How to teach “modern C++” to someone who already knows programming?

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 our university 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 new approach. We find that with our new approach, students’ perceptions about learning C++ is largely positive.

CCS CONCEPTS
• Social and professional topics → Computer science education;

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

ACM Reference format:

1 INTRODUCTION
C++ is one of the most popular programming languages today. It is currently ranked as the third most popular language next to Java and C [27]. 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." [26]. 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 harmful for learning modern C++ [10, 23]. Since C++11 is considered as 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 that 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 programming language to someone who already knows some other programming language(s)?”

2 RELATED WORK
Bjarne Stroustrup [23] 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 argues to use 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 teaching modern C++ (C++11 and beyond) using a top-down approach.

Accelerated C++ [12], 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 the year 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 [13–21], Koenig and Moo explain the rationale behind the organization of topics in their C++ course at Stanford University. These articles show that the order in which we teach is as important as what we teach. We have adopted these principles to organize the content for teaching modern C++.

Howe et. al. propose a components-first approach to teach introductory CS courses [11]. They compare and contrast two such approaches namely Koenig-Moo (KM) version [12] and Reusable Software Research Group (RSRG) version [9, 22]. Our approach is similar to their approach, but our goal isn’t to teach component-based software engineering but instead to teach modern C++ using a top-down approach.

Ivaylo Donchev has shared his experiences teaching C++11 to undergraduates [8]. The report mainly focuses 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 teaching C++ in an undergraduate freshmen course [24]. 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++ [25] 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 similar philosophy with a larger focus on the C++ standard library.

3 COURSE DETAILS

The C++ course in this study was taught at a large, research-intensive, public university during Fall 2016. It was a one credit course with one 50 minutes 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 7 programming assignments, 1 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 Principles for Course Content Organization

The course content was organized based on the following principles from Koenig and Moo [12]:

(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.

3.2 Previous Course Content

The contents of the 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 on the last day of class.

Table 1: Syllabus for C++ course till Spring 2016

<table>
<thead>
<tr>
<th>Week #</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>course information, history, high-level differences, process of writing C++ programs using Linux tools</td>
</tr>
<tr>
<td>2</td>
<td>constants, enumerations, structures, arrays</td>
</tr>
<tr>
<td>3</td>
<td>enum, struct, arrays, vectors, parameter passing modes</td>
</tr>
<tr>
<td>4</td>
<td>pointers to structs/classes, arrays, dynamic allocation</td>
</tr>
<tr>
<td>5</td>
<td>abstract memory model, reference variables, passing params to and return values from functions</td>
</tr>
<tr>
<td>6</td>
<td>.h and .cpp files, defining classes, multi-file compilation</td>
</tr>
<tr>
<td>7</td>
<td>makefiles, constructor, member initialization</td>
</tr>
<tr>
<td>8</td>
<td>copy constructor, copy assignment, destructor</td>
</tr>
<tr>
<td>9</td>
<td>operatorX syntax and use, member vs. non-member options (assignment and arithmetic operators)</td>
</tr>
<tr>
<td>10</td>
<td>explicit, member / non-member function pairs</td>
</tr>
<tr>
<td>11</td>
<td>overloading, condition states, string class, C strings</td>
</tr>
<tr>
<td>12</td>
<td>file I/O, manipulators, C I/O</td>
</tr>
<tr>
<td>13</td>
<td>templatized functions and classes, containers</td>
</tr>
<tr>
<td>14</td>
<td>iterators, generic algorithms, function objects</td>
</tr>
</tbody>
</table>

3.3 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 students in our course were equipped to write such a program using the C++ Standard Library after the second lecture.

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

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

3.4 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: (1) File I/O and Strings, (2) Sequence Containers
(Vectors), (3) Associative Containers (Sets and Maps), (4) Algorithms (std::find, std::sort, std::reverse, std::transform), (5) 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.

### 3.4.1 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

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

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

This code snippet deletes all the elements in the vector `v` that has a value of 3. The `remove` 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 doesn’t specify what those elements should be and hence it is implementation dependent. After the `remove` does its work, the size of the vector is unchanged since `remove` does not have the functionality to delete elements from the vector. Instead, a subsequent operation is needed to delete the unwanted elements using the `erase` member function of the vector class, which has the power to directly modify the structure of the container.

The `erase` member function takes 2 arguments namely, the first and the last iterators and deletes 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 two elements shown as `?` at the end of the vector are deleted and the vector size changes from 5 to 3 elements. This common C++ technique to eliminate elements that satisfy a certain condition is popularly known as the "erase-remove idiom" [4].

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

#### 3.4.2 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, the second assignment in the course required the students to read a file with movie reviews and ratings, and then predict the ratings for new reviews. This assignment is naturally done in modern C++ using constructs in 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.)

#### 3.4.3 Why teach lambdas?

We taught lambdas early in the course since it is an important construct that enables using the C++ Standard Library more effectively. Lambdas are heavily used in code examples of the Algorithms Library (e.g. `std::remove_if`, `std::sort`). 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 its 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: \[
\text{square}(x) = x \times x
\]

In lambda calculus:
\[
x \rightarrow x \times x
\]

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

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

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

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.

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

Once we finished teaching the first component of our course (i.e., C++ Standard Library), we realized that 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 two of the three principles that our course was based on. i.e., Principles (1) and (3) outlined in Section 3.1.

### 3.5 C++ Language Essentials

The next component focused on multiple essential topics in the C++ core language. The topics that were covered in this component of the course were: (1) Object-Oriented Programming (OOP), (2) Operator Overloading, (3) Generic Programming, and (4) Error Handling.

Since the students in the course had already learnt OOP in Java, we didn’t spend time teaching the concepts of OOP, but instead spent more time teaching C++ specific details including syntax. The topics we focused in OOP were: (1) member initializer lists, (2) splitting the interface and implementation of a class between the .hpp and .cpp files respectively, and (3) virtual functions & pure virtual functions.

Next we needed to introduce to the students the C++ mapping to hardware; i.e., reference types and pointer types. Since the students came from a Java background, they were already aware about the primitive data types like int, double, char, etc., and had a good understanding about references in Java. To understand OOP within the C++ context, one needs to understand how references in C++ work, as that concept is necessary to pass a vector of objects to functions to avoid unnecessary copying of data (a key part to the efficiency of C++ implementations).

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.

```cpp
// 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 3.1, we took a detour introducing references in C++, before diving into OOP (for C++).

### 3.6 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: (1) Raw Pointers and Smart Pointers, (2) Dynamic Memory Allocation, (3) Rule of 3 & Rule of 0, and (4) Resource Acquisition Is Initialization (RAII).

Up until this point in our course, we had not discussed pointers, since it wasn’t necessary. The rationale behind not teaching pointers till the very end of the course was that we wanted to focus on the higher-level abstractions in C++ first before we dwell 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 works in the background. We discussed how the vector may resize itself when new elements are added. This helped the students understand that the vector class manages 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 memory that was allocated in a vector class.

Next, we discussed how copying works in std::vector. We also showed that in the vector class we had created, copying doesn’t 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 to appreciated their need for making a deep copy of the 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.

Although we were using the “resource acquisition is initialization” (RAII) programming idiom right from the beginning of the course (e.g., reading and writing files with file streams std::ifstream and std::ofstream, managing memory with vectors, smart pointers) we didn’t actually explain how that 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 are necessary to understand what RAII means and how it works.

3.7 Resources

We didn’t require a textbook for the course and all the materials we suggested were freely available on the Internet. We used the following resources:

(1) Readings
   (a) Keith Schwarz’s C++ Course Reader [1]
   (b) David Kiera’s Lambdas [2] and Smart Pointers [3].
(2) C++ Reference: http://en.cppreference.com
(3) Videos
   (a) ISOCPP [5]
   (b) Meeting C++ [6]

3.7.1 Programming Assignments. From the students’ feedback we found that one of the most important things that helped them learn modern C++ were the programming assignments. These assignments were written in a way to reflect the top-down approach that we used to teach modern C++. The programming assignments that we used in our course covered the following topics:

(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 aren’t releasing the assignments in this manuscript, since we want to maintain our anonymity. The complete set of assignments that we used for our course will be released in our camera-ready version.

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.

4.1 Students’ Rating of Top-Down Approach

We asked the following questions 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). 64 students answered this question and the summary of student responses are shown in Figure 2. 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 student responded extremely negatively (rating of 1).

4.2 Open-ended Feedback

To further understand the students’ perceptions of our new teaching 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:

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”.

“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 X (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’ perception on our top-down approach. We also highlight the difficulties we faced when teaching modern C++ using our updated curriculum. We also 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, there are only two types of variables namely, primitive type variables and object type variables. Primitive variables are stored directly on the stack whereas object variables are implicit references. Both these types of variables have simple behavior and so one can become a competent Java programmer without really understanding how the stack and the heap work.

In C++, there are references and pointers. Objects aren’t automatically passed by reference, which was confusing to 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 we were passing a container of objects using a reference as shown below.

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

We could have avoided this problem if we had warned the students about this difference, and continued on with our approach of using a top-down approach and letting the students know that this topic will be explored in detail in a later lecture.

5.2.2 References for Inheritance. The reader may wonder how did we teach inheritance without introducing pointers. We followed an approach suggested by Kate Gregory in her talk [10] where she recommends using references instead of pointers for polymorphism. Although references work with polymorphism, we faced some problems doing so because references cannot be directly stored in vectors.

The reason why we cannot store references in a vector is because the element type of containers like vectors must be assignable\(^6\) but references are not assignable (i.e., we cannot change a reference 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 a copyable, assignable object and so they may be stored in standard containers like vectors. Even though this is a straightforward way to deal with this problem, we felt that it may have been syntactically easier if we had used pointers instead of references for inheritance.

The following function shows how we used `std::reference_wrapper` to write a polymorphic function that prints the details of a group of students. We note that a function signature like the one given below may be confusing for students who are learning C++ for the first time.

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

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 you 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. If we teach the course again, one important change that we would make is to teach a lecture on C++ mapping to hardware (i.e., references, pass by value vs pass by reference) right at the beginning of the course. We believe that this could reduce the difficulties faced by students due to how things work differently under the hood in Java and C++.
3. One of the important things that helped us teach modern C++ effectively are the new programming assignments that we developed to reflect our top-down approach. Therefore, we recommend instructors to use a similar approach when teaching modern C++.

Instructors are most welcome to reuse our assignments for teaching their course. We are planning to release our assignments along with the camera-ready version of our paper.

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 didn’t have enough time to cover useful topics like move semantics, rule of 5, 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 as it uses the same symbol as the address-of operator (&) used in C.

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 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. 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?"

REFERENCES