Teaching Hardware to Demystify Foundational Software Concepts

Dr. Christopher Miller, Rose-Hulman Institute of Technology

Chris is an Assistant Professor of Electrical & Computer Engineering at Rose-Hulman Institute of Technology. His interests include engineering education, embedded systems, and ubiquitous computing.

Teaching Hardware to Demystify Foundational Software Concepts

Abstract: Both top-down and bottom-up approaches have been proposed for software and programming education. Motivations can be cited for both approaches, but empirical data for either approach can be difficult to obtain. In this paper, we explore potential benefits of a bottom-up approach which begins at the architecture and machine language level. Abstractions of basic software concepts such as data storage and pointers can lead to misconceptions. Understanding how these abstractions are implemented in the underlying hardware can provide clarity of foundational software concepts.

An introductory course on embedded systems and microcontrollers for electrical and computer engineers was modified in an attempt to strengthen student understanding of foundational software concepts. The material covered in the course primarily remained the same, but the course schedule was modified to move the system architecture and instruction set material to the beginning of the course, rather than the end. Data was collected for common exam questions for offerings both prior to and following the course modification. The data indicates that students who were exposed to the functionality of the underlying architecture prior to high-level programming languages had a better understanding of basic concepts such as storage allocation and referential pointers.

This paper contributes to the fields of education in electrical and computer engineering and computer science by providing data on student outcomes for alternate approaches to content delivery. We hope that this information is useful in curriculum design and development for related fields.

Introduction

When teaching a course based in programming or structuring a curriculum with emphasis in programming there are two approaches which may be pursued: bottom-up and top-down. In a bottom-up approach[1,2,3], students are first introduced to basic low-level concepts, and the course continues to build upon past-learned concepts, thus forming a foundation for future concepts. The course gradually builds up to a more high-level abstracted representation of programs. In a top-down approach[4,5,6], students are first introduced to high-level, abstracted representations of programs, and gradually dig deeper into the details of implementation, towards the low-level concepts. This can have the benefit of abstracting the complications of low-level implementation, allowing students to first become familiar and comfortable with basic programming concepts while working with higher-level languages. Benefits for both approaches can be stated, but quantifying the differences can be difficult since there is rarely an opportunity for applying the same measure of evaluation to both approaches.

In a sophomore-level introduction to microcontrollers course, the course outline was modified to address difficulties in student comprehension of the impact of program design on the underlying hardware. The original course began with programming in C, which provided a higher level of

abstraction. Towards the end of the course, the instruction set architecture was introduced along with assembly programming. For most students, this was their first experience with either language. Many students struggled with understanding the C programming language. They also struggled to switch to the lower-level of abstraction when assembly programming was introduced. Since the instruction set was introduced in the latter portion of the course, it aligned with the more complex projects, which made these projects particularly tedious since they had to be implemented in assembly.

To address these issues, the outline of the course was modified to begin with an introduction to the instruction set architecture (ISA) and assembly programming, and later introduce programming in C. There were several intended benefits to this switch. By starting with the ISA, students would have a better understanding of data storage on the device and how variables correlate to physical storage. Students would better understand how special function registers are connected to the underlying hardware. With a better understanding of data storage on the physical hardware, students would have a better understanding of how C pointers function. Since the course begins with assembly programming and ends with programming in C, assembly programming is only needed for earlier projects, which tend to be smaller and simpler implementations. Later projects, including the final project, can now be implemented in C, which allows students to be more creative and attempt more adventurous projects.

While the order of material in the course outline changed, the material covered was largely the same, and thus similar exam problems were used for both versions. We therefore have a consistent set of evaluations under each method for similar population samples. In this paper, we focus on the impact to student understanding of C pointers. This tends to be a challenging topic for many students, leading to many misconceptions of relational operations. It was hypothesized that by first helping students to understand how data is stored on the physical hardware, and how data type assignments correlate to storage, that pointers would be less of a mystery, and students would exhibit a better understanding of their implementation. We present the course outline both prior to and following the transition, and the common exam questions which were used in evaluation. We show student performance on the common exam questions to evaluate improvement in student understanding.

Methodology

i. Original course design

This course is offered in a 10-week, quarter-based institute. In the original outline of the course, the first eight weeks were primarily focused on working with microcontrollers and developing embedded systems using the C programming language. In the last two weeks, we introduced the instruction set architecture and discussed microcontroller organization and assembly programming. Since the majority of students had no experience with C prior to this course, the first two weeks were primarily dedicated to a quick ramp-up in C programming. Hands-on lab and project work is an integral part of the course. There are weekly lab projects, except for the final two weeks when students focus on a final project. We use an 8-bit Microchip PIC16 microcontroller, which has a simple RISC architecture with only 35 instructions. An outline of

the course is shown in Table 1. The integral labs are shown in Table 2. These labs will vary a bit from year to year, but each lab will generally cover the same concepts, with slightly different implementations. Each of these labs were completed using the C programming language. The final project assignment was completed using assembly programming.

Table 1. Original outline of course.

Week	Topics
1	Introduction, circuit review, software development tools, and programming in C overview
2	Programming in C and the PIC microcontroller, basic I/O and mechanical switch debounce methods
3	Hardware timers, compare modules, interfacing with LCD displays and internal pull-up resistors
4	Hardware timers with interrupts, compare/capture modules, interfacing with keypads, and driving high-current loads
5	RS232 Universal Asynchronous Receiver-Transmitter (UART) serial communication, framing and parity
6	Inter-integrated circuit (I^2C) synchronous serial communication, temperature sensors, and stepper motors
7	Servo motors, pulse width modulation, and analog-to-digital conversion
8	PIC architecture and instruction set
9	Assembly programming
10	Final project completion

ii. Updated course design

Between the 2014-2015 academic year and the 2015-2016 academic year, the course was updated to the new outline. The coverage of material is mostly unchanged in the new outline, with the primary difference being the move of instruction set architecture and assembly programming to the beginning of the course. This necessitated some modifications to the labs, since the early labs must now be completed in assembly. This was a good fit, since the earlier labs tend to be shorter and simpler programs, and thus are more easily accomplished in assembly than would be projects later in the course. The final project was now completed in C. This emboldened students to be much more creative and bold in their project goals. One drawback of the revised outline is it resulted in a more compressed schedule, particularly the first seven weeks, since all topics must be covered which are necessary in order for the students to complete the labs. Tables 3 and 4 show the updated course outline and lab schedule. Labs 1 thru 3 are completed in assembly, and the remaining are completed using the C programming language.

Table 2. Example of original course labs.

Lab	Title	Description
1	Pushbutton and LED	Familiarize students with lab kits, microcontrollers, development software and basic circuit concepts. Timing delays – fixed-length instruction loops.
2	Pushbutton input with timing	Familiarize students with testing/debugging tools. Introduce timers and de-bouncing concepts. Timing delays – timer with overflow flag.
3	LED whack-a-mole game	Introduce 16-bit timers and compare modules. Introduce function calls and parameter passing. Timing delays – timer with compare module.
4	Music note player and frequency tuner	Introduce capture module and compare with output effect. Introduce LCD display, potentiometer, speaker, nMOSFET, square-wave signals. Introduce interrupts. Timing delays – timer and compare module with interrupts.
5	Remote combination lock system	Introduce UART and serial communication – baud clock generation, data framing, parity. Introduce keypad input.
6	Temperature control system	Introduce I^2C serial communication – data framing, addressing. Introduce stepper motors.
7	Light sensor with servo	Introduce analog-to-digital conversion. Analog input sources. Introduce PWM and servos.

Table 3. Updated outline of course.

Week	Topics
1	Introduction, software development tools, and PIC microcontroller architecture
2	PIC instruction set, assembly language programming, hardware timers, internal pull-up resistors, basic I/O and mechanical switch debounce methods
3	Programming in C, timers, and compare modules
4	Hardware timers with interrupts, compare/capture modules, and driving high-current loads
5	Servo motors, pulse width modulation, analog-to-digital conversion, and interfacing with LCD displays
6	RS232 Universal Asynchronous Receiver-Transmitter (UART) serial communication, framing and parity, and interfacing with keypads
7	Inter-integrated circuit (I^2C) synchronous serial communication, temperature sensors, and stepper motors
8	Embedded system design principles and advanced topics
9	Final project design and modern development tools
10	Final project completion

Table 4. Example of updated course labs.

Lab	Title	Description
1	Pushbutton and LED	Familiarize students with lab kits, microcontrollers, development software, assembly programming and basic circuit concepts. Timing delays – fixed-length instruction loops.
2	Double-click detector	Familiarize students with testing/debugging tools. Introduce variable data storage, timers and de-bouncing concepts. Timing delays – timer with overflow flag.
3	LED quick reaction game	Introduce 16-bit timers and compare modules. Introduce function calls and parameter passing in assembly. Timing delays – timer with compare module.
4	Music note player and frequency tuner	Programming in C. Introduce capture module and compare with output effect. Introduce interrupts. Introduce potentiometer, speaker, nMOSFET, square-wave signals. Timing delays – timer and compare module with interrupts.
5	Light sensor with servo	Introduce analog-to-digital conversion. Analog input sources. Introduce LCD display, PWM and servos.
6	Remote combination lock system	Introduce UART and serial communication – baud clock generation, data framing, and parity. Introduce keypad input.
7	Temperature control system	Introduce I^2C serial communication – data framing, addressing. Introduce stepper motors.

Results

Since the material covered with both versions of the course is primarily the same, and only the order of delivery has been changed, the examination questions also remained mostly unchanged other than the order. This provided a common metric for evaluation of student understanding of C programming concepts, such as C pointers. In both cases, an introduction to programming in C occurred in the first half of the term, and was thus covered on the midterm exam. There were three questions pertaining to programming in C on the exams – each with multiple sub-parts. The first question primarily measured student understanding of C data types, data storage, and declarations. The second question primarily measured student understanding of program flow concepts such as loops, conditional blocks, function calls, and basic C operations. This question included concepts such as pass-by-value, pass-by-reference and arrays, so it has some connections with C pointer concepts. The third question focused on student understanding of C pointers. An example of a portion of the question is shown in Figure 1 (students were instructed to assume a 1-byte addressable system for these types of questions).

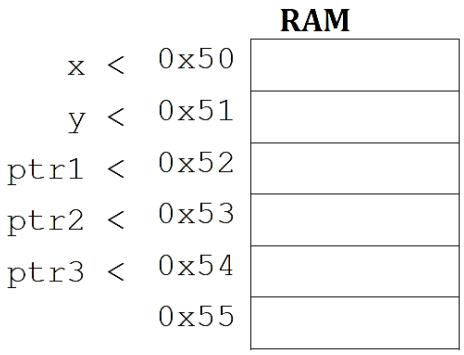
In the delivery of material on C pointers, the discussion always connected the implementation to the effect on the underlying hardware (data storage). It was believed that this would strengthen the student understanding of the functionality of C pointers, and they would thus be less of a mystery. Students, however, continued to perform very poorly on these questions. It typically scored one of the lowest question averages on the exam, indicating a poor student understanding of the function of C pointers. It was believed that the change in the course outline to introduce the underlying organization and instruction set architecture would help to address this gap in

understanding. In the year following the course outline change, students performed dramatically better on this question, resulting in one of the highest averages among questions on the exam. That trend has continued in the current year, though we only have a limited sample thus far. Figure 2 shows the students' average for the question on C pointers for the year prior to the change and following, as well as the current year (limited sample). The 95% confidence intervals are also provided. Data was captured over multiple sections for each year. The 2014-2015 academic year data captures 91 students across three sections. The 2015-2016 academic year data captures 78 students across three sections. The current academic year data only captures a single section of 25 students, and thus has a wider confidence interval. We can see that the average climbed nearly 25%, and that gain has thus far been maintained into the current year.

Figure 1. Example of question on exam problem relating to C pointers.

*Given the following declarations, and assuming that the variables are allocated to memory as shown below, fill in the RAM for each of the allocated bytes to indicate what value is stored at that location. (**Put values in hexadecimal**)*

```
char x = 0x5, y = 0xF;
char *ptr1 = &x;
char *ptr2 = &y;
char *ptr3 = ptr1;
```



The averages for the other questions related to programming in C were also evaluated, though they were less conclusive. Shown in Figure 3 are the student averages for the exam questions pertaining to C data types and storage and program flow in C. There is an indication of improvement in student understanding of program flow, which could be a result of improved understanding of pass-by-value and pass-by-reference functionality. Further analysis is needed, however, of the scores of the individual sub-parts of this problem to identify where exactly the gains occurred. For the question on C data types and storage, there were modest gains in the year following the course change, but oddly the average dipped below the original average this year. There are several factors that may have resulted in this phenomenon. First, there were some new sub-parts added to this question this year, which differed from the style of question in previous years. Second, this is a limited sample size (only one section), so it may change after further samples are captured. Third, there was not much room for improvement on this question, since the average was already in the mid-80s, indicating that students did not have much difficulty in understanding these basic concepts.

Figure 2. Student averages (per academic year) on exam question relating to C pointers. 95% confidence intervals included.

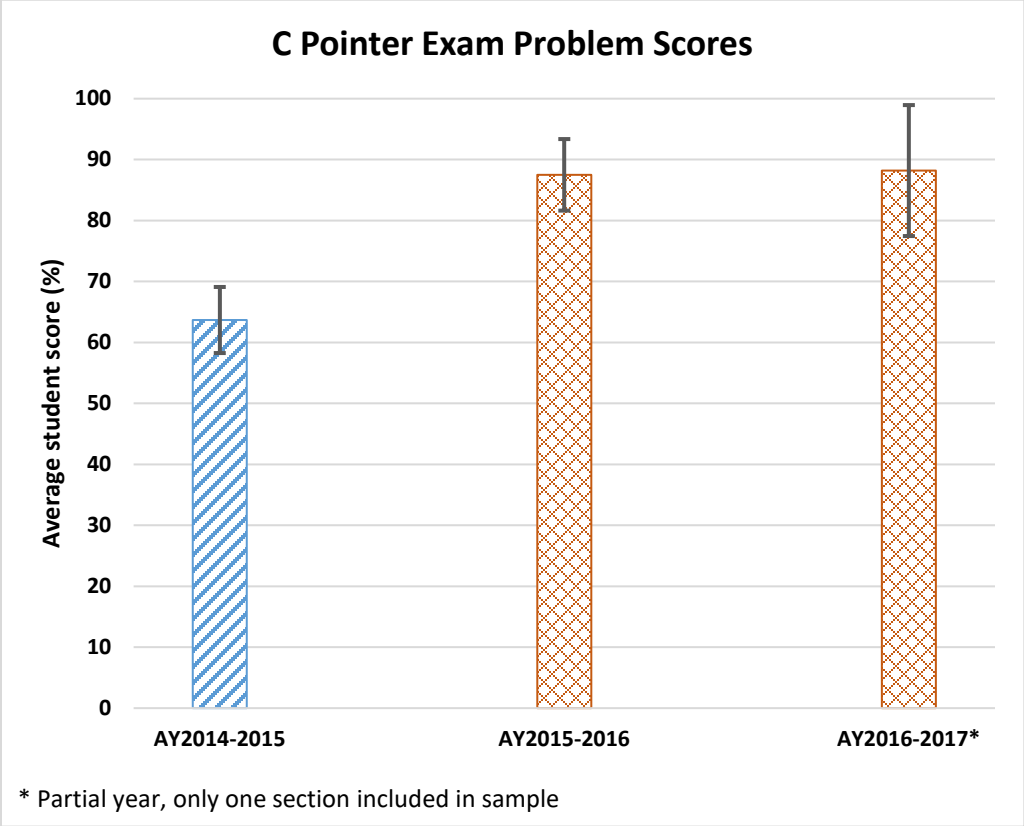
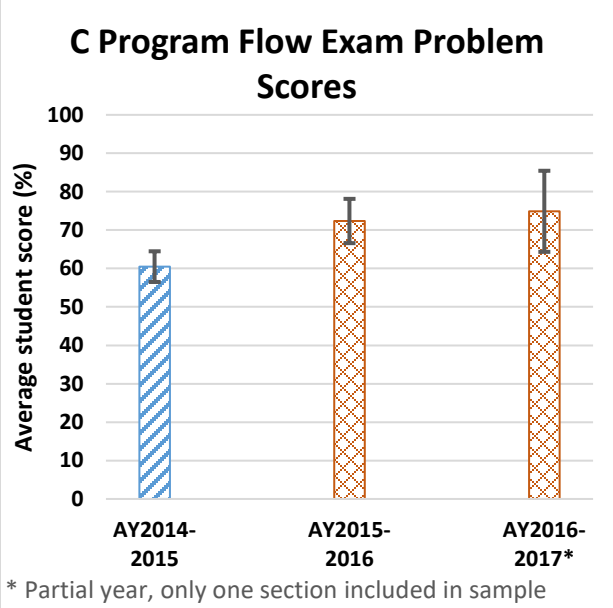
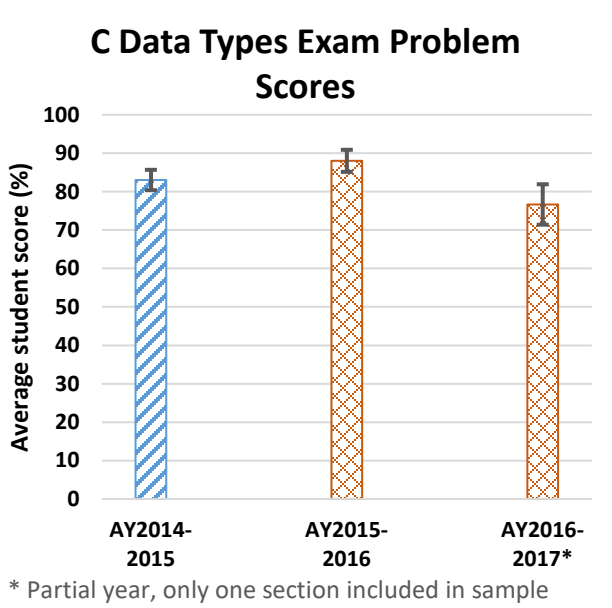


Figure 3. Student averages (per academic year) on exam questions relating to program flow and C data types. 95% confidence intervals included.





Conclusions and future work

The data indicates that providing students with a background in the underlying architecture of the hardware does benefit their understanding of the functionality and implementation of C pointers. Benefits to their understanding of other core concepts in C is not clear from the data. There are a number of other factors which may affect student understanding. Students entering the course do not all have the same background, and in particular, the same programming experience. Some students have already worked with C, C++ or Java extensively, and some have prior experience with microcontrollers or Arduinos. For the majority of students entering the course prior to the 2016-2017 academic year, however, their only programming experience was with an introductory course in Python. Beginning in the 2016-2017 academic year, students entering the course had prior experience with Arduinos, which uses a language built on C for development. While it is not possible to fully isolate the factors impacting student understanding, the significant and consistent increase in student understanding of C pointers would indicate that the change in the order of delivery of course material has resulted in gains in student understanding.

Further analysis of the data will be made to obtain greater detail of changes in student understanding on specific concepts within C by looking at performance on the individual sub-topics within each question. Data will continue to be gathered for future sections of the course to evaluate consistency.

References

1. Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas. 2006. A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bull.* 38, 4 (June 2006), 182-194.

2. Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (ITiCSE '11). ACM, New York, NY, USA, 168-172.

3. Judy Sheard and Dianne Hagan. 1998. Our failing students: a study of a repeat group. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education* (ITiCSE '98). ACM, New York, NY, USA, 223-227.

4. Rick Decker and Stuart Hirshfield. 1993. Top-down teaching: object-oriented programming in CS 1. In *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education* (SIGCSE '93). ACM, New York, NY, USA, 270-273.

5. Margaret M. Reek. 1995. A top-down approach to teaching programming. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education* (SIGCSE '95), Curt M. White, James E. Miller, and Judy Gersting (Eds.). ACM, New York, NY, USA, 6-9.

6. Peter Van Roy, Joe Armstrong, Matthew Flatt, and Boris Magnusson. 2003. The role of language paradigms in teaching programming. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (SIGCSE '03). ACM, New York, NY, USA, 269-270.