission.
- 2 submissions used function-style concepts, which I did not present.
- Crashes in the compiler were unfortunately common. It caused 00015 to not submit a solution for for_each.

Interesting Quotes from Respondents

- 00002: I don't like the idea of using a concept in-line like "f(Concept x)" Seems bad. A Concept feels like a restriction on a type (a type-class from functional programming), not a type itself (which is a restriction on values).

- 00010: I don’t think concepts will be as powerful as Haskell’s typeclasses in the current draft but I would love to be wrong.
- 00017: I am realy disapointed that we cannot use concepts as placeholder in C++20. This functionnality make template code readable.
- 00017: I tend not to consider concept as alias for types but as the abstract semantical part of types. When I discovered that "int foo(AConcept x,AConcept y)" implied that x and y had the same type, I was really surprised. It is like the detail (the actual type) was breaking the static abstraction layer offered by Concepts. It is as if in the language "int foo(abstract_base_class& a, abstract_base_class& b)" would implied that a and b to had the same dynamic type!!!

After Teaching Concepts and Reading All these Submissions

Counting the instances of *constrained-type-specifier*s (S3) took the most time and mental thought, for me, by far.
Jumping from the implementation to the Concepts it used seemed to negatively impact understanding the function signature.
*requires-clause*s and *constrained-parameter*s were far easier to visually scan for.
*constrained-type-specifier*s resulted in far less code to read.
Concepts make for far longer solutions when used at this scale.
Users will write a significant amount of additional code.

Conclusions

To Consider for Standardization

1) Users will readily mix the syntaxes within their code and even the same function signature.
2) Preference for the 3 syntaxes seem fairly evenly distributed.
3) Users are demonstrably asking for the ‘Natural/Terse Syntax’. (*constrained-type-specifier*s)
4) Users are all over the map on what they believe the constraints are or should be on their types. This should be the larger concern about how to teach concepts in practice. The syntaxes showed little distinction in ability for developers to use them.

Based solely on this experiment, my recommendation is that we additionally include the ‘Natural/Terse Syntax’ into the working draft, as defined in the Technical Specification. It is demonstrably learnable, and users appear to prefer it, albeit slightly.

For Teaching

1) Users should be told to write concepts with standard-provided type-traits, whenever possible. Leveraging the language to infer CopyConstructible or similar constructs tripped up a lot of users and led to errors.
2) Users should be directed to write concepts for likely breakages, only, to avoid risk of over-constraining. It turns out that it is rather easy to accidentally over constrain, if you are not careful.
3) Users are already discovering different syntaxes than presented, and choosing to use them, as well. Teaching just on syntax is unlikely to effectively eliminate the presence of other syntax choices in your code base. (ie, function vs variable concepts)
4) Novice developers were more willing to change or omit function signatures.
5) Veteran developers were more likely to over-constrain.
6) People are smart. If you question whether or not they can learn a thing, bet they can. But expect them to use it naively.

Future Efforts

I found this exercise encouraging at engaging the community at large, to get a sense of how people may actually use the facilities we standardize. I received a number of compliments and thanks for reaching out to them on the topic.

Though the result set for this trial was smaller than I hoped for, I believe that in time, we can solicit for larger and better data sets, but the process needs refinement. It would be worth organizing an email list or forum about how to engage the community and better teach the language.

WG21 should consider how we might leverage additional communities for similar feedback on other controversial decisions.

Acknowledgements

A major thank you to everyone who took the time to develop and submit code samples. This turned out to be a more arduous task for them, than I expected, and I am truly grateful for the time.

Thank you also to Alex Kondratskiy, Nate Wilson, and Nevin Liber for feedback on an early draft of these results.

Thanks to Matt Godbolt for letting me (a)buse compiler explorer for a month to get these results, and for his help on obtaining stats.

Appendix

The following tables were used for the statistics throughout this paper. These values were determined by hand.
For yes/no values, 1 -> yes, 0 -> no.
S1 is whether the code contained an instance of a *requires-clause*.
S2 is whether the code contained an instance of a *constrained-parameter*.
S3 is whether the code contained an instance of a *constrained-type-specifier* or a *template-introduction*.
“A solution” means that the respondent had some form of a solution that compiled.
For std::min, some respondents included extra overloads, specifically, taking a comparator. These were not explicitly asked for, but have been noted under “Included Callable” if they chose to do so.
“Syntactically correct” implies that it compiled with a simple set of test calls. These tests were simple calls that work with non-concept based std::min.

	std::min							
file	S1	S2	S3	A Solution	Syntaxes Used	Included Callable	Syntactically Correct	
00001.cpp			1	1	1	0	1	
00002.cpp	1			1	1	1	0	
00003.cpp	1			1	1	0	1	
00004.cpp			1	1	1	0	1	
00005.cpp		1		1	1	0	1	
00006.cpp		1		1	1	1	1	
00007.cpp			1	1	1	0	1	
00008.cpp		1		1	1	0	1	
00009.cpp			1	1	1	1	1	
00010.cpp	1		1	1	2	1	1	
00011.cpp	1	1		1	2	1	1	
00012.cpp	1	1		1	2	1	1	
00013.cpp			1	1	1	1	1	
00014.cpp	1	1		1	2	1	1	
00015.cpp	1			1	1	0	1	
00016.cpp	1			1	1	1	1	
00017.cpp			1	1	1	1	1	

00018.cpp			1	1	1	1	1	
00019.cpp		1		1	1	0	1	
00020.cpp		1		1	1	0	1	
Total	8	8	8	20		11	19	
	0.4	0.4	0.4					

	std::for_each					
file	S1	S2	S3	A Solution	Syntaxes Used	Syntactically Correct
00001.cpp			1	0	1	0
00002.cpp	1			1	1	0
00003.cpp	1			1	1	1
00004.cpp			1	1	1	1
00005.cpp		1		1	1	1
00006.cpp		1		1	1	1
00007.cpp			1	1	1	1
00008.cpp	1			1	1	1
00009.cpp				0	0	0
00010.cpp			1	1	1	1
00011.cpp		1	1	1	2	0
00012.cpp	1	1		1	2	1
00013.cpp			1	1	1	1
00014.cpp		1		1	1	1
00015.cpp	1			0	1	1
00016.cpp	1			1	1	1
00017.cpp			1	1	1	1
00018.cpp			1	1	1	1
00019.cpp	1		1	1	2	1
00020.cpp	1			1	1	1

Total	8	5	9	17		
	0.47058823 53	0.29411764 71	0.52941176 47			

For the following table,
Q1: Is C++ your preferred language?
Q2: In years, what would you say is your experience in C++?
Q3: Are you interested in using Concepts in your code in the future?

file	Concepts Written	Q1	Q2	Q3
00001.cpp	3	0	9	0
00002.cpp	3	1	15	1
00003.cpp	3	1	5	1
00004.cpp	4	1	5	1
00005.cpp	3	1	7	0
00006.cpp	12	1	9	1
00007.cpp	3	0	4	1
00008.cpp	2	1	9	1
00009.cpp	2	1	10	1
00010.cpp	5	0	7	1
00011.cpp	12	1	1	1
00012.cpp	6	1	11	1
00013.cpp	3	1	4	1
00014.cpp	56	1	13	1
00015.cpp	3	1	26	1
00016.cpp	5	1	0.5	1
00017.cpp	4	1	5	1
00018.cpp	13	1	10	1
00019.cpp	3	1	10	1
00020.cpp	1	1	1	0.5
Total		17		17.5