Dereferee: Exploring Pointer Mismanagement in Student Code

Anthony Allevato, Stephen H. Edwards, and Manuel A. Pérez-Quiñones
Department of Computer Science
Virginia Polytechnic Institute and State University
2050 Torgersen Hall, MC 0106
Blacksburg, VA 24061
allevato@vt.edu, {edwards, perez}@cs.vt.edu

ABSTRACT
Dynamic memory management and the use of pointers are critical topics in teaching the C++ language. They are also some of the most difficult for students to grasp properly. The responsibility of ensuring that students understand these concepts does not end with the instructor's lectures—a library enhanced with diagnostics beyond those provided by the language's run-time system itself is a useful tool for giving students more detailed information when their code fails.

We have designed such a toolkit, Dereferee, which students can incorporate into their code with minimal intrusion into the learning process. To demonstrate its effectiveness, we examine C++ code from students in a course that relied solely on the built-in memory management behavior of the language, without any significant additional diagnostic or debugging facilities. We instrument this code with Dereferee in order to explore the causes of errors that result in program crashes and to expose hidden faults that previously lay undetected. Dereferee provided enhanced diagnostics for bugs in 63% of student submissions, and pinpointed the source of 83% of abnormal program terminations. 95% of the students would have received extra diagnostic help from using Dereferee.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging—testing tools; D.3.2 [**Programming Languages**]: Language Classifications—C++.

General Terms
Languages, Verification.

Keywords
Dynamic memory management, test-driven development, TDD, test-first coding, smart pointer, pointer checking, dangling pointer, memory leak, null dereference, programming assignment.

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

1. INTRODUCTION
Many computer science programs include C++ as a core part of their curricula. In our courses, we have consistently seen that memory management techniques in C++ are one of the most, if not *the* most, frequent sources of difficulty for our students. This observation is consistent with the experiences of others [5][6].

Unfortunately, the tools typically used to teach C++, such as Microsoft Visual C++ or the GNU gcc compiler, are professional tools that do not provide the most appropriate feedback for educational purposes, especially for students at an introductory level. We have also observed that the complexity of the debugging tools available in those environments acts as a barrier to student use; instead, they prefer "caveman debugging" such as inserting output statements to trace execution. A better strategy might be to "push" diagnostic information to the student, rather than requiring them to actively "pull" these data.

We have created Dereferee to address these issues. By using this toolkit, students receive detailed diagnostics when they make memory-related errors in their code—errors that normally would cause a program to crash, or worse, would silently lead to undefined or incorrect behavior. The toolkit detects and diagnoses 40 unique kinds of pointer mismanagement errors ranging from dereferencing null, uninitialized, or dangling pointers, to more advanced problems such as faulty pointer arithmetic or accessing arrays out of bounds. In addition, memory leaks are detected at the precise moment that access to a live block of memory is lost.

In order to illustrate the real benefits of using Dereferee for teaching C++ memory management techniques, we have collected the source code submitted by students in an introductory C++ course that made use of an automated grading system, but only for checking correctness of output without offering any debugging assistance. We then modified the code to use Dereferee and analyzed the extent to which additional bugs are discovered and diagnosed, as well as the frequency of different types of errors that students produce. This analysis indicates that Dereferee provided enhanced diagnostics in 63% of student submissions. It pinpointed the causes of 83% of abnormal program terminations by **identifying all dynamic memory management errors**. When used in conjunction with CxxTest [2] to provide crash recovery, a statistically significant increase in test pass rates was achieved while formerly hidden pointer errors were explicitly revealed and diagnosed. Indeed, 95% of the

students in the study would have received additional diagnostics on their pointer mismanagement at some point in the course.

2. BACKGROUND

Dereferee was inspired by Checkmate [7], a library that provides a checked pointer template for C++. Like Checkmate, we maintain a memory allocation table that keeps track of reference counts for each block of memory allocated, but we also add other useful properties such as the C++ type name of the object(s) allocated, whether each block holds a single object or an array, the length of dynamically allocated arrays, and a stack trace identifying the location and context of the allocation. Each pointer is viewed as an ad hoc "state machine" that transitions between live, null, dead, and out-of-scope states as a result of the operations performed on it during the execution of a program [7].

While the designers of Checkmate restrict their users to a small, basic set of operations, our toolkit was created with the goals of achieving maximum transparency (minimal intrusion into the user's development process) and completeness (support for as many pointer operations as possible). We also wished to address shortcomings in Checkmate's design and improve its syntax and usage. Notably, Checkmate does not support any array operations, and in order to capture information about the locations of allocation and deallocation actions, it uses its own non-standard macro-based syntax instead of overloading the global *new* and *delete* operators. That approach does not support the full expressiveness of *new* expressions in C++, such as the ability to invoke parameterized constructors or to directly allocate a derived class object and assign it to a base class pointer (in Checkmate, an intermediary pointer of the derived type must be used).

Furthermore, we feel that using a custom syntax is an unhelpful departure from the standard C++ usage that we wish to teach, particularly for students who are seeing the language for the first time and will eventually be required to apply the skills that they learn without the use of a toolkit such as Checkmate or Dereferee.

3. DEREFEREE USAGE

In contrast, using Dereferee only requires one to include a header file, link to the library, and (like Checkmate) modify pointer type declarations. The source code modification involves replacing heap-allocated pointer declarations (variables, fields, and function arguments) of the form `T*` with the construct `checked(T*)`, a macro call that resolves to `checked_ptr<T*>` after preprocessing. The `checked()` macro serves two purposes. First, it conceals the fact that the underlying declaration is actually a template in order to prevent the need for introducing the concept of templates earlier than an instructor may wish. Second and more importantly, it provides a way to "compile out" the checked pointers entirely from a release build, making the toolkit appealing even in performance-sensitive code. We have attempted to design Dereferee so that it is strong enough to be usable in non-academic production environments as well as in the classroom.

This transparency makes it easy for users to adopt Dereferee early in their development process and also simplifies the process of retrofitting existing code, which greatly accelerated our experimental process. With a small Perl script, we could easily add `#include` statements into each source file and replace pointer declarations with their checked counterparts.

Once the student's code has been instrumented with these checked pointers, any errors or warnings will be reported with a detailed explanation of the fault that occurred, including a full stack trace that identifies the location and context of the error. Even non-fatal pointer errors are identified **immediately at the point where they occur**, rather than only when they affect externally observable behavior. These errors are reported to standard output by default, but Dereferee's modular design allows this behavior to be changed by linking in a different "listener" module.

It is important to note that Dereferee's pointers are **diagnostic only**, meaning that they provide no additional semantics beyond enhanced error reporting. This is in contrast to "smart pointers" such as the standard C++ `auto_ptr` or those provided by third-party libraries such as Boost [1], which take ownership of the memory they point to and assume the responsibility of releasing it automatically. In teaching our students how to manage dynamic memory, we do not want to absolve them of this responsibility; rather, we want to provide them with more helpful feedback about erroneous behaviors so that they can master proper pointer management skills. In many ways, the lack of information provided in the context of most pointer errors is a serious inhibitor to learning—if students do not know what they did wrong or where to look, they are frustrated rather than educated.

4. EXPERIMENTAL EVALUATION

The following experiments analyze the student-submitted code for three assignments in a sophomore-level data structures course that also serves as the first exposure to C++ for many of our students. These assignments—an ordered singly-linked list (OrderedSList), a doubly-linked list (DList), and a binary search tree (BST)—required the students to develop and apply a clear understanding of pointers and memory management concepts. Students worked individually, developed and tested their own solutions, and then submitted them to an automatic grading system for assessment. Each submission received feedback in the form of test results from instructor-written reference tests. Students were allowed a fixed number of submission attempts for each assignment: 15 each for OrderedSList and DList, and 10 for BST.

For each assignment in this experiment, we modified the instructor's original custom test harness to use a version of CxxTest [2], a C++ unit testing framework similar to JUnit that we have used in other C++ courses. CxxTest provides a uniform way to collect data on test results and, more importantly, provides crash recovery for test cases so that even when fatal errors such as segmentation faults occur, the test harness can recover and continue to execute any remaining test cases.

We first compiled and executed the students' submitted code and the instructor's tests under the same conditions originally used in the classroom in order to collect the results that correspond to the students' experiences. In this phase, each test case can generate a result of *success, test case failure* (failure of a test case assertion that checks behavioral correctness), *abnormal termination* (an untrapped signal or exception), or *unexecuted* (because an earlier test case abnormally terminated the test run). Two other possible results, *timeout* and *unbounded recursion*, are not of interest in this analysis, since they do not involve pointer mismanagement.

The source code for each submission was then modified to include Dereferee support, and then compiled and executed again

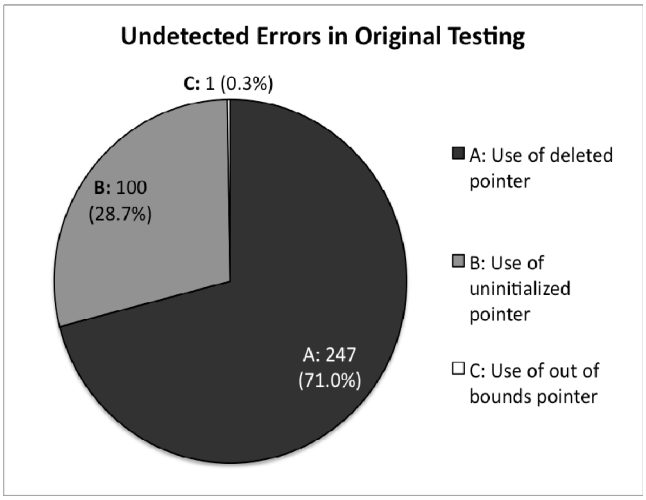

Figure 1. Frequency of errors that were undetected in original testing.

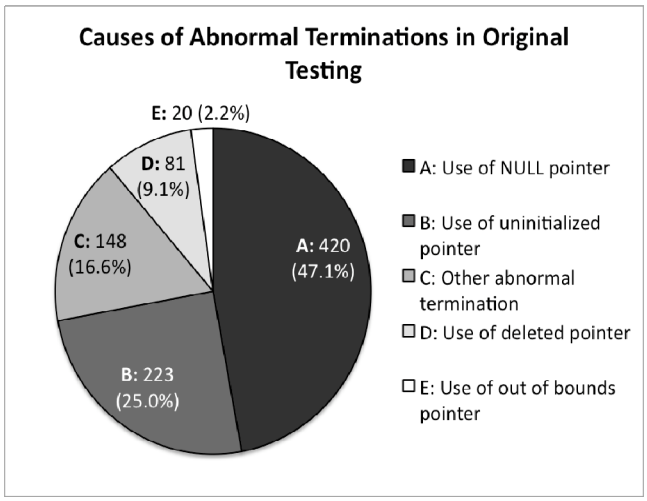

Figure 2. Actual reasons for errors that caused abnormal termination during original testing.

to obtain more detailed information about any errors that occurred. Test cases executed in this phase can result in any of the three outcomes described before—success, test case failure, or abnormal termination—or can produce a specific diagnostic message from Dereferee describing a pointer error that was detected. Because Dereferee prevents pointer errors from corrupting the program state or causing memory faults, and because CxxTest provides crash recovery for all other fatal errors, no test cases went unexecuted in this treatment condition.

There were 91 students enrolled in this course who made a combined total of 731 submission attempts across the three assignments. Over this entire submission set, we have collected results from 9,480 unique test case executions. Each of these results includes the reason for the success or failure of the test case under the original testing conditions as well as when using Dereferee. Of the 9,480 test cases, there were 7,659 successes, 594 test case failures, 390 abnormal terminations, and 617 unexecuted tests under the original grading conditions.

4.1 Finding Undetected Errors

For reasons of efficiency, C++ runtime systems typically do not rigidly police memory usage in an application beyond the most egregious offenses (such as dereferencing a null pointer or one that points to memory that does not belong to the application). For instance, a C++ implementation might use a *malloc/new* that allocates a large pool of memory from the operating system and then satisfies user requests by partitioning this pool. When the user deletes an object under this scheme, the memory is simply marked as free in the pool but is not returned to the operating system. This behavior can silently mask some errors, such as exceeding an array's bounds or dereferencing a pointer to memory that has been freed, resulting in undefined behavior that may or may not cause the application to fail at a later time.

We first looked at how many hidden errors went undetected in the original testing. Specifically, we considered test cases that resulted in *success* under the original testing conditions, but the result using Dereferee was one of the toolkit's specific error codes. Of the 720 submissions that passed at least one test case, 21.3% (153) experienced at least one "false positive" test case result. 48 students were responsible for making this set of submissions, so undetected pointer errors were written by 52.7% of students in the class at some point during the semester.

Figure 1 shows the reasons for these undetected errors. Students made two major kinds of errors that went undetected—at least not immediately: using a dangling pointer (a pointer to memory that has already been deleted), and using a pointer variable that has been declared but not yet initialized. The remaining class of errors is interesting because it indicates problems related to arrays and pointer arithmetic, neither of which were concepts that were used in any of the assignments being analyzed. They show up here as artifacts in the work of a student who mistakenly treated pointers themselves as the values to which they pointed, and in doing so performed arithmetic and relational operations on them.

Note that Dereferee provides more detailed explanations for the error categories in Figure 1. Thanks to the overloaded operators that the checked pointer template provides, we give the student specific information at the point of an error, such as "dereferenced a deleted pointer with the arrow operator," "…with the star operator," "compared an uninitialized pointer with another value," "assigned an uninitialized value to a pointer", and so forth. The clusters in Figure 1 emphasize the higher-level causes.

4.2 Better Diagnostics for Detected Errors

When a fatal memory-related error is detected by the C++ run-time, the result is a segmentation fault or unhandled system exception that provides no useful information about what caused the error or where the problem occurred. When this happens, the student must try to locate the source of the problem by examining their partial output up to the point of the crash to see which operations were being performed, and even then there may not be enough information to determine the cause.

Dereferee is helpful in these cases not only because it describes the reason for the error in greater detail, but also because it provides a stack trace that shows where the error occurred. Figure 2 shows a breakdown of the 9.4% of all test cases (892/9480) that failed due to abnormal termination under the original conditions. Of these, Dereferee provided a more detailed diagnosis for 83.4% (744) of the abnormal terminations. Null pointer dereferences are by far the most common of these errors, explaining almost half of such events. The use of uninitialized

pointers is the second most common, representing exactly one quarter of this type of error.

Why do uninitialized pointers sometimes cause a test to crash, and sometimes silently succeed as shown in the previous experiment? Since the value of an uninitialized variable is simply whatever leftover bits were previously stored in that slot, whether or not those bits represent a valid memory address is left to chance.

Dereferee tackles this problem in a unique way. When the memory manager is created at the start of execution, a "token" is allocated in memory for the lifetime of the process. The address of this token is used to represent "uninitialized" pointers. Any checked pointer variables that are not immediately assigned another value are set to point to this token. When a pointer is used in an expression, we can check for this unique value and report that the pointer is, in effect, uninitialized.

Finally, Figure 2 shows that *abnormal termination* remains undiagnosed by Dereferee in 16.6% of cases. This indicates errors that did not involve checked pointers, but rather arose from misuse of unchecked components such as STL containers, or from other untrapped exceptions and signals.

One other way that better diagnostics can be provided for detected errors arises from examining test case failures, which simply indicate the program produced incorrect results. 316 submissions resulted in at least one test case failure without the Dereferee. However, when using Dereferee, 19.9% of these actually resulted in specific pointer error diagnostics that pinpointed the location of the error. 34.2% (27) of the students in the course wrote at least one such submission during the course, and would have received improved feedback on what specifically they did wrong.

4.3 Information Lost In Original Testing

The crashes described in Section 4.2 pose another problem as well. Test cases that were executed under the original conditions are not completely independent because they run without any form of signal protection or fault recovery. If a severe bug manifests itself in an early test case, then the program will crash and none of the tests that follow will be executed. This unfairly penalizes the student, because any of those tests that would have passed are potential points that go unearned. Furthermore, even if those tests would not have passed, seeing the failures and diagnostics can be helpful in identifying additional bugs.

Figure 3 shows a scatter plot of all submissions, where the *x*-axis represents the percentage of tests that passed under original conditions, including the fact that when an abnormal termination occurred during execution, none of the tests that follow it would be executed. The *y*-axis represents the percentage of tests that passed when using Dereferee (which prevents pointer errors from causing program crashes) and CxxTest with crash recovery (which then handles crashes due to other causes). Most submissions fall above the identity line, indicating that merely by improving the automated testing environment, a significant number of students' scores would have improved—not just in the numerical sense, but also to better reflect the correctness of the work that was submitted. Some submissions occur below the identity line as well; these are reflective of hidden errors discussed in Section 4.1 that went undetected in the original testing.

An analysis of variance shows the statistical significance of this comparison. By considering all 731 submissions, grouped by

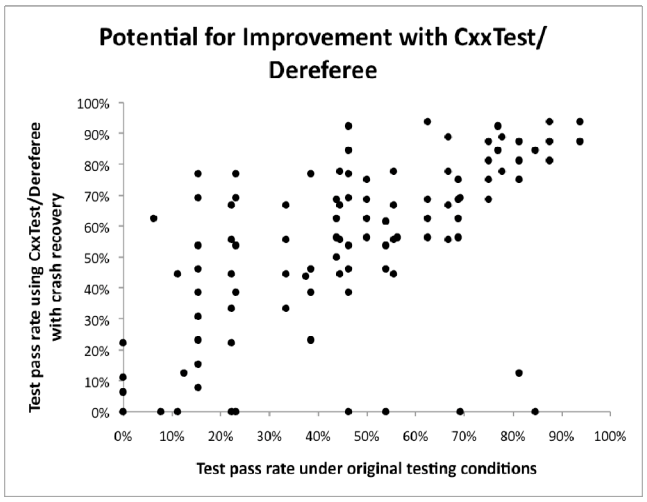

Figure 3. Original test pass rates compared to what they would have been if Dereferee were used.

assignment and by individual student, we performed a matched comparison between each submission under original conditions and the same submission with the Dereferee and CxxTest. The Dereferee pass rates depicted in Figure 3 were significantly higher ($F(1, 1) = 9.5972$ with $p = .0020$). Using Dereferee changed the number of passed tests for 336 of the 731 submissions—the points that lie off the identity diagonal in Figure 3.

Of this subset, the test pass rate mean of 57.7% before Dereferee improved to 65.6% with Dereferee; the median in this same group improved from 53.8% to 75.0%. This shift is noticeable, especially given the points that lie along the horizontal axis in Figure 3—those submissions represent students who passed some tests (as much as 85%) under the original grading conditions even though they consistently mismanaged pointers and thus passed no tests with the Dereferee. These submissions are representative of the hidden errors discussed in Section 4.1. For example, a student who consistently dereferences a dangling pointer, but happens to be lucky enough that the deallocated block's contents have not yet been overwritten, would fall into this category.

4.4 Detecting Memory Leaks

The original test harness for these assignments made no effort to detect or penalize for memory leaks, so it is likely that some students made little effort to prevent them, either because they knew that their scores would not be affected, or because they simply did not understand the concepts well enough to know what to look for and they had no assistance in finding errors.

Before examining memory leak statistics, we must be cautious about which submissions we investigate. Assertion failures and Dereferee memory errors cause a test case to abort prematurely without giving the code a chance to release any allocated resources gracefully. Even if this were not the case, such an error could leave the program's data structures in an undefined state that may not be possible to clean up. Therefore, we can reliably examine only memory leak statistics in submissions for which all tests were successful (153 submissions, or 21% of the entire set).

Of that subset, a majority (85/153, or 56%) contained no memory leaks. Of the remaining 68 submissions, there is significant clustering at both extremes; 42 submissions had minor memory leaks at less than 8% of all objects allocated, 5 submissions

leaked between 14% and 57% of all objects, and 21 submissions leaked significantly, between 73% and 94% of all objects.

Further examination of these data revealed an interesting trend for the BST assignment in particular: 11 of the 27 students who made perfect submissions not only had the same number of objects allocated (as expected in a properly functioning implementation), but also an identical number of leaked objects. This led us to suspect that each of these students may have made the same logical error. Indeed, a manual inspection of each student's code confirmed this: in the binary tree's *remove* operation, each student had failed to delete the node after it was removed from the tree.

Had these students been using Dereferee, not only would they have received a detailed memory usage report at the end of execution, they would also have been notified of these leaks the instant that any live pointers went out of scope or were overwritten, pinning the problem's location down precisely.

This experiment does, however, demonstrate the need for a future enhancement: the inability to separate true memory leaks from those that are secondary results of other bugs not only manifests in our analysis, but provides incorrect information to the student as well. When a test case fails, any objects allocated in that test case up to that point are reported as leaked. While this is technically accurate, since they are never deleted, whether or not they would have been true leaks or are merely artifacts of the testing framework remains unknown. We can improve this behavior by tracking allocated objects more intelligently and marking those that remain after a test failure as "accidentally leaked." This enhancement requires changes to the testing framework and is planned for future work.

5. CONCLUSIONS

Figure 4 shows the entire submission set for the course partitioned into categories based on a comparison between the test pass rate sin the original testing environment and what that rate would have been if the student and instructor had both used Dereferee. These groups are defined based on the amount of additional information that Dereferee provides. Disregarding memory leak information, the toolkit provides no additional information for submissions in Groups A and F (a combined 37% of all submissions). Those in Group A earned perfect scores in both testing scenarios; those in Group F failed some tests identically in both scenarios, but this was due to test assertion failures (value correctness) rather than memory related errors.

The remaining 63% of submissions—affecting **95.6% of students in the study**—would have benefited from using Dereferee. Of them, Groups B and C are indicative of those in which more tests failed due to the uncovering of hidden errors. Group D contains those that would have passed more tests thanks to crash recovery. Group E is the set of submissions that would have passed the same number of tests but would have generated more useful feedback for the student.

This analysis shows that there is a wealth of useful information that students can gain by using Dereferee. Furthermore, when combined with CxxTest unit tests with crash recovery, students' scores would improve even before they investigated the added diagnostic information that this toolkit provides by returning to them points that were lost due to inadequacies in the testing environment rather than their own errors.

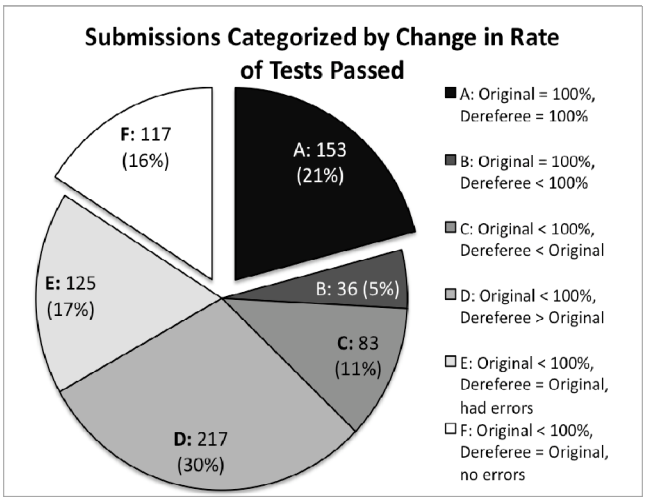

Figure 4. Categorization of submissions based on tests passed under original testing vs. tests passed with Dereferee.

Web-CAT [3, 4] is an automated testing and grading tool that allows instructors the option of using CxxTest and Dereferee for assignments in their C++ courses. From our project web site, http://web-cat.org, interested users can download the Web-CAT server software, a standalone distribution of the Dereferee toolkit, or plug-ins for Eclipse and Visual Studio.NET that provide comprehensive support for using CxxTest and Dereferee.

ACKNOWLEDGMENTS

Dereferee owes its existence to prior work on Checkmate by Scott M. Pike and Bruce W. Weide of The Ohio State University and Joseph E. Hollingsworth of Indiana University Southeast. We also thank William D. McQuain of Virginia Tech, who graciously provided the code from his course for these experiments. This work is supported in part by the National Science Foundation under Grant No. DUE-0618663. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Boost home page. http://www.boost.org/

[2] CxxTest. http://web-cat.org/WCWiki/CxxTest

[3] Edwards, S.H. Improving student performance by evaluating how well students test their own programs. *J. Educational Resources in Computing,* 3(3):1-24, Sept. 2003.

[4] Edwards, S.H. Using software testing to move students from trial-and-error to reflection-in-action. In *Proc. 35th SIGCSE Tech. Symp. Computing Science Education,* ACM, 2004, pp. 26-30.

[5] Lahtinen, E., Ala-Mutka, K., and Järvinen, H.M. A study of the difficulties of novice programmers. In *Proc.10th Ann. SIGCSE Conf. Innovation and Tech. in Computer Science Education*, ACM, 2005, pp. 14–18.

[6] Milne, I. and Rowe, G. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies*, 7(1):55–66, 2002.

[7] Pike, S.M., Weide, B.W., and Hollingsworth, J.E. Checkmate: Cornering C++ dynamic memory errors with checked pointers. In *Proc. 31st SIGCSE Tech. Symp. Computer Science Education,* ACM Press, 2000, pp. 352-356.