2018 ASEE ANNUAL CONFERENCE & EXPOSITION
JUNE 24–27, 2018
SALT PALACE CONVENTION CENTER
SALT LAKE CITY, UT

Paper ID #21439

An Analysis of Common Errors Leading to Excessive Student Struggle on Homework Problems in an Introductory Programming Course

Nabeel Alzahrani, University of California, Riverside

Nabeel Alzahrani is a computer science PhD student at the University of California, Riverside. Nabeel's research interests include causes of student struggle, and debugging methodologies, in introductory computer programming courses.

Prof. Frank Vahid, University of California, Riverside

Frank Vahid is a Professor of Computer Science and Engineering at the Univ. of California, Riverside. His research interests include embedded systems design, and engineering education. He is a co-founder of zyBooks.com.

Dr. Alex Daniel Edgcomb, zyBooks

Alex Edgcomb finished his PhD in computer science at UC Riverside in 2014. Alex works in research and development at zyBooks.com, a startup that develops interactive, web-native textbooks in STEM. Alex has also continued working as a research specialist at UC Riverside, studying the efficacy of web-native content for STEM education.

Prof. Roman Lysecky, University of Arizona

Roman Lysecky is an Associate Professor of Electrical and Computer Engineering at the University of Arizona. He received his Ph.D. in Computer Science from the University of California, Riverside in 2005. His research interests include embedded systems, runtime optimization, non-intrusive system observation methods, data-adaptable systems, and embedded system security. He has recently coauthored multiple textbooks, published by zyBooks, that utilize a web-native, interactive, and animated approach, which has been shown to increase student learning and achievements.

Dr. Susan Lysecky, zyBooks

Susan received her PhD in Computer Science from the University of California, Riverside in 2006. She served as a faculty member at the University of Arizona from 2006-2014. She has a background in design automation and optimization for embedded systems, as well as experience in the development of accessible engineering curricula and learning technologies. She is currently the Director of Content at zyBooks, a startup that develops highly-interactive, web-native textbooks for a variety of STEM disciplines.

An Analysis of Common Errors Leading to Excessive Student Struggle on Homework Problems in an Introductory Programming Course

Abstract

Students make many errors in an introductory programming course (aka CS 1). While previous research reports common errors, some errors are normal, being corrected by students in a reasonable amount of time, and being part of the learning process. However, some errors may lead to frustration due to excessive struggle, which may lead to student attrition. We defined a struggle metric using a combination of excessive time spent and excessive attempts, relative to other students in a course and reasonable thresholds. We analyzed struggle on 78 short, auto-graded coding homework problems for an 80-student Spring 2017 introductory C++ programming course at a research university. We found the struggle rate to be 10-15%. Our main focus was to determine the errors that led to such struggle, and thus we manually examined the student submissions for the 10 homework problems having the highest struggle rates. We described the errors and potential underlying student misconceptions that seemed to lead to that struggle. We found that most common errors belong to the following: nested loops, else-if vs. multiple if, random range, input/output, for loop and vector, for loop and if, vector index, negated loop expression, and boolean expressions. Having a deeper understanding of these common errors may aid teachers and authors to help students avoid or correct such errors, thus reducing struggle, which may reduce frustration and potential attrition.

Introduction

Frustration is a common source of attrition or dissatisfaction in introductory programming courses [1][2], also known as CS 1 courses. A common cause of frustration is getting "stuck", what we call struggling, on programming tasks, wherein students spend excessive time or make excessive attempts on a problem with little progress.

Previous work examined common errors encountered by students in introductory programming courses [3][4][7][8][9][11]. Spohrer [3] analyzed programming errors using a cognitive science model. Spohrer used a Goal And Plan tree to trace the root causes of errors, which defined plans (steps/procedures) as the techniques to solve the problem, and the goals as the desired result to achieve or accomplish. Spohrer found that once there is a mismatch between a plan and a goal, an error occurs. Yarmish used a similar approach but added more components to a plan [4].

Other work suggests that errors occur due to inaccurate mental models about program state [3][4][7][13][14]. Horstmann [5] presents a list of common errors when introducing C++ programming concepts and constructs. Horstmann presents common errors in each chapter, which may help students avoid such errors. Oualline [6] devises debug examples that ask the reader to find and correct those errors. Ginat suggests learning from student errors by building debugging examples based on students misconceptions of object oriented programming [10].

We focus specifically on errors that lead to student struggle in a CS 1 course. A closely related work is Cherenkova [12], who strives to find challenging errors in programming. Cherenkova's metric is number of attempts. Our metric considers both total time and number of attempts, and normalizes by comparing with the top 20% of students in a given class, and include reasonable thresholds to ensure the normalizations are reasonable. We note that many errors are not necessarily problematic. Students naturally make errors when learning new programming constructs, and in many cases quickly notice the error, correct the error, and learn from the experience. Problematic errors are those errors that seem to cause students to struggle, meaning the student was spending a lot of time or making numerous attempts on a problem. Such a situation could potentially cause the student to become frustrated, to feel out of place, or to feel alone, etc. Thus, we sought to focus not on any errors, but specifically on errors that caused large amounts of struggle.

In this paper, we describe our study of errors leading to struggle on auto-graded homework problems in an introductory C++ course. We introduce the type of homework problem used in the study. We define a quantitative metric for “struggle” and show the struggle rates across all 78 homework problems in our course of 80 students. We highlight the homework problems having the highest struggle rates, and summarize our manual investigation of the programming errors that seemed to lead to such struggle.

Knowing those errors may help teachers and publishers develop techniques or content that helps students avoid such errors. As an example, we showed that adding a hint to certain homework problems reduced struggle rates by 17%; more aggressive prevention and intervention should see even more reductions.

To be clear, we believe some struggle is a normal part of learning to program. Our goal is to reduce excessive struggle that goes beyond normal learning and instead may cause frustration and ultimately attrition.

Homework problems

We used online content that had several coding homework problems per content section [15]. Each homework problem is called a coding activity (CA). Each CA has the student complete some existing code to achieve a goal, typically only requiring about 1-10 lines of student code.

The student cannot modify the template code except in the region indicated. The student can submit their solution, and the system automatically compiles the program and runs the program with various test values, comparing with expected output. The student can re-attempt each activity as many times as desired. Figure 1 shows an example CA, on the hard end of the spectrum, requiring the student to create nested loops; many students struggled with this activity.

Figure 1 shows a sample CA, 5_3_2_Nested_loops_Print_seats, with a link "Download student submissions". A teacher clicks that link to get all student submissions for that CA. Table 1 shows a tiny snapshot of that file, showing the corresponding submissions for one particular student for CA 5_3_2_Nested_loops_Print_seats. The downloaded file has 4 columns: time of submission (timestamp), user ID# (occluded above as 0xxxx), whether the answer is correct (Yes/No), and the actual code submitted by the student (including whitespace).

The course was CS 1 in C++, having 80 students who were all non-computing majors, mostly in engineering (chemical engineering, bioengineering, etc.) and science (biology, chemistry, physics, math, etc.). The university is a research institution, and the CS department is typically ranked around 50-70 among U.S. CS programs. The content had 78 code activities, distributed through 9 chapters, covered in 9 weeks of a 10-week course. The chapters were: Intro/Input/Output, Variables/Assignments, Branches, Strings/Loops1, Loops2, Functions1, Functions2, Vectors1, and Vectors2. The CAs were due on the Sunday at the end of the week the subject was covered in class, and worth 5% of the course grade. On average, student completion of CAs each week was: 98%, 96%, 95%, 85%, 87%, 88%, 88%, 82%, and 74%.

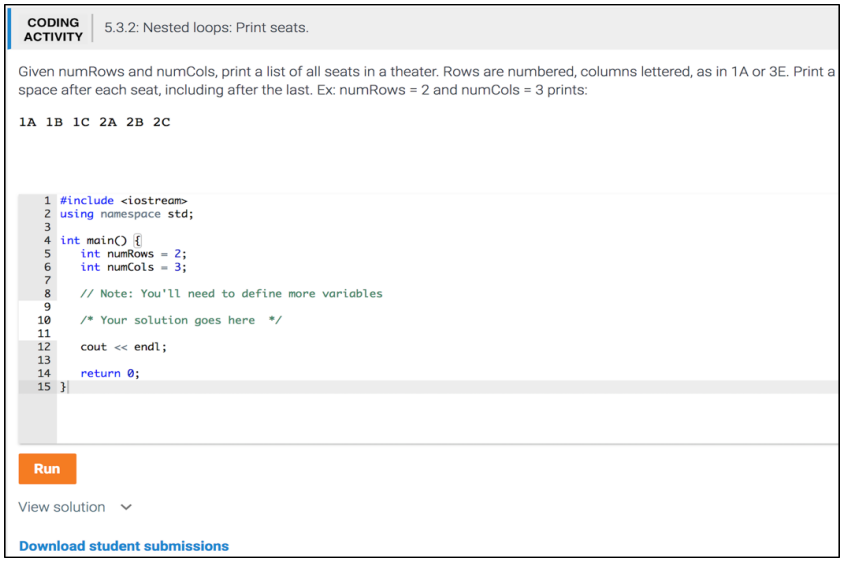

Figure 1: Coding activity _3_2_Nested_loops_Print_seats, on which many students struggled due to the need for nested loops.

Struggle rate as a metric

We desired a metric that would highlight CAs that caused students to struggle. Informally, struggle means a student works on a problem inefficiently, using excessive time or repeated attempts, and causing frustration. Ideally, we would measure struggle directly by observing the student and/or asking the student, but such data is not available and hard to obtain. Instead, we need a way to measure the struggle from the data we have, which involves every submission (wrong or right) of every student, each submission's date/time, and the submission's correctness.

One struggle measure is time spent. But, some activities require more time than others, so time alone is insufficient. Another measure is number of attempts. But, some activities may involve students detecting and correcting simple errors, converging quickly to a correct solution. Thus, we define a struggle metric that combines time and number of attempts. And, to avoid considering naturally long activities as having struggle, we consider the ratio of time and attempts compared to the "top" 20% of students for each CA. "Baseline time" is the average of the top 20% of student times for a CA. "Baseline attempts" is the average of the top 20% of the student attempts for a CA. Because CA's are designed to take 5-7 minutes, as a catch-all we also define 15 or more minutes as struggle.

Time of submission	User #	Answer correct	Submitted solution

5/7/2017 11:50:03 PM	0xxxx	No	for (i=0; i <= numRows; ++i) {
			for (j=0; j<= numCols; ++j) {
			return static_cast<char>('A' - 1 + i);
			cout << i << j;}}

...	0xxxx	No	...

5/7/2017 11:52:54 PM	0xxxx	No	for (i=0; i <= numRows; ++i) {
			for (j=0; j<= numCols; ++j) {
			cout << i << j << " ";
			}}

...	0xxxx	No	...

5/8/2017 12:01:37 AM	0xxxx	Yes	for (i=1; i <= numRows; ++i) {
			for (j=0; j< numCols; ++j) {
			cout << i << static_cast<char>('A' + j) << " ";
			}}

Table 1: A snapshot of rows of one particular student's submission for the CA 5_3_2_Nested_loops_Print_seats.

Figure 2 illustrates our struggle metric calculation. For a CA, we consider each student individually. Student1 has n submissions, each with a time, and the n^{th} being correct. (We ignored any submissions from a student following a correct submission, as that student was

likely just experimenting). For Student1, we compute the total time for that student as the n^{th} submission's time minus the first submission's time. Note that this time may be an underestimate, as the time doesn't include the time the student spent reading the instructions and developing the first submission. If two successive submissions are separated by at least 10 minutes, we assume the student was perhaps taking a break (this is not a perfect measure but the best we can do as we cannot directly observe the student), and thus we exclude that time from the total time. For every student (two are shown in Figure 2), such total time is computed. We then compute the average of the shortest 20% of such times to yield the baseline time. The same approach is done for the number of attempts per student.

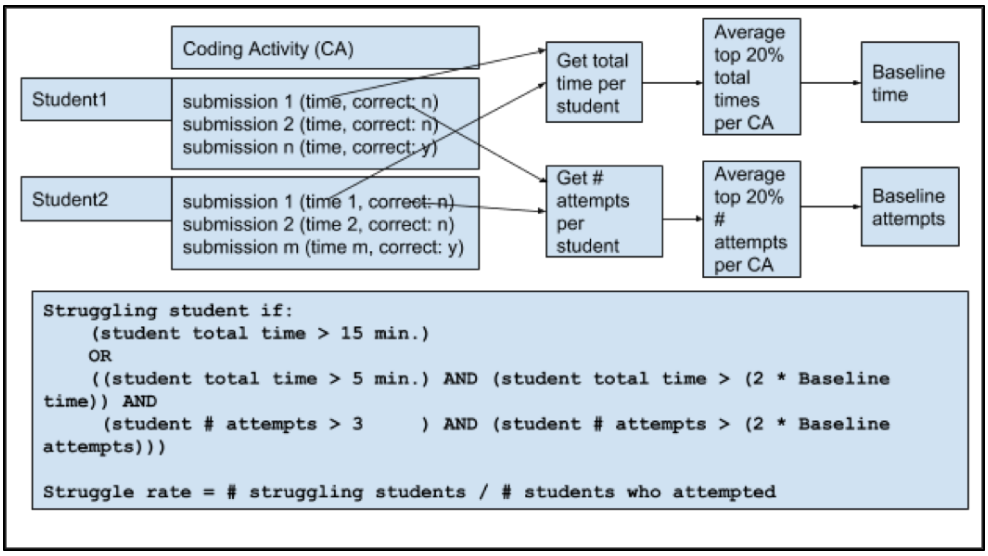

Figure 2: Definition of struggle rate for a particular CA.

Figure 2's bottom part shows how we define that a particular student struggled on a particular CA. The key features are that the student spent more than 2x the baseline time and more than 2x the baseline attempts. Furthermore, to account for very easy CAs, we also require the time be more than 5 min. And to account for the fact that a few errors on any CA are normal, we require more than 3 attempts. If all four of the above are true, we classify the student as having struggled on that CA. Also, any student spending more than 15 minutes is classified as struggling, since each CA was designed to take about 5-7 minutes. Given that definition of a struggling student, the struggle rate for a particular CA is then just the number of struggling students for that CA divided by the total number of students who attempted that CA.

Struggle rates for the CA's

For reference, Figure 3 shows average time and average number of attempts, along with struggle rate, for CAs sorted by struggle rate. While time or number of attempts obviously correlates with struggle rate, the struggle rate metric provides a more pronounced measure. Time alone, or

number of attempts alone, may fluctuate due to the inherent complexity of the particular CA, not necessarily indicating student struggle. As we see in Figure 3, some CAs have low number of attempts and spent time but has high struggle rate. For example, CA 3_3_2_If-else_statement_Fix_errors (which is represented by a blue and yellow dot on the x-axis coding activity 50 value), has low average attempts (less than 3 times) and low average spent time (less than 3 minutes), but has high struggle rate (above 10%). Figure 4 shows struggle rates for all CA's. The last bar is the average struggle rate: 12.3%.

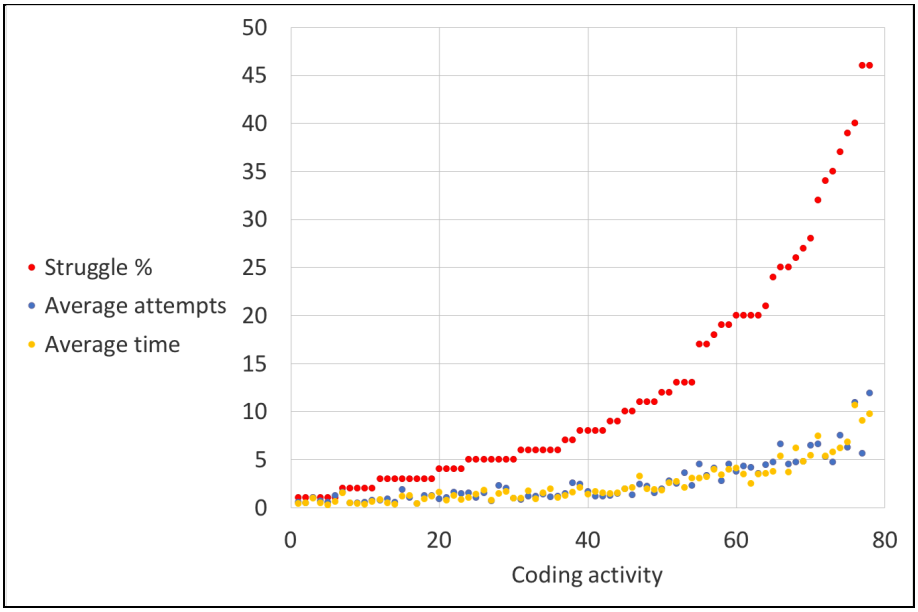

Figure 3: Coding activities versus average attempts, average time, and struggle rate (ordered by struggle rate).

Student errors for challenge activities with the highest struggle rates

Our goal was to understand what errors seemed to cause students to struggle, so that teachers or authors can devise appropriate means for students to avoid or fix those errors and thus reduce struggle (of course while balancing providing help with letting students figure things out themselves).

Table 2 highlights the 10 CAs with the highest struggle rates, all being above 25%. For each CA, the table presents the struggle rate, the CA name, the CAs sample correct code, one sample of wrong code from a student, and our conclusion of what errors yielded such struggle based on our manual code analysis of student submissions.

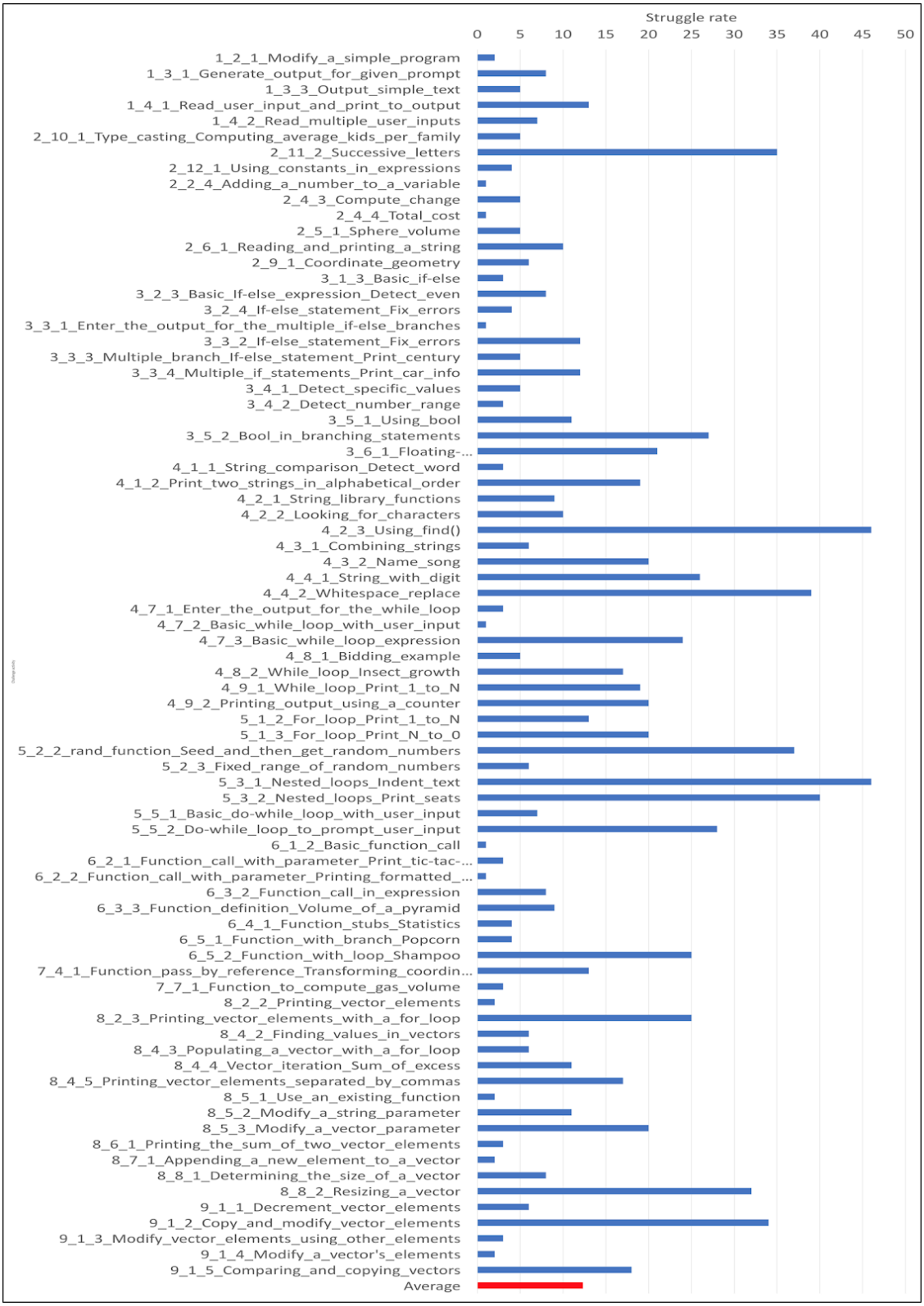

Figure 4: Struggle rates for the 78 CAs, in the order the CAs appeared in the 9 chapters of course content

%	CA name	Sample correct code	Sample wrong code	Common errors yielding struggle

46%	4_2_3_Using_find()	if (userInput.find("darn") != string::npos) {
		  cout << "Censored" << endl;}
		else {
		  cout << userInput << endl;}
			if (userInput.find("darn")) {
			cout << "Censored" << endl;
			}
			else {
			  cout << userInput << endl;}
				• Missing if statement condition to check if the word is found or not

46%	5_3_1_Nested_loops_Indent_text	for (i = 0; i <= userNum; i++) {
		  for (j = 0; j < i; j++) {
		  cout << " ";}
		  cout << i << endl;}
			for(i = 0; i <= userNum; i++) {
			  for(j = 0; j <= userNum; j++) {
			cout << j << endl;}
			  cout << " "; }
				• Wrong inner loop condition
				• Wrong cout() locations and arguments

40%	5_3_2_Nested_loops_Print_seats	char currColLet = 'A';
		for (currRow = 1; currRow <= numRows; currRow = currRow + 1) {
		  currColLet = 'A';
		  for (currCol = 1; currCol <= numCols; currCol = currCol + 1) {
		  cout << currRow << currColLet << " ";
		  currColLet = currColLet + 1; }}
			char letter = 'A';
			
			for (i=1;i<=numRows;++i) {
			  cout << i ;
			  for (j=0;j<=i;++j){
			  cout << letter +1 << endl;
			  }
			}
				• Missing initialization of "letter" in the outer loop.
				• Wrong initialization of "j" in the inner loop
				• Wrong condition for inner loop
				• Missing "letter" increment

37%	4_4_2_Whitespace_replace	if (isspace(passCode.at(0))) {
		  passCode.at(0) = '_';
		}
		if (isspace(passCode.at(1))) {
		  passCode.at(1) = '_';
		}
			if (isspace(passCode.at(0))) {
			passCode.replace(0,1,"_");
			  }
			  else if (isspace(passCode.at(1))) {
			  passCode.replace(1,1,"_");}
				• Using else if instead of multiple if

37%	5_2_2_rand_function_Seed_and_then_get_random_numbers	srand(seedVal);
		cout << (rand() % 10) << endl;
		cout << (rand() % 10) << endl;
			srand(time(0));
			cout << (rand() % 9) << endl; cout << (rand() % 9) << endl;
				• Using seed of time(0) (taught earlier) rather than obeying the instruction to use seedVal
				• Using %9 instead of %10

34%	2_11_2_Successive_letters	char letterStart;
		
		cin >> letterStart;
		cout << letterStart;
		letterStart = letterStart + 1;
		cout << letterStart << endl;
			char letterStartA = 'a';
			  char letterStartB = 'b';
			
			  letterStartB = letterStartA + 1;
			  cout << letterStartA << letterStartB << endl;
				• Missing cin()
				• Missing cout() after cin()
				• Missing incrementing the only variable
				• Missing printing that variable

34%	9_1_2_Copy_and_modify_vector_elements	for (i = 0; i < SCORES_SIZE; ++i) {
		if (i != (SCORES_SIZE-1)) {
		newScores.at(i)= oldScores.at(i+1);
		}
		else{
		newScores.at(i) =oldScores.at(0);}}
			for (i = 0; i < SCORES_SIZE - 1; i++) {
			newScores.at(0) = oldScores.at(oldScores.size() - 1);
			newScores.at(i) =oldScores.at(i - 1);
			}
				• Wrong for-loop condition
				• Missing if statement to update a variable in a loop
				• Wrong use of vector index to update variable

32%	8_8_2_Resizing_a_vector	countDown.resize(newSize);
		
		for (i = 0; i < newSize; ++i) {
		countDown.at(i) = newSize - i;
		}
			for(i = 0; i < SCORES_SIZE - 1; i++){
			if(i == 0){
			newScores.at(i) = oldScores.at(oldScores.size() - 1);}
			else{
			newScores.at(i) = oldScores.at(i - 1);} }
				• Missing resize of the vector
				• Wrong condition for the for-loop
				• Wrong code to change the vector elements

28%	5_5_2_Do-while_loop_to_prompt_user_input	do { cout << "Enter a number (<100): \n"; cin >> userInput; } while (!(userInput < 100));	cin >> userInput; do { cout << "Enter a number (<100): " << endl; cin >> userInput; } while (userInput < 100);	• Wrong condition

27%	3_5_2_Bool_in_branching_statements	if (isBalloon && !isRed) {
		cout << "Balloon" << endl;
		}
		else if (isBalloon && isRed) {
		cout << "Red balloon" << endl;
		}
		else {
		cout << "Not a balloon" << endl;}
			if ((isBalloon != false) && isRed) {
			cout << "Balloon" << endl;
			}
			if ((isBalloon = true) && (isRed = true)) {
			cout << "Red balloon" << endl;
			}
			else {cout << "Not a balloon"<< endl; }
				• Incorrect use of if statement
				• Incorrect condition for the if statement

Table 2: The CAs with the highest struggle rates, and our analysis of the errors leading to the struggle.

We considered errors as falling into two categories. “Core errors” are errors that likely would occur for a wide variety of courses and programming languages. “Specialized errors” are errors that seem rather specific to a particular function or language feature.

Some common core errors were:

- Nested loops: Students had trouble defining the inner loop’s initialization and expression, and knowing how to produce output within such loops.

- Else-if vs. multiple if: Students used else-if in cases where multiple if statements were needed, and vice-versa.
- Random range: Students often used % N rather than % (N+1) to generate random numbers in the range 0 to N.
- Input/output: Students early on had trouble putting output and input statements in the correct order.
- For loop and vector: Students had trouble creating a for loop header when the iteration wasn't through every element.
- For loop and if: Students had difficulty using an if statement inside a for loop when iterating through all elements.
- Vector index: Students had trouble when vector element updates weren't simple assignments while iterating through a vector, using the loop's counter variable as the vector index.
- Negated loop expression: Students often negated a loop expression, using while (x) rather than while (!x).
- Boolean expressions: Students struggled writing simple expressions with Booleans, comparing with false or true unnecessarily, increasing complexity and thus mistakes.

Some common specialized errors were:

- C++ find() function: Students did not understand C++'s unusual return value of the find() function, namely std::npos.
- Character increment: Students had trouble understanding that incrementing a character variable holding a letter yields the next letter in the alphabet.
- Random seeding: Students often improperly seeding a random function, by copying a particular example in our content that seeded with current time, rather than following instructions to seed with a particular variable's value.

Reducing struggle

The above information can be used in many ways, such as spending more time in lecture going over examples focused on avoiding a common error, creating more content for teaching a particular subject matter, improving code auto-graders to detect common errors and provide specific hints (perhaps after the student has spent at least a few minutes trying on their own), etc. The purpose of this paper is to summarize the common errors so that others can adjust their teaching/content accordingly.

However, we happened to have some historical data that was relevant to this work, and thus we chose to include that data. In particular, for 10 CAs that we'd noticed in the past that students asked the most questions on (so not based on a struggle metric, which we had not developed at that time), we in January 2017 added a hint link that discussed common errors. For those CAs,

we obtained submission data from before the hints were added, and after the hints were added. In fact, we obtained the data from our university which used the same CAs. The analysis compared Spring 2016 (262 students) and Spring 2017 (175 students).

Figure 5 shows the effectiveness of adding hints to some CAs (CA1 to CA8). The blue and orange bars represent the struggle rate for a CA before and after adding a hint, respectively. The hint drops the struggle rate modestly for most CAs, with a substantial drop for some. On average, the struggle rate for those CAs dropped from 23% to 19%, representing a 17% decrease in the number of struggling students. Note that some CAs had low struggle rates even before the hints were added, indicating that students asking questions about a problem does not necessarily mean that students are struggling on those problems.

As a reminder, adding a hints link is just one way, and a fairly modest way, to strive to reduce struggle rates. We suggest that more aggressive techniques be utilized.

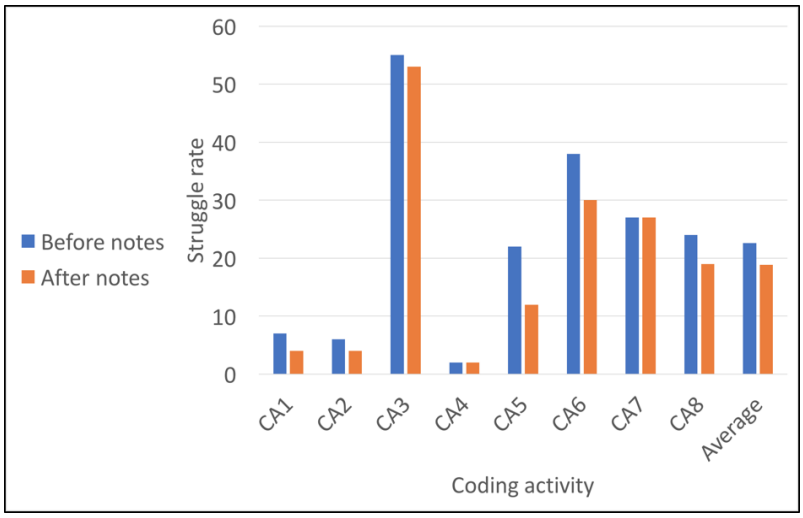

Figure 5: CAs versus struggle rate, without hints (blue) and with hints (orange).

Conclusion

We measured student struggle rates on homework problems (coding activities / CAs) in an introductory CS 1 programming course. We found struggle rates to be 10-15%, and as high as 30-40% for some CAs. We listed common errors that led to struggle, such as errors related to nested loops, use of else-if rather than multiple ifs, etc. We analyzed historical data showing that adding hints to particular CAs could reduce struggle rate. We hope that our publishing these common errors on homework problems of a CS 1 class will aid teachers and publishers to devise aggressive approaches to help students avoid or fix such errors. To be clear, we believe some struggle is necessary to learn programming; our goal is to address errors that lead to excessive struggle. Hints are one way to reduce struggle but others ways exist. Also, we do not believe that automatically providing hints for every student error, as done in some auto-grading homework

systems, is the best approach, as that approach may lead to excessive dependency and may hamper critical thought or effort. Our goal instead is to focus in on the specific errors that yield excessive struggle. In future work, we will build debugging activities focused on reducing the errors found in this paper. We also hope to automatically detect these common errors and to provide custom hints. Both approaches we hope will reduce student struggle.

References

[1] Mahmoud, Q. H.; Dobosiewicz, W. & Swayne, D. Making computer programming fun and accessible Computer, IEEE, 2004, 37, 106-108.
[2] Beaubouef, T. & Mason, J. Why the high attrition rate for computer science students: some thoughts and observations, ACM SIGCSE Bulletin, ACM, 2005, 37, 103-106.
[3] Spohrer, J. C.; Soloway, E. & Pope, E. A goal/plan analysis of buggy Pascal programs, Human--Computer Interaction, Taylor & Francis, 1985, 1, 163-207.
[4] Yarmish, G. & Kopec, D. Revisiting novice programmer errors, ACM SIGCSE Bulletin, ACM, 2007, 39, 131-137.
[5] Horstmann, C. & Budd, T. Big C++, Wiley, 2004.
[6] Oualline, S. How not to program in C++: 111 broken programs and 3 working ones, or why does 2+ 2. No Starch Press, 2003.
[7] Denny, P.; Luxton-Reilly, A. & Tempero, E. All syntax errors are not equal, Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, 2012, 75-80.
[8] Becker, B. A. An effective approach to enhancing compiler error messages, Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 2016, 126-131.
[9] Altadmri, A. & Brown, N. C. 37 million compilations: Investigating novice programming mistakes in large-scale student data, Proceedings of the 46th ACM Technical Symposium on Computer Science Education, 2015, 522-527.
[10] Ginat, D. & Shmalo, R. Constructive use of errors in teaching CS1, Proceeding of the 44th ACM technical symposium on Computer science education, 2013, 353-358.
[11] Alqadi, B. S. & Maletic, J. I. An Empirical Study of Debugging Patterns Among Novices Programmers, Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, 2017, 15-20.
[12] Cherenkova, Y.; Zingaro, D. & Petersen, A. Identifying challenging CS1 concepts in a large problem dataset, Proceedings of the 45th ACM technical symposium on Computer science education, 2014, 695-700.
[13] Hertz, M. & Jump, M. Trace-based teaching in early programming courses, Proceeding of the 44th ACM technical symposium on Computer science education, 2013, 561-566.
[14] O'Dell, D. H. The debugging mind-set, Communications of the ACM, ACM, 2017, 60, 40-45.
[15] zyBooks. https://www.zybooks.com/, accessed Mar, 2018.