How to Teach "Modern C++" to Someone who Already Knows Programming?

Adalbert Gerald Soosai Raj
Department of Computer Sciences and Education
University of Wisconsin-Madison
gerald@cs.wisc.edu

Varun Naik
Department of Computer Sciences
University of Wisconsin-Madison
vnaik@wisc.edu

Jignesh M. Patel
Department of Computer Sciences
University of Wisconsin-Madison
jignesh@cs.wisc.edu

Richard Halverson
Department of Educational Leadership and Policy Analysis
University of Wisconsin-Madison
rich.halverson@wisc.edu

ABSTRACT

The C++ programming language has undergone major changes since the introduction of C++11. 'Modern C++,' defined here as C++11 and beyond, can be viewed as a new language compared to C++98 (the version of C++ introduced in 1998). Many new features have been added to modern C++, including lambda expressions and automatic type deduction. The standard library has also been dramatically updated with constructs such as `std::unordered_set` and smart pointers. The traditional way of teaching C++ by first teaching C's low-level features, such as raw pointers and `char *` strings, is potentially ineffective when teaching modern C++. Based on this hypothesis, we updated the way in which we teach C++ at UW-Madison by teaching the most important high-level features (containers, iterators, and algorithms) first, and introducing the low-level features (raw pointers, dynamic memory management, etc.) only when they are necessary. In this paper, we present our experiences teaching modern C++ with this top-down approach. We find that with our new approach, students' perceptions about learning C++ are largely positive.

CCS CONCEPTS

•**Social and professional topics** → *Computer science education;*

KEYWORDS

Modern C++ pedagogy, Top-down approach, Problem-oriented approach, Syllabus design, Ordering of topics, Course experiences

ACM Reference format:
Adalbert Gerald Soosai Raj, Varun Naik, Jignesh M. Patel, and Richard Halverson. 2018. How to Teach "Modern C++" to Someone who Already Knows Programming?. In *Proceedings of 20th Australasian Computing Education Conference, Brisbane, QLD, Australia, Jan 30-Feb 2, 2018 (ACE 2018),* 8 pages.
DOI: 10.1145/3160489.3160503

DOI: 10.1145/3160489.3160503

1 INTRODUCTION

C++ is one of the most popular programming languages today. It is currently rated as the third most popular language next to Java and C [26]. The first standardization of C++ happened in 1998 (namely C++98). Some *major* changes to the core language and the C++ Standard Library happened in 2011 with the introduction of C++11. Some minor extensions over C++11 happened in 2014 as a part of C++14. The latest revision of the language, namely C++17, was released in 2017.

The new features that were introduced in C++11 change the language in significant ways, arguably making it very different from C++98. The creator of C++, Bjarne Stroustrup, notes, "*Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before*" [25]. The standard library was also improved immensely with new algorithms, container classes, and smart pointers.

The traditional approach of teaching C first before teaching C++ is considered ineffective for learning modern C++ [8, 22]. Since C++11 is considered a new language, the way we teach C++ needs to change so that we can help students become better *modern C++ programmers.* With this intent, we updated our C++ curriculum so that it better reflects the philosophy behind modern C++.

In this paper, we report the changes we made to our C++ course and share our pedagogical experiences. We also present students' perceptions on learning modern C++ using a top-down approach. We believe that our work has the potential to foster interest among computing teachers/researchers to answer the following question: *How do we teach a new programming language to someone who already knows some other programming language(s)?*

2 RELATED WORK

Bjarne Stroustrup [22] has shown an approach for learning and teaching standard C++ (C++98). This paper focuses on the design and programming techniques to be emphasized, subsets of the language to be learnt first, and the subsets of the language to be emphasized in real code while learning standard C++. Stroustrup suggests using C++ as a higher-level language without loss of efficiency when compared to lower-level style (C style of C++). We consider our work to be an extension of this work specifically focused on learning modern C++ (C++11 and beyond) using a top-down approach.

Accelerated C++ [10], a textbook written by Andrew Koenig and Barbara E. Moo, takes a practical approach to solving problems using C++. This book helps students write non-trivial C++ programs right from day one. It starts with the most useful concepts (e.g., string and vector) and postpones the most primitive ones (e.g., pointers and dynamic memory allocation) to the end until they are actually needed (e.g., for writing a customized vector class). The book mainly focuses on real problems and solutions instead of just teaching the features of C++. The features are introduced as and when they are needed for solving problems. It also covers the language and the standard library together, and the students are required to use the standard library right from the beginning. This book was published in 2000 and was written for standard C++ (C++98). The ordering of topics in our course was inspired by this book, although we updated our syllabus to be relevant for modern C++.

In a series of articles [11–19], Koenig and Moo explain their rationale behind the organization of topics in their C++ course at Stanford University. These articles show that the order in which topics are taught is as important as the topics themselves. We have adopted these principles to organize the content for teaching modern C++.

Howe et. al. propose a components-first approach for teaching introductory CS courses [9]. Components-first approach is the process of developing software by reusing a library of existing software components (e.g., C++ Standard Library). They compare and contrast two such approaches, namely Koenig-Moo (KM) version [10] and Reuseable Software Research Group (RSRG) version [20, 21]. Our approach is similar to their approach, but our goal is not to teach component-based software engineering but instead to teach modern C++ using a top-down approach.

Ivaylo Donchev has shared his experiences of teaching C++11 to undergraduates [7]. The report focuses mainly on the new C++11 features (e.g., automatic type deduction, uniform initialization, range-based for loops, lambda expressions, etc.) that were added to an existing C++ course. This report highlights the problems that the instructor faced when teaching C++11 features to students (e.g., the difficulty of teaching lambda expressions because of their non-intuitive syntax). Our work differs significantly from this work since we have modified the course completely by focusing on the higher-level abstractions provided by modern C++, and then going to the lower-level details as and when needed.

Stroustrup has shared his experiences about teaching C++ in an undergraduate freshman course [23]. In this report, he has described in detail how he brought about a curriculum change for the introductory programming course at Texas A & M University (TAMU). He mainly argues that the primary goal of software education is to be the foundation for professional work. He wrote a book on programming using C++ [24] based on his experiences teaching this course. He also highlights the importance of teaching C++ as a high-level language with cleaner abstractions instead of following a C-first approach. Our curriculum change also focused on a similar philosophy with a larger focus on the C++ Standard Library.

3 COURSE DETAILS

The C++ course in this study was taught at the University of Wisconsin-Madison during the Fall 2016 semester (the first semester of the 2016-2017 academic year). It was a one credit course with one 50-minute lecture every week. There were 102 students enrolled in the course. The course was intended to teach C++ to students who had already learnt Java in their introductory CS course. There were seven programming assignments, one final project, and no exams. The duration of the course was 15 weeks. There was one instructor and one Teaching Assistant (T.A.).

3.1 Java Background of Students

The students in our course had previously taken an introductory programming course in Java. This Java programming course was a prerequisite for taking our C++ course. The following topics were covered in the Java course:

(1) **Programming basics:** variables, conditionals, loops, methods, arrays, and ArrayLists.
(2) **Object-oriented programming:** objects, classes, composition, inheritance, polymorphism, exceptions, file I/O, abstract classes, and interfaces.

We note that our introductory Java programming course did not focus on using Java's in-built packages except ArrayList. More details on the programming assignments and the projects that students did as a part of our Java course can be found in this url: http://pages.cs.wisc.edu/~gerald/cs302/

3.2 Issues with the Previous Course Content

The contents of our C++ course before Fall 2016 are shown in Table 1. As can be seen, the previous course content was targeted primarily towards teaching low-level topics in C first before introducing additional, more complex concepts in C++. The C++ Standard Library, which is useful for writing real-world programs, was introduced only in the last two classes.

Our previous course content was suitable for teaching C++98. We identified the following issues with our previous syllabus and ordering of topics for teaching modern C++:

(1) There is too much emphasis on C in a C++ course. The first half of the course (weeks 1 - 7) focuses mainly on the low-level features of C rather than the high-level abstractions of C++.
(2) Most useful ideas are taught at the very end. For example, the topics in the C++ Standard Library (i.e., containers, iterators, and algorithms) that are very useful to write modern C++ code are taught at the very end of the course.
(3) Language features are introduced without proper motivation. For example, topics such as the big three (copy constructor, copy assignment operator, and destructor) are introduced as C++ language features at a time when students may not appreciate the need for these features.
(4) Students' prior knowledge in Java is not utilized properly. For example, file I/O is taught very late in the course (during week 12). If we can teach file handling at the beginning

of the course, then this would enable us to develop non-trivial programming assignments (using file I/O) right from the beginning of the course. Usually, when we teach a programming language to beginners, we save file I/O till the end because it is considered to be a complex topic for beginners. But since our students already know file handling in Java, it is potentially ineffective to postpone file handling in C++ till the end.

After identifying the issues with our previous course content, we decided to update our syllabus and ordering of topics for teaching modern C++. As a first step in this process, we were interested in finding out the parts and features of the language that were most frequently used in modern C++ code bases. We contacted several graduate students and software engineers using modern C++ in their everyday work. As a result of the conversations we had with these people, we found that the C++ Standard Library plays an important role in writing modern C++ programs since people generally write code by reusing the features provided by the standard library.

Table 1: Previous syllabus for our C++ course

Week #	Topic
1	Course information, history, high-level differences, process of writing C++ programs using Linux tools
2	Constants, enumerations, structures, arrays
3	Enum, struct, arrays, vectors, parameter passing modes
4	Pointers to structs/classes, arrays, dynamic allocation
5	Abstract memory model, reference variables, passing params to and return values from functions
6	.h and .cpp files, defining classes, multi-file compilation
7	Makefiles, constructor, member initialization
8	Copy constructor, copy assignment, destructor
9	OperatorX syntax and use, member vs. non-member options (assignment and arithmetic operators)
10	Explicit, member / non-member function pairs
11	Overloading, condition states, string class, C strings
12	File I/O, manipulators, C I/O
13	Templated functions and classes, containers
14	Iterators, generic algorithms, function objects

3.3 Principles for Course Content Organization

The course content was organized based on the following principles from Koenig and Moo [10]:

(1) Explain how to use language and library facilities before explaining how they work.
(2) Motivate each facility with a problem that uses that facility as an integral part of its solution.
(3) Present the most useful ideas first.

We added one principle, namely:

(4) Teach an idea/feature only when it is necessary to achieve a specific goal.

We added the fourth principle because we wanted to teach the features of C++ with proper motivation. We believe that if C++ features are taught at the right time when they are really necessary, then the students will be better able to appreciate the value of these features, when compared to introducing these C++ features as language technical details/constructs without proper motivation.

3.4 Modified Course Content

The first major change that we made was to update the course content. We also ordered the topics so that the most useful ideas were taught first. We define most useful ideas as the ones that would enable a new C++ programmer to write a non-trivial application program as quickly as possible. We define a non-trivial program to be something that may be useful in the real world, such as finding the set of students who are enrolled in two different courses using set intersection.

The modified course content that was used in Fall 2016 is shown in Table 2. As we can see in this table, the topics are divided into the following three components:

(1) Using the C++ Standard Library: weeks 1 - 6
(2) C++ language essentials: weeks 7 - 11
(3) Low-level programming: weeks 12 - 15

Table 2: New syllabus for our C++ course

Week #	Topic
1	Introduction to C++
2	I/O streams, file I/O, and strings
3	Sequence containers
4	Associative containers
5	Iterators and algorithms
6	Lambdas
7	References and classes
8	Inheritance and polymorphism
9	Operator overloading
10	Generic programming
11	Error handling
12	Smart and raw pointers
13	Resource management
14	Rule of zero and rule of three
15	C vs C++

The topics we taught in the three components of the course are described in more detail in the following sections.

3.5 Using the C++ Standard Library

The first component of the course focused on how to use the C++ Standard Library. The focus here was to help students understand the importance of using the C++ Standard Library. We wanted them to understand how the standard library is organized so that they could make use of it effectively.

The topics that were covered in this first component of the course are as follows: file I/O and strings, sequence containers (vectors), associative containers (sets and maps), algorithms (`std::find`, `std::sort`, `std::reverse`, `std::transform`), and lambdas (with `std::remove_if` and `std::transform`).

Next, we explain our rationale behind emphasizing the importance of understanding the organization of the C++ Standard Library.

3.5.1 Organization of the C++ Standard Library. The C++ Standard Library consists of three major elements, namely *containers, iterators, and algorithms.* The organization of these three elements is shown in Figure 1.

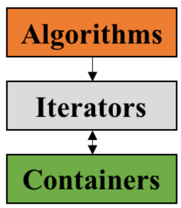

Figure 1: The organization of containers, iterators, and algorithms in the C++ Standard Library.

Containers (e.g., vector, set, map) store data. Iterators are used to access the elements in containers. Algorithms may perform some operations on containers with the help of iterators. *Algorithms cannot directly act upon the containers, but instead they can only act upon the iterators over the containers.* This fact was important for students to reason about code snippets like the one shown below.

```
std::vector<int> v = {1, 3, 2, 3, 4};
v.erase(std::remove(v.begin(), v.end(), 3), v.end());
```

This code snippet eliminates all the elements in the vector v that have a value of 3. The *remove*[1] algorithm moves all the elements that are not equal to 3 to the beginning of the vector and returns an iterator to one element past the last unremoved element. The vector after applying `std::remove` is: $\{1, 2, 4, ?, ?\}$. In this example, `std::remove` returns an iterator to the first element following the number 4 (i.e., the first unwanted element). The unwanted elements are shown as ? since the C++ standard does not specify what those elements should be. `std::remove` does not have the functionality to change the size of the vector. Instead, a subsequent operation is needed to eliminate the unwanted elements using the *erase*[2] member function of the vector class, which has the power to directly modify the structure of the container.

The *erase* member function takes two arguments, namely the first and the last iterators, and eliminates elements in the range `[first, last)`. The first iterator in our example is the one pointing to the first unwanted element (iterator returned by `std::remove`) and the last iterator is the `end()` iterator of the vector. Now, when we apply *erase* on the vector from the first to the last iterator, the vector size changes from 5 to 3 elements, leaving the two elements shown as ? out of bounds. This common C++ technique to eliminate elements that satisfy a certain condition is popularly known as the *erase-remove idiom* [4].

If we did not teach the organization of the C++ Standard Library at the beginning of the course, then usage patterns like these could confuse the students. They might not appreciate the subtle differences between the *erase* and the *remove* functions when removing some elements from a container. But with our approach, they can understand why both these functions are necessary to eliminate elements from a container and also gain a better understanding of how the C++ Standard Library is organized.

3.5.2 Rationale behind starting with the C++ Standard Library. We wanted our students to start writing *non-trivial C++ programs* right from the beginning. A natural way to do so is for students to start using the features offered by the C++ Standard Library. For example, if one wants to sort a set of objects, it is better to use the `std::sort` provided by the C++ Standard Library instead of writing a sort function from scratch. If the implementation of the sort became a performance issue, one could choose to write a custom sort function later. However, a good starting point is to use the C++ Standard Library's implementation, which is developed and maintained by experts in the field.

3.5.3 Why teach lambdas? We taught lambdas[3] early in the course since they enable more effective use of the C++ Standard Library. Introducing lambdas earlier in the course helped us when we taught how to sort user-defined objects using `std::sort`. As shown below, `std::sort` takes in a function object that defines the comparison function to sort a set of custom defined objects. In this example, a lambda expression is used as a function object to define the comparison function for `std::sort`.

```
// sort using a lambda expression
std::sort(s.begin(), s.end(), [](int a, int b) {
  return b < a;
});
```

Lambdas are heavily used in code examples of the Algorithms Library (e.g. `std::remove_if`[4], `std::sort`[5]). When students refer to the algorithms in the C++ Standard Library, they encounter lambda expressions in multiple places since function objects are written using lambdas instead of their predecessor, functors. Therefore, it is very important for students to understand lambdas so that they are able to read and make sense of the (many) algorithms in the C++ Standard Library.

We gave an introduction to lambda calculus in mathematics before introducing lambda expressions in C++. We introduced lambda calculus as follows:

A function to square a number:

`square(x) = x * x`

In lambda calculus:

`x -> x * x`

Using this example for a square function in lambda calculus, we helped students to understand that functions can be expressed without a name in lambda calculus. Similarly, in C++, we showed them that lambda expressions are functions without names (i.e.,

[1] http://en.cppreference.com/w/cpp/algorithm/remove
[2] http://en.cppreference.com/w/cpp/container/vector/erase
[3] http://en.cppreference.com/w/cpp/language/lambda
[4] http://en.cppreference.com/w/cpp/algorithm/remove
[5] http://en.cppreference.com/w/cpp/algorithm/sort

anonymous functions). We showed them how a normal function in C++ can be converted to a lambda expression as shown below:

```
// A square function in C++:
int square (int x) {
  return x * x;
}

// A square function using lambdas:
[] (int x) {
  return x * x;
}
```

Once we finished teaching the first component of our course (i.e., C++ Standard Library), we had given the students the most important tools to write non-trivial C++ programs. By doing so, we were able to align our course to principles (1) and (3) outlined in Section 3.3.

3.6 C++ Language Essentials

The second component focused on multiple essential topics in the C++ core language. The topics that were covered in this component of the course were: object-oriented programming (OOP), operator overloading, generic programming, and error handling.

Since the students in the course had already learnt OOP in Java, we did not spend time teaching the concepts of OOP, but instead spent more time teaching C++ specific details including syntax. The topics we covered in OOP were: member initializer lists, splitting the interface and implementation of a class between the .hpp and .cpp files respectively, virtual functions, and pure virtual functions.

At this time, we introduced the C++ reference types. Since the students came from a Java background, they were already aware of the primitive data types like `int`, `double`, `char`, etc., and they had a good understanding of references in Java. In C++, to avoid inefficient copying of data when passing a container of objects as an argument, students needed to pass the container by reference explicitly.

We were mindful of the need to introduce references when teaching OOP in C++. For example, when teaching polymorphic functions in OOP, we used reference variables as shown in the sample code snippet below.

```
// A polymorphic function to print student details
// based on the type of the student.
void printDetails(Student &sRef) {
   sRef.printStudent();
   std::cout << std::endl;
}
```

Based on our principle (4) as outlined in Section 3.3, we took a detour introducing references in C++, before delving into OOP (for C++).

3.7 Low-Level Programming

The third and the final component of our C++ course focused on low-level programming. We taught the following topics in this component: raw pointers and smart pointers, dynamic memory allocation, rule of three and rule of zero, and resource acquisition Is initialization (RAII).

Up until this point in our course, we had not discussed pointers, since they were not necessary. The rationale behind waiting until the end of the course to introduce pointers was that we wanted to focus on the higher-level abstractions in C++ first before we delve into the lower-level details of how they work. For example, we wanted students to understand how to use `std::vector` before learning how they could write their own vector class.

We motivated the need to learn *pointers* by explaining how the `std::vector` is implemented. We discussed how the vector may resize itself when new elements are added. This helped the students understand that their vector class should manage its own memory. This was an ideal place to introduce pointers to manage resources like memory using dynamic memory allocation. Also, the stage was set for introducing *destructors* to free resources that were allocated in their vector class.

Next, we discussed how copying works in `std::vector`. We also showed that in the vector class we had created, copying does not work as expected since only a shallow copy is made by the default copy constructor provided by C++. This was a good time to introduce *copy constructors* and *assignment operators*, and students appreciated the need for making a deep copy of their vector.

We also taught *smart pointers* and showed how freeing of memory works automatically when we use them. Specifically, we discussed the `std::unique_ptr` and `std::shared_ptr` constructs. We taught smart pointers mainly because most modern C++ code bases use them, and the students should be prepared to understand and work with them.

We used the Resource Acquisition Is Initialization (RAII) programming idiom right from the beginning of the course for concepts such as reading and writing files with file streams such as `std::ifstream` and `std::ofstream`, managing memory with vectors, and smart pointers. Although we were using RAII throughout the course, we did not actually explain how RAII works till the very end. The reason for this sequencing is that we introduced resource management and destructors only at the end of the course, and these concepts were necessary to understand how RAII works.

We aligned with our principle (1) in Section 3.3 by teaching low-level programming at the end. For example, we first taught how to use a library facility (e.g., vector APIs in week 3) before teaching how it works (e.g., internals of a vector in week 13).

3.8 Resources

We did not require a textbook for the course and all the materials we suggested were freely available on the Internet. We recommended the following resources:

(1) Readings
  (a) Keith Schwarz's C++ course reader [1]
  (b) David Kieras' lambdas [2] and smart pointers [3]
(2) C++ reference: http://en.cppreference.com
(3) Videos
  (a) ISOCPP [5]
  (b) Meeting C++ [6]

The URL for our course webpage is shown below:
http://pages.cs.wisc.edu/~gerald/cs368/

3.9 Programming Assignments

The order of assignments reflects the top-down approach that we used to teach modern C++. We covered the following topics in our programming assignments:

(1) Assignment 0: Programming fundamentals
(2) Assignments 1 - 2: C++ Standard Library
(3) Assignments 3 - 5: C++ Language Essentials
(4) Assignment 6: Low-Level Programming

We enabled students to write non-trivial programs as early as possible. In Assignment 0, students applied their prior knowledge of Java to write a simple C++ program that interacted with the user in the shell. In Assignment 2, students already had enough knowledge of C++ to implement a simple machine-learning model for predicting the ratings of movie reviews. This assignment required students to use associative containers, iterators, and algorithms from the C++ Standard Library. We note that it would be nearly impossible for students to complete a similar non-trivial assignment if they weren't taught file I/O and the C++ Standard Library right at the beginning of the course.

The design of our programming assignments aligned with our principle (2) in Section 3.3. The complete set of programming assignments that we used for our course can be found on our course webpage[6].

3.10 Issues Resolution

We resolved the four issues we identified with our previous syllabus (shown in Section 3.2) using our modified course content as follows:

(1) We focused more on teaching the high-level abstractions in modern C++ over the low-level details of C. Apart from the topics on raw pointers, dynamic memory allocation, and the differences between C and C++, all topics in our new syllabus focused entirely on C++.
(2) We taught the most useful topics in the C++ Standard Library (i.e., containers, iterators, and algorithms) at the beginning of our course (weeks 2 - 6). This enabled our students to use the features of the standard library throughout the course.
(3) We introduced the low-level topics only when they were necessary to achieve a particular task. For example, we taught the big three (copy constructor, copy assignment operator, and destructor) only when students were required to write their own custom vector class.
(4) We taught file I/O as one of our first topics (in week 2) since our students already knew file handling in Java. This enabled us to create non-trivial programming assignments involving file I/O starting from assignment 1.

4 STUDENTS' FEEDBACK

We collected feedback from the students using a survey that consisted of Likert scale questions and open-ended feedback. In this section, we present the feedback that we collected.

Throughout our course, we communicated with our students about the changes we made to the course content and ordering of topics for teaching modern C++. We showed the contents of the previous syllabus (shown in Table 1) to our students and explained our rationale behind the new syllabus (shown in Table 2) for teaching modern C++. In this way, our students were aware about the fact that they were learning modern C++ using a new and updated syllabus when compared to the previous semesters.

[6]http://pages.cs.wisc.edu/~gerald/cs368/assignments.html

4.1 Students' Rating of Top-Down Approach

We asked the following question to understand how many students liked our top-down approach:
'*How much did you like the top-down approach for learning C++?*' and the responses were collected using a five-level Likert scale survey (1 - not at all, 5 - completely). This question was answered by 64 students and the summary of student responses is shown in Figure 2. Students' responses were as follows: 85% of the students responded positively (rating of 4 or 5), 12% responded neutrally (rating of 3), 3% responded slightly negatively (rating of 2) and no students responded extremely negatively (rating of 1).

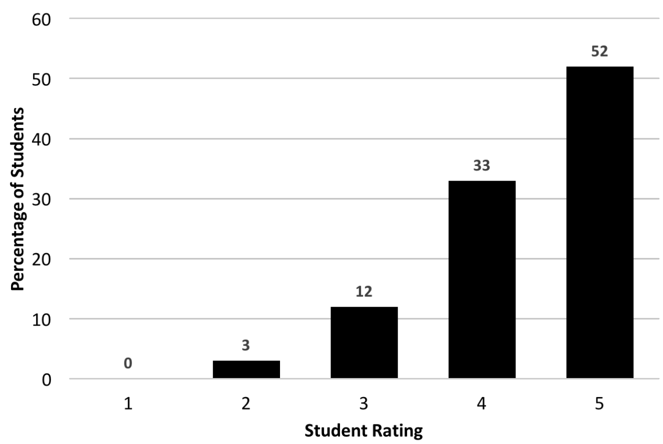

Figure 2: Student responses for the five-item Likert scale question 'How much did you like the top-down approach for learning C++?' where a rating of one (1) means 'not at all' and a rating of five (5) means 'completely'.

4.2 Open-Ended Feedback

To further understand the students' perceptions on our top-down approach, we also collected open-ended feedback from the students for the following question:
'What are your thoughts about the organization of the course during this semester?'

Some sample responses from the students are shown below:

"As compared to how this class was previously structured, this semester's curriculum was MUCH better. The course flowed really well and all topics were taught in a relevant, easy to understand order. Moreover, in the previous version, it seemed to focus a lot on C type C++, which could be learned in CS 354 [a course on machine organization and programming] when learning C anyway. The larger emphasis on modern C++ was really good."

"The way that this class was reordered and the new syllabus was great. It seemed like everything we learned kept building on each

other and always felt like a progression and not skipping anything. This class was extremely enjoyable and has furthered my want to learn more about C++."

"The course is fantastic. It is very well organized and laid out in a fashion which teaches fundamentals in a growing fashion for a first time experience with C++. I have no complaints about the course material."

"I like that the course was redesigned for this semester. The outline of the course seems well-organized and how it is laid out makes sense to me."

5 DISCUSSION

In this section, we summarize our findings about the students' perceptions on our top-down approach. We also highlight the difficulties we faced when teaching modern C++ using our updated curriculum, and suggest some recommendations to teachers who may be interested in adopting our syllabus and organization of topics for teaching modern C++.

5.1 Students' Perceptions

The overall feedback from the students was positive, which is evident from the responses to the Likert scale question shown in Figure 2 and the student responses to the open-ended question shown in Section 4.2. Based on these student feedback data, we summarize that most students have likely found our approach to be valuable when learning modern C++.

5.2 Challenges Faced

The challenges that we faced when teaching with the new format were primarily due to the way in which the topics were organized in our course. We highlight two of the major difficulties faced by the instructor and students in our course.

5.2.1 C++ Mapping to Hardware. In Java, every variable of a primitive type is passed by value, while every object is passed by reference. Both these types of variables have simple behavior, and so one can become a competent Java programmer without really understanding how references work.

In C++, the parameter passing convention is decoupled from the type of the parameter. The fact that objects are not automatically passed by reference confused many students since they come from a Java background. There were many students who were good Java programmers but struggled to become proficient C++ programmers.

The main reason for this issue is because we had not introduced the *C++ mapping to hardware* at the beginning of the course. We felt that those were low-level details without which students may be able to understand the C++ Standard Library. Although the students were able to use the facilities offered by the C++ Standard Library, most students were confused about why were we passing a container of objects using a reference as shown below.

```
void removeWhiteSpaces(std::vector<std::string> &tokens) {
    // TODO: Implement this method
}
```

We could have avoided this problem if we had warned the students right at the beginning about the differences between Java and C++ regarding the mapping to hardware, and continued on with using our top-down approach after letting the students know that this topic would be explored in detail in a later lecture.

5.2.2 References for Inheritance. The reader might wonder how we taught inheritance without introducing pointers. We followed an approach suggested by Kate Gregory in her talk [8] where she recommends using references instead of pointers for polymorphism. Although references can generally support polymorphism, we faced some issues when trying to store references in containers.

Containers cannot store references because the element type of containers must be assignable[7], but references are not assignable (i.e., a reference cannot be changed to refer to some other object once it has been initialized).

To overcome this problem, we used `std::reference_wrapper`, which wraps a reference in an assignable object. The following function signature shows how we wrote a polymorphic function using `std::reference_wrapper` that prints the details of a group of students.

```
/* @brief Prints the details of a group of students.
   @param students A vector of student references. */
void printStudents(
   const std::vector<std::reference_wrapper<Student>> &students);
```

We note that such a function signature may be confusing for students who are just starting to learn C++. It may have been syntactically easier if we had used pointers instead of references for inheritance.

5.3 Recommendations to Teachers

Based on our experiences teaching modern C++, we recommend the following for C++ instructors interested in adopting our approach.

(1) If instructors are teaching C++ for Java programmers, then starting with the C++ Standard Library is a great way to teach C++ as a modern language.
(2) An important change that we recommend is to teach a lecture on *C++ mapping to hardware* (i.e., references and pass by value vs pass by reference) right at the beginning of the course. We believe that this could reduce the difficulties that students face because of fundamental differences between Java and C++.
(3) Another change that we recommend is to introduce pointers at the same time as OOP, to avoid issues with using references for inheritance. In particular, some of the smart pointer types in C++11, rather than raw pointers, should be sufficient. Although pointers are a low-level topic, this organization aligns with Principle (4) in Section 3.3.
(4) The new programming assignments that we developed to reflect our top-down approach were crucial to teach modern C++ effectively. Therefore, we recommend that instructors use a similar approach when teaching modern C++. Instructors are most welcome to use our assignments for teaching their course.

[7] http://en.cppreference.com/w/cpp/concept/CopyAssignable

5.4 Limitations

The following are the limitations of our course.

(1) **Limited time:** This course had a total of 15 meetings, each of 50 minutes duration. This was very little time to introduce even the most important features of C++. For example, we did not have enough time to cover useful topics like *move semantics*, *rule of five*, *concurrency*, etc. Therefore, the total time we had for lectures was a limiting factor in our course.

(2) **Programming background:** The students in our course had varied programming backgrounds. Although all the students had learnt Java before, the knowledge and experience with the C programming language differed vastly among students. This created some problems within the classroom when students asked questions like, 'Are iterators similar to pointers?' before we had introduced pointers. Also, some students who knew C before were confused about the syntax of references, which uses the same symbol as the address-of operator (&) in C.

5.5 Future Work

The following are some ideas for future work in the area of teaching and learning a second programming language:

(1) Conduct a survey of the different ways in which C++ is taught as a second programming language by different instructors who teach modern C++. Understand the rationale behind their syllabus and organization of topics. This study may enable us to develop a standard approach for teaching modern C++.

(2) Compare the effectiveness of teaching modern C++ using a top-down approach versus a bottom-up approach. Teach C++ to two groups of participants. Use the top-down approach to teach one group and the bottom-up approach to teach the other group. Conduct a pre-test and a post-test to evaluate the effectiveness of these two approaches for learning modern C++.

(3) Evaluate the effectiveness of different approaches for teaching modern C++ by comparing the programming assignments that students would be able to complete at various points in time during the course when learning modern C++ using a particular approach.

6 CONCLUSIONS

In order to keep up to date with the changes in the C++ programming language, and to help create modern C++ programmers, it is necessary for C++ instructors to update their syllabus and ordering of topics when teaching modern C++. In this work, we have presented our experiences with such a curriculum change for teaching modern C++ using a top-down approach. We have also presented the students' perceptions on these changes, and they were mostly positive. Based on our experiences, we believe that the order in which the topics are taught is as important as the topics themselves. We hope that our work is a step towards answering a bigger question: *How do we teach a new programming language to someone who already knows programming in a different language?*

ACKNOWLEDGMENTS

This work was supported in part by grants from the National Science Foundation (NSF) under grant IIS-1250886. We thank Bjarne Stroustrup, the inventor of C++, for his valuable comments on our course organization and earlier versions of this paper. We also thank the graduate students at UW-Madison and software engineers at Google-Madison who provided useful feedback on our work. Our sincere thanks to all our students who were enrolled in our C++ course during the semester of Fall 2016, when we updated the syllabus and the ordering of topics. We also thank the anonymous reviewers for their valuable feedback on earlier drafts of our paper.

REFERENCES

[1] 2010. Keith Schwarz's C++ Course Reader. http://web.stanford.edu/class/cs106l/handouts/full_course_reader.pdf. (2010).
[2] 2015. David Kieras's Using C++ Lambdas. http://umich.edu/~eecs381/handouts/Lambda.pdf. (2015).
[3] 2015. David Kieras's Using C++ Smart Pointers. http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf. (2015).
[4] 2017. Erase-remove idiom in C++. https://en.wikipedia.org/wiki/Erase-remove_idiom. (2017).
[5] 2017. ISOCPP's C++ Videos. https://isocpp.org/blog/category/video-on-demand. (2017).
[6] 2017. Meeting C++ Videos. http://meetingcpp.com/. (2017).
[7] Ivaylo Donchev. 2013. Experience in teaching C++ 11 within the undergraduate informatics curriculum. *Informatics in Education-An International Journal* Vol12_1 (2013), 59–79.
[8] Kate Gregory. 2015. Stop teaching C. https://www.youtube.com/watch?v=YnWhqhNdYyk&t=2002s. (2015).
[9] Emily Howe, Matthew Thornton, and Bruce W Weide. 2004. Components-first approaches to CS1/CS2: principles and practice. *ACM SIGCSE Bulletin* 36, 1 (2004), 291–295.
[10] Andrew Koenig and Barbara Moo. 2000. *Accelerated C++: practical programming by example.* Pearson Education India.
[11] A Koenig and BE Moo. 2000. Rethinking how to teach C++ Part 1: Goals and principles. *JOOP-Journal of Object-Oriented Programming* 13, 7 (2000), 44–47.
[12] A Koenig and BE Moo. 2000. Rethinking how to teach C++ Part 2: Two interesting decisions. *JOOP-Journal of Object-Oriented Programming* 13, 8 (2000), 36–40.
[13] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 3: The first data structures. *JOOP-Journal of Object-Oriented Programming* 13, 9 (2001), 35–38.
[14] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 4: Emphasizing the library. *JOOP-Journal of Object-Oriented Programming* 13, 10 (2001), 25–27.
[15] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 5: Working with strings. *JOOP-Journal of Object-Oriented Programming* 13, 11 (2001), 29–32.
[16] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 6: Analyzing Strings. *JOOP-Journal of Object-Oriented Programming* 13, 12 (2001), 29–32.
[17] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 7: Payback time. *JOOP-Journal of Object-Oriented Programming* 14, 1 (2001), 36–40.
[18] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 8: An interesting revision. *JOOP-Journal of Object-Oriented Programming* 14, 2 (2001), 43–47.
[19] A Koenig and BE Moo. 2001. Rethinking how to teach C++ Part 9: What we learned from our students. *JOOP-Journal of Object-Oriented Programming* 14, 3 (2001), 44–47.
[20] Timothy Long, Bruce Weide, Paolo Bucci, David Gibson, Joe Hollingsworth, Murali Sitaraman, and Steve Edwards. 1998. Providing intellectual focus to CS1/CS2. 30, 1 (1998), 252–256.
[21] Timothy Long, Bruce Weide, Paolo Bucci, and Murali Sitaraman. 1999. Client view first: an exodus from implementation-biased teaching. 31, 1 (1999), 136–140.
[22] Bjarne Stroustrup. 1999. Learning standard C++ as a new language. *CC Plus Plus Users Journal* 17 (1999), 43–54.
[23] Bjarne Stroustrup. 2009. Programming in an undergraduate CS curriculum. In *Proceedings of the 14th Western Canadian Conference on Computing Education*. ACM, 82–89.
[24] Bjarne Stroustrup. 2014. *Programming: principles and practice using C++*. Pearson Education.
[25] Bjarne Stroustrup. 2016. C++11 - the new ISO C++ standard. http://www.stroustrup.com/C++11FAQ.html. (2016).
[26] TIOBE. 2017. TIOBE index for C++. https://www.tiobe.com/tiobe-index/. (2017).