COMP6772 Advanced Software Engineering

Document #:	P6772R1
Date:	2021-06-04
Project:	Programming Language C++
Audience:	School of Computer Science and Engineering

Reply-to:	Christopher Di Bella <cjdb.ns@gmail.com>

Contents

§ 1 Abstract

P6772 seeks to remedy this by introducing a new C++ course that follows on from where COMP6771 stops.

§ 2 Revision history

§ 2.1 R1

- Moves from Google Docs to Markdown-generated HTML.
- Adds Motivation and Design sections to better communicate why content is necessary and how it will be delivered.
- Removes the open-source component to give students breathing room.

§ 2.2 R0

- Initial revision.

§ 3 Motivation

§ 3.1 C++

C++ is a language that rapidly evolves over time and is applicable to most domains, making it difficult for *COMP6771 Advanced C++ Programming* to cover all the relevant content. While the delta between C++11, C++14, and C++17 is quite small

between consecutive standards, the change between C++17 and C++20 is on a completely different scale. A lot of C++20's new features either introduce new programming paradigms (e.g. ranges, coroutines), change the way in which we write C++ at a language level (e.g. concepts, consistent comparison, modules), or change the way in which we build C++ (modules). All of these new features immensely simplify C++ programs, but are competing for time with existing content that is almost entirely still relevant, but in some cases no more important than the new potential material.

COMP6771 (and COMP3171/9171) has also always taken an interest in exposing tooling to make sure that students are building correct programs. Since the author's involvement with the course, it has always encouraged tools to detect memory leaks (Valgrind until 2014; AddressSanitizer since 2015). Since 2019, the course has also encouraged the use of IDEs (CLion in 2019; Visual Studio Code with clangd since 2020), debuggers, build systems (Bazel in 2019; CMake since 2020), a code formatter, a linter, a unit test framework, and package management. As C++ tooling has matured over the past decade, we have increasingly been exposing students to more and more tools to make their lives easier. It isn't a free lunch, however, and we require time to educate students on the purpose of these tools, and how to use them to get the best experience (often disguised as "how to maximise your mark"). In a course that's already filled to the brim, this is fairly difficult: we can either only superficially expose the tools, or hide them behind something else like an IDE, and while they reap the benefits, they're none the wiser about losing the tooling if they move to a different ecosystem.

§ 3.2 Software engineering practices

§ 3.2.1 Testing

> If you liked it, you should have put a CI test on it.
>
> – Google, with apologies to Beyoncé

The author's software engineering education, from *Software Design and Development* in the HSC, through their bachelor's degree in computer science at UNSW has always encouraged testing (and at times, test-driven development). Unfortunately, this never translated into *how* to appropriately write tests, or how much testing is enough. Additionally, the author didn't come across continuous integration until their first full-time job, and has learnt about various forms of testing outside of unit and integration testing over the years. This suggests that UNSW can expand on the tests it pushes for (e.g. CI, fuzzing), which can improve student development.

§ 3.2.1.1 Why this should be taught in COMP6772

Since having graduated, the author has seen a lot of code checked into projects that hasn't been adequately tested, and has marked COMP6771 assignments that would have benefited from better testing. This is evidence that universities in general have room to improve when it comes to ensuring that graduates meet learning outcomes 1, 3, and 4.

While COMP6772 will continue to beat the drum about unit testing and TDD, it will also bring up the importance of CI and fuzzing.

§ 3.2.2 Benchmarking

> You only care about performance that you have benchmarked. If you did not write a benchmark for your code, you do not care about its performance.
>
> – Chandler Carruth, 2017

The author never formally learnt about benchmarking while at university. COMP3171 (COMP6771) has emphasised the need for high performance code: its first (and possibly second) assignment is required to run in under a certain amount of time. This was facilitated by the POSIX tool `time`, which is a good start, but fails to tease out critical analysis related to practical performance. Worse still, computer science theory isn't always in sync with the real world. For example, the theoretical time complexity for traversing the contents of a contiguous array is $O(n)$, as it is for traversing the contents of a linked-list. However, due to cache locality, benchmarking reveals that contiguous arrays outperform linked lists in sequenced operations (and we haven't even talked about harnessing parallelism or heterogeneous computing). COMP6771 touches on briefly brings this up when contrasting `std::vector` and `std::list` (and anecdotally, there's always a non-significant number of surprised people), but has never had the time to prove, in large part due to the fact that educating students about how to write benchmarking code is hardly within course scope.

§ 3.2.2.1 Why this should be taught in COMP6772

C++ is a lightweight abstraction programming language, that gives you *control* over performance. As noted at the top, performance is only considered to be a genuine concern if a relevant benchmark has been built and analysed. While a good start (and good enough for COMP6771), writing a proper benchmark in C++ that meets Chandler's assertion about performance isn't as simple as the following.

```
auto const start = std::chrono::high_resolution_clock::now();
work_to_measure();
auto const stop = std::chrono::high_resolution_clock::now();
std::cout << (stop - start).count() << " ns\n";
```

The above lacks accuracy, and ~~docsn't~~ can't take into consideration the latency of operations such as allocation, which might irrationally impact a program's run-time.

§ 3.2.3 Ethics

The author is a part of various communities in the world of software engineering, and has worked as a software engineer in three different countries (Australia, Scotland, USA). In all cases, the author has enjoyed immense privilege, and has either witnessed or heard from trusted primary sources: workplace sexism, racism, homophobia, transphobia, ableism, intimidation, harassment, abuse, mental health belittlement, disregard for others' physical health and safety, and other antisocial behaviour. Due to this toxic behaviour, the author is aware of several people who have either left the community that they are a part of, or will leave very soon, if the behaviour continues. All three of the countries that the author has lived in have laws preventing this sort of behaviour, and yet it persists. This is unacceptable, and needs to be addressed as early

on in a person's career, to ensure that the listed behaviour does not become normalised for them.

Another perspective to consider on the importance of ethics in software engineering is that software is built for people, by people, but it doesn't necessarily treat everyone equally. Even large corporations such as Google and Twitter with enormous resources at their disposal have produced software that has engaged in discrimination. As with workplace behaviour, software that produces these results is completely unacceptable, and it is again within our ability to educate on the need for diverse and inclusive workplaces so that software solutions do not enage in antisocial behaviour.

§ 3.2.3.1 Why this should be taught in COMP6772

The author strongly believes that a single ethics course in the final year is unable to teach values that promote a safe workplace, where software solutions cater to all. It's far too late in a student's development, and it's far too limited a time for teaching staff to cover all the relevant material. Worse, the author understands that many students (from universities in multiple countries) hold their compulsory ethics classes in high contempt, because they don't understand why education on ethics is necessary. The author does see the value in a consolidated ethics class at the end of a computer science or engineering program, but only if ethics subtopics are explored in every course that CSE has access to. Then, the final ethics course can do two things: draw links between the ethics components across courses, and explore topics that genuinely require a certain amount of maturity to discuss.

The author is aware that this is a system-wide change to multiple programs, and while they would like to see this happen, is also aware that this kind of change is well beyond the scope of the proposal and would likely take years to complete. In the meantime, the author is of the opinion that a course could pilot having an ethics week to gauge its success, and relay this to the program committees.

§ 3.3 There is a *lot* going on here. Are you sure this you're going to fit this all into ten weeks?

In writing P6772R1, the author has internalised that there are at least two courses competing for attention within this proposal: a course that extends C++ education further than COMP6771 has the bandwidth to offer, and a course that teaches contemporary software engineering practices with a real-world project. It might be worth splitting the course in two: COMP6772 *Advanced C++ Programming*, and COMPXXXX *Software Engineering for Computer Scientists*.

Note that in the event of a split, an ethics module would be present in both courses.

§ 3.4 Overlap with other courses

The author is aware that some of the software engineering content in this section may be taught in other courses. This shouldn't be a discouraging factor from COMP6772 incorporating the material. Different lecturers will provide different perspectives on matters, and explain material in ways that may resonate with different people.

Some topics—like ethics and testing—not only benefit from varied input, but also from repetition. We stand a better chance of getting students to care about critically important topics if they're brought up at multiple junctions in their career.

Finally, some overlap will be necessary because this course would otherwise have too many prerequisites. If we take a look at just the C++ content that COMP6772 proposes, without overlap, it would require students to take COMP6771 (C++ 101), COMP3141 (composition), COMP3121 (algorithms), COMP3231 (allocators), COMP3131 (lexers and parsers), and both MATH1081 and MATH3711 (group and ring theories). That's an awful lot of prerequisites, and would probably result in only a handful of students ever even meeting the bar for entry. By cherry-picking relevant content and giving a light introduction/refresher, COMP6772 can drop almost all of these prerequisites.

§ 4 Design

The motivation section identifies many problems, and attempts to address them. The proposed course does not attempt to tackle them single-handedly, but rather aims to focus on exposing students to areas that the author is either adept or an expert in. Material that is repeated in previous courses will likely be presented with a different perspective, or serve as repetition to emphasise its importance.

§ 4.1 C++

The course is a sequel course to *COMP6771 Advanced C++ Programming*, so it will expand students' knowledge on current C++ programming techniuqes. More specifically, the course will explore program composition in C++, container design, and generic programming.

§ 4.1.1 Tooling

- Duration: two weeks.
- Introduces students to C++ tools beyond a compiler and debugger.
  - Advanced compiler and debugger techniques, build systems, compile-time analysis, run-time analysis, benchmarking, fuzz testing, continuous integration
- Assessment: passive, through other assessments.

§ 4.1.2 Program composition

- Duration: two weeks.
- Gently introduces students to category theory and group theory.
- Details ranges, monads, coroutines, and how to effectively debug composed programs, as opposed to imperative programs.
- Assessment: a lexer and parser that must be written using the above.

§ 4.1.3 Containers design

— Duration: two weeks.

— Details constant expressions, conditional noexcept, using allocators, placement new.

— Lectures will implement `std::vector` (COMP6771 approximates a `vector` implementation to teach RAII, whereas COMP6772 will implement a genuine `std::vector` to show how it works under the hood).

— Assessment: Implement a standard container (specific container TBD).

§ 4.1.4 Generic programming

— Duration: two weeks.

— Gently introduces abstract algebra.

— Details algorithm design, iterators, implementing an iterator, designing a range adaptor, and designing a concept.

Readers should note that the "algorithm design" does not overlap with COMP3121/3821. We instead look at generalising algorithms, analysing their requirements, building an interface that broadly caters to as many users as possible (while meeting certain requirements), and then looking for optimisation opportunities.

§ 4.1.5 Relationship with COMP6771

The author expects, that over time, the two courses will trade material. As features in more recent C++ standards become more mainstream, they may bubble their way up to COMP6771, and the older ways would end up in COMP6772. For example, C++20 introduced a feature known as consistent comparison, which reduces the number of overloaded comparison operators one needs to write from two-to-six to one or two (the compiler will generate the remainder). This is fairly new technology, and since it's not widely deployed, it makes sense for COMP6771 to continue teaching the way that's been available since the 80s. When consistent comparison is used by more of the community, then the respective LiCs may discuss whether or not it's appropriate to trade this material, thereby giving casual students the easier to understand and more-likely-to-be-correct option, and leave the manual one to folks who are hungry for more C++.

§ 4.2 Software engineering tools and practices

§ 4.2.1 Community

Popular programming languages, while tools, have thriving communities made up of people. That's how tools come into being, and how the language evolves.

§ 4.2.2 Ethics

Until such a time that a first year COMP course teaches about diversity, equity, and inclusion, COMP6772 will dedicate one week to this topic. Once this criterion is met, COMP6772 will explore the ramifications of software that permits the proliferation of fake news and misinformation.

§ 4.3 Should this be a COMP or SENG course?

The author firmly believes that this should be a course available to students studying computer science. Computer science may not be an official engineering degree, but many computer scientists end up graduating and finding themselves employed as software engineers. Therefore, the author is of the opinion that non-core courses that expose software engineering practices should be available for anyone who may find themselves employed as a software engineer, provided they meet the minimum maturity requirements.

§ 5 Proposed course

- **Units of credit:** 6
- **Contact hours:**
  - **Lectures:** 4
  - **Tute/Lab:** 2
- **Prerequisites:** COMP6771, and at least one of COMP3121/3821, COMP3131, or COMP3231/9201/3821/9283.

§ 5.1 Course abstract

This course continues teaching C++ where COMP6771 leaves it. Deepens knowledge learnt in COMP6771; introduces advanced topics that will reshape the way in which one writes C++ programs.

Detailed study of complex C++ tools for better, faster, and correct programs. Program composition in C++. Building containers that isare performance-sensitive. Generic programming.

§ 5.2 Objectives

§ 5.2.1 Knowledge and understanding

Students will develop knowledge and understanding of:

- contemporary C++ software engineering practices
- the philosophy of C++
- social and ethical issues, and their effect on software engineering

§ 5.2.2 Skills

Students will develop skills in:

- using C++ tools
- more in-depth C++ programming techniques

§ 5.2.3 Values and attitudes

Students:

— appreciate the complexity of software engineering beyond the scope of university
— employ software engineering best practices in their projects
— value the need for considering social and ethical issues as they develop software

§ 5.3 Outcomes

§ 5.3.1 Knowledge and understanding

Objectives	Outcomes
Students develop knoweldge and understanding of:	A student:
1. contemporary C++ software engineering practices	— describes incidents in a detailed manner when asking for help — discusses the benefits and drawbacks of using libraries — explains the benefits of using tooling — describes the trade-offs of link-time and profile-guided optimisation — distinguishes between standard library implementations
2. the philosophy of C++	— describes the concepts underpinning generic programming — explains the purpose of allocators — discusses the benefits and drawbacks of argument-dependent lookup
3. social and ethical issues, and their effect on software engineering	— discusses and evaluates social and ethical issues in several contexts — constructs software solutions that address social and ethical issues

§ 5.3.2 Skills

Objectives	Outcomes
Students develop skills in:	A student:

Objectives	Outcomes
4. using C++ tools	— synthesises a build environment — employs compile-time analysis to ensure best practices are followed — employs run-time analysis to narrow down the existence of bugs — constructs unit tests to ensure correctness — constructs fuzz tests to ensure correctness — constructs benchmarks to measure performance — constructs documentation to detail how and why interfaces should be used — applies tools to automate development processes — synthesises documentation for their designs and interfaces
5. more in-depth C++ programming techniques	— composes solutions using range adaptors, monads, and coroutines — synthesises generic algorithms using concepts — constructs an allocator-aware container — constructs a range adaptor — evaluates the need for esoteric C++ features

§ 5.4 Assessment structure

— *Technical assignments:* 40%

— *Essay:* 30%

— *Lab work:* 15%

— *Final exam:* 15%

— To ensure student participation in all assessments, students need to pass all four components.

— Students all have one “bad week” card, where they can request a two-day extension for *an* assignment, no questions asked, and no evidence required.

§ 5.4.1 Assignment 1

— *Area of focus:* Composition in C++.

— *Brief description:* Students use range adaptors, monads, and coroutines to build a lexer and parser.

— *Weighting:* 20% of final mark.

§ 5.4.2 Assignment 2

— *Area of focus:* Ethics.

— *Brief description:* Students write an essay on a contemporary issue concerning diversity, equity, and inclusivity.

— *Weighting:* 25% of final mark. The weighting reflects the importance of the topic rather than its difficulty.

§ 5.4.3 Assignment 3

— *Area of focus:* Containers and allocators

— *Brief description:* Students implement an allocator-aware container.

— *Weighting:* 20% of final mark.

§ 5.4.4 Lab work

Specifics to be fleshed out, but will cover outcomes that assignments do not.

§ 5.4.5 Exam

Specifics to be fleshed out, but will cover most technical outcomes.

§ 5.5 Class structure

Week	Lectures	Tute/Lab	Assessment
1	— Course outline — What is software engineering? — The C++ community — Tools beyond the compiler — Libraries	TBD	Assignment 1 released

2-3	— Category theory basics
	— Composition using
	  — Range adaptors
	  — Monads
	  — Coroutines
	— Debugging techniques
		TBD	

4	— Unit testing (revised) — Fuzz testing — Documentation	TBD	Assignment 1 due EOW

Week	Lectures	Tute/Lab	Assessment
5	— Equity — Diversity and inclusivity — Ramifications of poor/no ethics via case study	TBD	Assignment 2 released
6	No classes		
7-8	— ‘Advanced’ templates — Constant expressions — `noexcept` specifiers — Allocators — Uninitialised memory algos	TBD	Assignment 2 due EOW7 Assignment 3 released
9-10	— Abstract algebra basics — What is generic programming? — Producing a generic algorithm — Iterators and coordinates — Synthesising an iterator — Designing a range adaptor — Designing a concept	TBD	Assignment 3 due EOW9