Learning the Interaction Between Pointers and Scope in C++

Amruth N. Kumar
Ramapo College of New Jersey
505, Ramapo Valley Road
Mahwah, NJ, USA, 07430-1680
(201) 684 7712
amruth@ramapo.edu

ABSTRACT
Traditionally, pointers, and their interaction with scope in C++ have been a source of frustration and confusion for students in our Computer Science II course. Since problem-solving is known to improve learning [6], we set out to develop software that would help our students better understand these concepts by repeatedly solving problems based on them.

In this paper, we will first describe the design and features of this software. We conducted tests in two sections of our Computer Science II course this fall to evaluate the effectiveness of using this software. The results have been very encouraging: the class average in both the sections increased by 100% from the pretest to the post-test. We will also present the design and results of these tests.

1. INTRODUCTION
Pointers have traditionally been a source of frustration and confusion for students. Some of the problems students face when using pointers in C++ include:

• Pointers are not automatically set to null in ANSI C++, unless they are global variables. Students who do not develop the discipline of initializing pointers to null during declaration may attempt to dereference them before assignment. This could lead to run-time or semantic errors.

• If a pointer is assigned to point to a variable, and the variable goes out of scope, the pointer becomes a dangling pointer. Once again, dereferencing it can lead to run-time or semantic errors.

The scope of variables in a C++ program with nested blocks is also a complex concept for novice programmers:

• Students often have problems with the fact that a variable does not exist outside the block in which it is declared, and may not be assigned or referenced after the closing brace of the block. Often, even the students who understand the concept of the scope of functions find the scope of blocks hard to comprehend since blocks have no names with which they can be identified.

When a program includes both nested blocks and pointer variables, the code may suffer either syntax errors (variables are assigned or referenced out of their scope) or dangling pointers (pointers are dereferenced before being assigned or pointers are dereferenced after the variable to which they point goes out of scope).

We had observed that our students had problems with both these concepts. Therefore, we set out to develop software that would help our students better learn these concepts through problem-solving.

In this paper, we will first describe the design and implementation of our software to generate problems on the interaction between pointers and scope. We are currently using this software in our Computer Science II course where students learn about working with pointers.

We conducted tests in two sections of our Computer Science II course this fall to evaluate the effectiveness of using this software. The results have been very encouraging: the class average in both the sections increased by 100% from the pretest to the post-test. We will discuss the design of our tests and their results later in this paper. Finally, we will discuss how our work relates to the work of others in this area, and list some items for future work.

2. THE DESIGN OF THE SOFTWARE
The objective of the software is to repeatedly generate problems involving blocks and pointers, and either solve the problems or check the user’s answers to the problems.

The problems the software generates are C++ function definitions, which include pointers and possibly nested blocks. The user is asked to indicate whether a function has any syntax errors or dangling pointers, and if neither, whether the code works correctly. After grading the user’s answer, the software provides detailed feedback to the user.

2.1 Problem Generation
The software randomly generates the problems, rather than serving them from a canned repository of problems. It generates the problems as parameterized instances of problem templates. Currently, the software knows 21 problem templates: [[PVA]D], [P[V]AD], [P[V]A]D], [P[VAD]], [PVAD], [V[PA]D], [V[P]AD], [V[PD]A], [V[PD]A], [VPAD], [V[PA]D], [P[V]AD], [PVAD], [PVAD], [P[VDA]], [P[VD]A], [P[V]A], [P[VD]A], [[PD]VA], [PD[VA]], [PD[VA]], [PD][VA], [PD][VA]. In these templates, P is a pointer declaration statement, V is a variable declaration statement, A is a pointer assignment statement, and D is a statement dereferencing the pointer. Each pair of brackets refers to a block, the outermost pair referring to the function itself.

The software randomly selects one of these templates and generates a function based on the template. It randomly selects all
the following: a name for the function, the names of variables and pointers, and the data type of the variables and pointers. Therefore, the software is capable of generating a combinatorially explosive number of un-identical problems based on the 21 templates, and a user may rarely, if ever see the same problem twice. Therefore, a user may use the software to practice solving problems as long as (s)he pleases.

Note that in the templates, the pointer is always declared before it is assigned or dereferenced, and the variable is always declared before it is used in an assignment statement. Therefore, the software does not generate egregiously incorrect code such as assigning to a pointer that has not yet been declared. The software also randomly includes distracters in the code to make the problems more interesting.

2.2 User Interface

The transition diagram for user interaction with the software is shown in Figure 1. The answering options are provided as check boxes, one for each of “Syntax Error”, “Dangling Pointer” and “Code OK” – the user may select one or more of these boxes before clicking on the “Check My Answer” button. In response to this button, the software displays feedback, which includes the correct answer, whether the user’s answer is correct, and an explanation for the correct answer. The user may now choose to generate a fresh problem by clicking on the “Generate Problem” button, and repeat the above problem-solving sequence. The software makes the “Check My Answer” and “Generate Problem” buttons available only in their correct contexts to avoid confusion for the user. Please see Figure 2 for a snapshot of the user interface.

The feedback provided by the software is quite detailed. In the case of a dangling pointer, it explains why the pointer is dangling, e.g.,: that the pointer was declared and assigned properly, but the variable to which the pointer is pointing is out of its scope (See Figure 2). Similarly, in the case of a syntax error, it explains why the code has a syntax error, e.g., the pointer that is being dereferenced is out of its scope. Our objective in designing the feedback was to make it sufficiently explanatory so that students who did not know about the interaction between scope and pointers could learn by reading the feedback.

Finally, the software keeps track of usage statistics: the number of problems attempted by the user, and the number of problems the user solved correctly, incorrectly, or partially.

The software is implemented as a Java applet for obvious reasons – applets can be made available over the Web, and can be accessed from anywhere any time. It uses Swing classes, consists of 8 classes, and is about 75K in size.

3. EFFECTIVENESS OF USING THE SOFTWARE

We wanted to test our hypothesis that the software would be helpful to students. Therefore, we conducted tests in two sections of our Computer Science II course this fall to evaluate the effectiveness of using this software. In this section, we will first describe the protocol we used for the tests, and then discuss the results we obtained from the tests. The tests were conducted after we had already covered the concepts of dangling pointers and scope in our lectures.

The protocol for the tests was as follows:

PreTest: We first asked our students to answer a written pretest, which consisted of 20 questions on the interaction of pointers and scope. The students were allowed seven and a half minutes to answer the pretest. We had chosen the number of problems and the allowed time to avoid any ceiling effect in our observations. We urged the students to answer as many questions as they could.

Practice: Next, we asked our students to practice problem-solving with the software. Each student was seated at a separate computer, with the software already launched in a Web browser. After briefly describing how to use the software, we asked the students to solve as many problems as they could in the next seven and a half minutes. We asked them to pay special attention to the feedback provided by the software for each question.

Post-Test: Next, we asked the students to answer a written post-test consisting of 20 questions. We urged the students to answer as many questions as they could in the next seven and a half minutes. We asked the students to pay special attention to the feedback provided by the software for each question.

Post-Test: Next, we asked the students to answer a written post-test consisting of 20 questions. We urged the students to answer as many questions as they could in the next seven and a half minutes. At the end of the allotted time, we asked the students to record their impressions of the software on the back of the post-test book.

Finally, we visited each student workstation and recorded the usage statistics posted by the applet: the number of problems the student had attempted, and the number of problems the student
had solved correctly, incorrectly or partially during the practice session.

Note that we had generated the pretest and the post-test questions themselves by running the software. We had formatted and printed these questions on paper, and handed them out to students. When grading the tests, we awarded 2 points for each correct answer and 1 point for each partially correct answer. Partially correct answers were those where the user left out half the answer or included an option that was not part of the answer (i.e., negative grading). The maximum score on the pretest and post-test was 40.

**Observations:** Table 1 lists the scores for the 10 students in the Monday–Thursday section of our Computer Science II class. Note that the class average score increased from 11.4 in the pretest to 26.7 in the post-test, an improvement of over 100%! Therefore, using the software did help students learn problem solving through practice.

Table 1: Scores for CS 202 (Monday/Thursday) class

<table>
<thead>
<tr>
<th>202</th>
<th>Pre-Test Score</th>
<th>Practice Score</th>
<th>Post-Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>29.0</td>
<td>40.0</td>
<td>40.0</td>
</tr>
<tr>
<td>Average</td>
<td>11.4</td>
<td>20.1</td>
<td>26.7</td>
</tr>
<tr>
<td>Std. Dev</td>
<td>7.9</td>
<td>11.5</td>
<td>10.3</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.0</td>
<td>9.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>

Table 2 lists the number of problems attempted by the students in this class. Note that there was a 50% improvement in the average number of problems attempted by the students from the pretest to the post-test. Remarkably, the minimum number of problems attempted by any student on the post-test was close to the average number of problems the class had attempted on the pretest. This clearly indicates that students had gained confidence on the topic by using the software for practice.

Table 2: Number of problems attempted by CS 202 class

<table>
<thead>
<tr>
<th>202</th>
<th>PreTest Problems</th>
<th>Practice Problems</th>
<th>Post-Test Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>20.0</td>
<td>22.0</td>
<td>20.0</td>
</tr>
<tr>
<td>Average</td>
<td>12.7</td>
<td>14.2</td>
<td>18.7</td>
</tr>
<tr>
<td>Std. Dev</td>
<td>5.1</td>
<td>4.8</td>
<td>3.0</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.0</td>
<td>8.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>

Similar trends may be observed in the data for the 9 students in our Wednesday night class shown in Table 3.

The impressions that students recorded on the post-test book gave us insight into what the students thought about the software. The student who scored 0 on the pre-test (the minimum score on pretest in Table 1) ended up scoring 34 on the post-test. Much to our chagrin as the course instructor, it was as though he had learned everything he knew about pointers and scope from using the software. He wrote: “I think this is a very productive way in helping your students’ understand pointers. As you can see from my pretest compared to my post-test at least I thought I knew what I was talking about in my post-test. In the pretest, I did not have any idea. I definitely think you should do this with future classes.” All the student comments were similarly positive. Some of the other illustrative comments include: “Software was very helpful in identifying the problems as I went along. I’d use it more often”,”…”practice on several examples is beneficial so that we can see all the details”, “thought was helpful in teaching what I was not totally sure about”, “Program was easy to use, and gave useful info after your answer”, “Program definitely helped me a lot. It clarified scope for me”, “The software was easy to use due to its simplicity. It works on the principle of ‘practice makes perfect’”, “The software seem to help because I seemed to have better understanding of dangling pointers on the post-test”, “I believe the software was helpful in understanding dangling pointers”, “I would like to use it more often to understand concepts more”, “I feel that I improved with going through the tutorial”, “I feel that the program was helpful. I feel this way because I got feedback on the choice that I made. This helps to explain why I was correct or why I was wrong”.

Table 3: Scores for CS 201 (Wednesday night) class

<table>
<thead>
<tr>
<th>201</th>
<th>Pre-Test Score</th>
<th>Practice Score</th>
<th>Post-Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>21.0</td>
<td>46.0</td>
<td>38.0</td>
</tr>
<tr>
<td>Average</td>
<td>13.1</td>
<td>19.9</td>
<td>26.8</td>
</tr>
<tr>
<td>Std. Dev</td>
<td>5.4</td>
<td>13.0</td>
<td>7.1</td>
</tr>
<tr>
<td>Minimum</td>
<td>8.0</td>
<td>5.0</td>
<td>18.0</td>
</tr>
</tbody>
</table>

### 4. COMPARISON WITH RELATED WORK

Problem-based learning improves long-term retention [6], and is better than traditional instruction for improving the ability of students to solve real-life problems. Textbooks are generally inadequate as sources of problems because of their limited, non-interactive nature.

Moreover, textbooks do not or cannot present the steps required to solve a problem, the visualizations that are helpful in solving the problem, or the feedback that helps a student learn from his/her mistakes. Even in disciplines such as Physics and Mathematics, where textbooks generally tend to include many more practice problems than in Computer Science, faculty are increasingly turning to the use of technology to address this issue. Typically, programs are being written to generate problems, and such programs are being made available to students for practice, e.g., CAPA [7] for physics, and CHARLIE [3] for electronics and control systems.

Not many such problem generation systems have been attempted for Computer Science topics. We found numerous systems that solve problems entered by the student (e.g., [10]), or administer problems generated by the instructor (e.g., [1]), but only a few systems that generate and solve problems for the user namely, PILOT [4], SAIL [5] and Gateway labs [2]. PILOT is a problem generation tool for graph algorithms, SAIL is a LaTeX-based scripting tool for problem generation, and Gateway Labs generate problems on mathematical foundations of Computer Science. That there are very few problem-generators for Computer Science may be due to the fact that problems in Computer Science are often
based on arbitrary programs and their answers are not quantitative. This is true of the topic we picked for our software, viz., the interaction between pointers and scope.

In Physics, the use of problem generation systems has been shown to increase student performance by 10% [8], largely due to increased time spent on the task. Our evaluations seem to support this result, reinforcing that problem generation software does have a role to play in higher education. We plan to use the software as a supplement (and not a substitute) to large scale programming projects so that our students can learn the basic concepts by solving problems with the software, and later synthesize these concepts to write their programming assignments.

Problem-generation software may also be used to facilitate active learning in distance education courses, and individualized testing in traditional courses.

We have developed or are in the process of developing problets for many other topics in Computer Science, including expression evaluation in C++ [9] and nested selection statements in C++ [11]. We plan to extend our work on the pointer problet into a general framework for generating problems based on code.

5. FUTURE WORK
We plan to extend our work in many ways in the future:
- We plan to incorporate dynamic memory management concepts into the software.
- We plan to conduct further tests to assess the effectiveness of using the software.
- We plan to add templates with multiple functions and global variables to the software.

The software is currently available over the Web at http://orion.ramapo.edu/~amruth/problets

6. ACKNOWLEDGEMENTS
Partial support for this work was provided by the National Science Foundation’s Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864.

7. REFERENCES