Which Pointer Errors Do Students Make?

Conference Paper *in* ACM SIGCSE Bulletin · March 2007

DOI: 10.1145/1227310.1227317 · Source: DBLP

6 authors, including:

Joseph Hollingsworth
Rose-Hulman Institute of Technology
48 PUBLICATIONS **428** CITATIONS
SEE PROFILE

Timothy J. Long
The Ohio State University
47 PUBLICATIONS **805** CITATIONS
SEE PROFILE

Bruce W. Weide
The Ohio State University
165 PUBLICATIONS **2,211** CITATIONS
SEE PROFILE

Which Pointer Errors Do Students Make?

Bruce Adcock[1], Paolo Bucci[1], Wayne D. Heym[1], Joseph E. Hollingsworth[2], Timothy Long[1], and Bruce W. Weide[1]

[1] Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210 USA
+1-614-292-5813
{adcockb,bucci,long,weide}@cse.ohio-state.edu
w.heym@ieee.org

[2] Department of Computer Science
Indiana University Southeast
New Albany, IN 47150 USA
+1-812-941-2425
jholly@ius.edu

ABSTRACT
A model and a taxonomy to characterize pointer manipulations are introduced, along with an instrumentation technology that leverages them to provide students with immediate reports of pointer errors in C++ programs. Data collected from CS2 student assignments show that the vast majority of student pointer errors either would not have been noticed at all, or would have been detected only much later in execution, if this infrastructure were not used. Possible applications of the underlying technology—both to conduct long-term educational research into students' understanding of pointers, and to improve pedagogy directly—are discussed.

Categories and Subject Descriptors
K.3.2. [**Computer and Information Science Education**]: Computer science education, Curriculum. E.1. [**Data Structures**]: Lists, Stacks, and Queues. D.3.3. [**Language Constructs and Features**]: Data types and structures, Dynamic storage management.

General Terms
Languages, Experimentation.

Keywords
C++, CS2, data structures, linked list, pointers, references.

1. INTRODUCTION
Ask anyone who has taught a course such as CS2, in which students are learning to use pointers to build linked data structures: students make plenty of errors with pointers. Yet there is no body of research that expressly asks or answers the question of which errors they make, and why. Of course, pointers involve indirection—and this is a hard concept. But what are the details of students' misconceptions about writing programs that involve pointers? Which pointer manipulations do students most readily understand? Which do they find most challenging? Which errors do they most often make with pointers in their programming assignments? How do they debug these programs? Do they even realize there *are* bugs? Instructors now must appeal to personal experience and anecdotal evidence to try to answer such questions, and for guidance about what to emphasize in class and where to focus student attention. Is it possible for instructors to better appreciate student understanding and misunderstanding of pointers? As with any question of this kind, the answer is "perhaps—if only we had some data!"

Our chief objectives are to show how it is possible to collect such data, and to give a (necessarily short) report interpreting some data we have obtained from CS2 students. Specifically:

- We introduce a model for explaining to students the execution-time dynamics of C/C++ pointers, and a taxonomy for organizing discussion of the correctness issues that arise from their use.
- We explain how we have used this model and taxonomy to give students immediate feedback on pointer errors in their programming assignments.
- We describe how we have adapted the instrumentation used to provide feedback to students so it logs student errors for off-line analyses by the instructor and/or by researchers.
- We discuss data collected with this infrastructure over one year (multiple sections per term) of a CS2 course, and suggest future research studies and pedagogical innovations that it makes possible.

The contributions of the paper lie in all four of the above areas. The model itself is novel. It does not purport to explain what pointers are or exactly how pointers work—either directly, as in "a pointer is a memory address", or metaphorically, as in "a pointer is like an apartment key". Rather, it covers an orthogonal issue: the need for students to realize that every manipulation of pointers is either "always safe", "dangerous", or "never safe". It does this by defining a reasonably simple but language-specific finite-state machine (which we discuss in detail for C/C++) and by classifying its transitions into the above three categories. The pointer-instrumentation technique used is based on *checked pointers* [5], which uses a qualitatively different way to track pointer accesses than had previously been used (only on non-student code, e.g., [2]). We incorporate a few new twists to adapt checked pointers to the pointer-error model and taxonomy of this paper. Finally, there has been previous work on organizing pointer tutorials for students (e.g., [4]) and test-like assessments of students' basic (mis)conceptions about pointers (e.g., [6]). However, ours appear to be the first empirical data to be reported about the actual pointer errors made in student C/C++ programs.

The focus of the paper on C/C++ requires a brief explanation. It is, practically speaking, partly a product of the fact that C++

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

is the delivery vehicle in the introductory courses at the authors' institutions. Though recent reports indicate that C and C++ are not as popular as Java in introductory CS courses, they are used in perhaps 25% of CS1 courses [9], and C++ probably accounts for a similar or even larger share of CS2 courses, where most students meet linked data structure implementations. Industry demand for C and C++ expertise means that even if introductory CS instruction uses languages like Java that try to eliminate certain kinds of pointer errors by providing automatic garbage collection, there remain good reasons to teach students how to build software in which automatic garbage collection is not assumed. Whether this material arises in CS2 as it does in our institutions and many others, or in a later course on system software as it does at most of the remaining ones, the infrastructure described in this paper can be used. Data (see Section 5) show that failure to reclaim storage properly is a common error made by students learning to use pointers. We conclude that students who learn how to develop software only with garbage-collected languages, and then are pressed into service developing C or C++ software on the job, are very likely to mishandle memory.

The paper is organized as follows. Section 2 introduces the model and taxonomy, which serves as the basis for providing students with immediate feedback on pointer errors in their C or C++ programming assignments. We discuss only C++; instructors and students wishing to use C can apply the ideas and software we describe by using a single included class template as a substitute for built-in C pointers, and compiling their otherwise purely-C code with a C++ compiler. Section 3 briefly describes how checked pointers work in C++ and outlines what we changed from the approach used in [5]. Section 4 discusses a few issues that arose during our first year of data collection, including some non-technical issues raised by the very act of logging student errors. Section 5 summarizes some of the data collected so far. Finally, Section 6 concludes with suggestions for long-term empirical research and immediate instructional innovations made possible by the infrastructure. Sections 4 and 5 discuss only the version of the software used, and the data collected, at OSU. Similar conclusions apply to the software used, and the data collected, at IUS.

2. MODEL AND TAXONOMY

This section outlines our model for explaining certain aspects of the execution-time dynamics of C++ pointers, and for classifying problems that may arise from their use.

2.1 Replacing Built-In C++ Pointers

In C++, it is possible to ensure that only "acceptable" (to the instructor) uses of pointers will compile. This prevents certain classes of errors students might otherwise make. For example, we outlaw pointer arithmetic, e.g., using an expression such as `p++` or `p+2`, though it is legal for built-in C++ pointers. Ruling out such constructs can be achieved by requiring students to use a `Pointer` class template in which only certain operators are defined [7]. To declare two pointers to `int` called `p` and `q`, for example, a student simply writes:

```
Pointer<int> p, q;
```

rather than:

```
int *p, *q;
```

The `Pointer` template makes public only operators that the instructor wants students to be able to use, say `*`, `->`, `=`, `==`, and `!=`; nothing else will compile. Figure 1 shows the operators that are public in our classroom-version `Pointer` template, which is used throughout this paper and is available for download at http://www.cse.ohio-state.edu/sce/SIGCSE2007. Here, `p` and `q` are variables of type `Pointer<T>` for the same parameter type `T`; or `q` may be the constant `NULL`. It is technically possible to override **`new`** and **`delete`** to retain all the syntax of built-in C++ pointers except declaration; or, to provide slightly different syntax for some operators (such as the Pascal-like syntax of `New` and `Delete` shown in Figure 1). We have tried both approaches to syntax at our institutions and have seen no apparent impact on student behavior. On the other hand, we have kept the semantics of built-in C++ pointers, warts and all, on the grounds that students should be aware of the problems that can arise from using language-supplied pointers.

Unary	Binary

`New(p);`
`Delete(p);`
`*p`
`p->...`
	`p = q` (assignment and copy constructor)
	`p == q`
	`p != q`

Figure 1: Allowable `Pointer<T>` Operations

So, as with any version of pointers, plenty of code that will *compile* with this approach can still be wrong. Using some of the operators under certain conditions is an error that a compiler, in general, cannot detect, and that typical C++ compilers do not report even when technically they might do so with a sophisticated static analysis. Such errors include dereferencing a null pointer, dereferencing a pointer that has been deleted, creating a memory leak by allowing the last pointer to a block of memory to leave scope, etc. Fortunately, as explained in [5] and discussed further in Section 3, the `Pointer` template can be implemented in a way that (almost) every such error can be detected and reported immediately at the point during program execution where it occurs. By contrast, most errors with built-in C++ pointers—indeed, *all* such errors except dereferencing a null pointer—might not be manifested through observably anomalous behavior during program testing. Even if they are manifested eventually, this might not happen until execution is far beyond the point where the pointer error actually occurred. And even then, there might be only a cryptic system-level error message such as "bus error" or "segmentation fault" that offers no help for debugging.

2.2 Abstract Pointer States and Transitions

The model in this paper has a slightly different purpose than the traditional explanatory devices for pointers. It abstracts the actual value of a pointer (i.e., now `Pointer`) variable—which could be any one of millions or billions of memory addresses—into one of a small set of states. This simplification implies that the model cannot be used to reason *in full detail* about what happens during execution of a program that uses pointers. So, regardless of whether an instructor uses our model, completeness demands that he/she still adopt a traditional explanation of pointer details, e.g., the direct version that a pointer variable's value is a memory address. The four states of a pointer variable in the simplified model are:

- **Alive** — the variable refers to memory that the storage management system has given to the program, i.e., the program "owns" that memory;

- **Dead** — the variable refers to memory that the storage management system has never given to the program or has reclaimed from it, i.e., the program does not "own" that memory;
- **Null** — the variable refers to no memory location at all.
- **Out of scope** — the variable is not in scope.

Figure 2 and an accompanying discussion in class help students understand the model. The diagram shows the states that a single pointer variable, say `p`, can be in; and the various transitions it might undergo via the allowable operators:

- "declare" means the variable `p` is being declared;
- "}" means the variable `p` is leaving the scope in which it was declared;
- "*" means the variable `p` is being dereferenced, using either `*` or `->`;
- "NULL" means the variable `p` is being assigned the constant `NULL`;
- "New" and "Delete" are obvious.

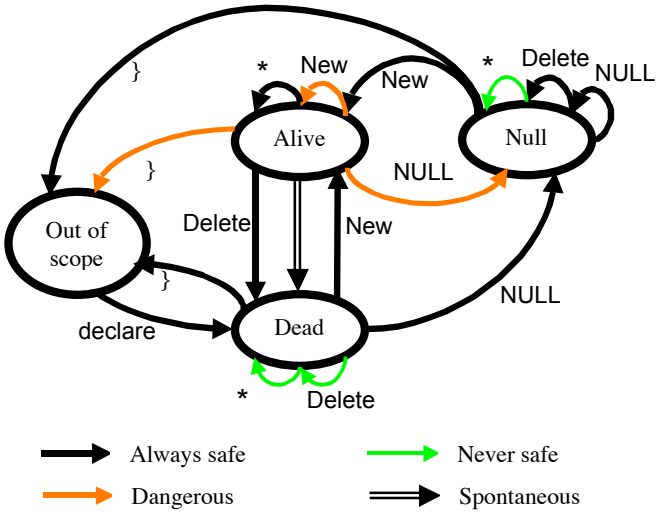

Figure 2: C++ State Machine and Taxonomy of Transitions

An exhaustive analysis reveals that there are exactly two other ways that a pointer variable `p` can change state via the operators in Figure 1: one obvious and the other subtle. Both arise from creating aliases. First, a variable `p` that is in scope can be assigned a pointer `q` that is in a different state, in which case the state of `p` changes to the state of `q`. For example, if `p` is in state **Null** and `q` is in state **Alive**, then after `p = q;`, `p` is also in state **Alive**. None of these six obvious transitions is shown in Figure 2. Second, if `p` is **Alive** and is aliased to another pointer `q` that transitions from **Alive** to **Dead**, then `p` also becomes **Dead**. This is the only *spontaneous* transition, i.e., the only possible change to the state of `p` that can occur as the result of a statement that does not syntactically mention `p`; it captures an interesting effect of aliasing. This transition is shown as an unlabelled double-line arrow to notify students that it is possible, and that they must beware of it.

Of course the other binary operators, which test equality and inequality of pointers, do not cause any state changes.

2.3 Taxonomy of Transitions

This model for explaining pointer states and transitions allows the classification of every manipulation of a pointer as "always safe" (i.e., "good"), "dangerous", or "never safe" (i.e., "bad"). Fortunately, as summarized in Figure 2, many transitions are always safe. A few are never safe, though of course they will compile: dereferencing a pointer that is **Null**, and dereferencing or deleting a **Dead** pointer.

The remaining transitions are "dangerous", in the sense that they may or may not result in memory leaks. If a pointer variable `p` is **Alive** and it leaves scope, then whether there is a memory leak depends on whether the memory referenced by `p` is also reachable via another **Alive** pointer that is aliased to `p`; similarly for the statements `New(p);` and `p = NULL;` in the same situation. That such a highly abstracted model of pointer behavior fails to predict, for sure, whether a memory leak will result from these transitions is a consequence of the fact that it abstracts away actual memory addresses. So, there is a trade-off between the model's completeness, or predictive power, and ease of reasoning. The model's simplicity helps a student (or, potentially, a static analysis tool) identify possible trouble spots in a program without demanding detailed reasoning about actual pointer values. Yet, as Section 3 explains, it provides the ability to immediately detect, report, and log at run-time transitions that are never safe, as well as dangerous ones that turn out actually to cause memory leaks.

Note that the corresponding finite-state machine for Java in Figure 3 is substantially simpler than the one for C++, having but one transition that is never safe and none that are either dangerous or spontaneous. Because there is no explicit storage reclamation in Java (i.e., no "Delete" transition) and no **Dead** state, life is much simpler for the student—and for the professional programmer who is willing to put up with the unpredictable performance of garbage collection. This is precisely why students who have learned how to use Java references to implement linked data structures must *not* be expected to do so competently in a non-garbage-collected language like C++, without further education and practice.

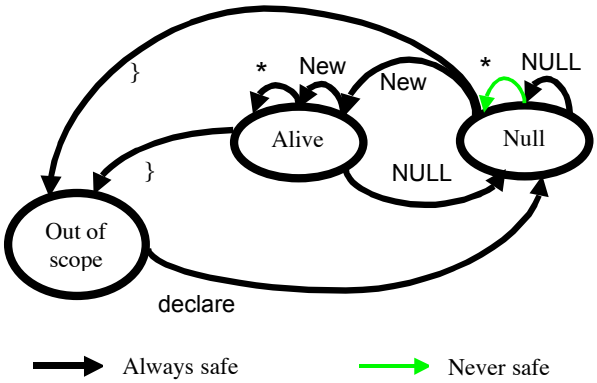

Figure 3: State Machine for Java References

3. CHECKED POINTERS

The `Pointer` template we use is based directly on checked pointers as described in [5]. It supports the operations in Figure 1, and reports at run-time immediately, upon occurrence, every "never safe" transition and (almost) every "dangerous" transition that actually leaks memory. In addition, the

template checks and reports whenever a **Dead** pointer variable is compared using operator == or operator !=. With respect to implementation of the `Pointer` template, interested readers can consult [5] for details and/or download our code.

The qualifier "almost" in the paragraph above is the result of using reference counting to identify memory leaks—an extension to the approach used in [5]. If a reference count for some memory block reaches zero, then there really is no way to access it. However, it is possible to construct circularly linked structures in which blocks of memory are inaccessible yet where all those blocks have positive reference counts: they point to each other. Like [1] and [5], we do not try to detect memory leaks of this sort upon occurrence, but instead wait until the end of the program and report all memory that has been allocated but not yet deallocated. In other words, any memory leak reported by our checked pointers is truly a leak, but not all leaks are immediately reported. Whether this has practical importance depends on how likely students are to write code that creates defective circularly linked data structures, either on purpose or by accident; and this, in turn, depends on the assignments made by the instructor.

Though details are beyond the scope of this paper, we observe that the above-mentioned ideas underlying checked pointers limit one neither to the small set of instructor-approved operators discussed in Section 2.1 nor to student programs. An industrial-strength version of checked pointers we have developed as part of a related research project delivers run-time checking for the normal functionality of regular C++ pointers. To work with code from the wild, we use a TXL program [3] to transform automatically an original C++ source program into code that uses checked pointers, in the process handling dynamic arrays, pointer arithmetic, and casting to checked pointers of different types (including `void` pointers). In addition, this version of checked pointers is thread-safe (at least, as safe as the original program).

4. LOGGING AND ASSOCIATED ISSUES

When a student program generates a pointer error at run-time, the code of the `Pointer` template first reports to the student the nature of the error (e.g., "Deleting dead pointer") and produces a call trace to help in debugging. The code then writes to a central log file all this information, plus the time of the error (accurate to the second), the student's encrypted login name, the name of the program the student was executing, the command the student used to run the program, and the total number of calls executed by the program. Finally, the code uses a C++ `abort` call to quit the program.

At OSU, students do their programming assignments in the introductory courses using department computers, which are servers that they can access from dedicated labs or via the internet. The centralized nature of this facility helped simplify logging at OSU, but it is not essential if it can be assumed that students have internet connectivity over which error reports can be sent from the student's computer to a central site. Error-logging functionality can thereby be provided at the cost of some additional code that might vary according to the institution's local circumstances.

We also faced some technical questions when interpreting preliminary log data. For example, we discovered a situation in which one student experienced the same error some thirty times in a row, at approximately one-second intervals. This could happen if the student ran a script as part of the testing process. Therefore, we decided to count—not while logging but while processing log data off-line—only the first of a series of closely spaced errors by the same student, and to ignore any error by that student within 10 seconds of the previous one. Based on our observations of students in closed lab situations and on our own attempts to make a simple change in a program and then re-compile and re-run it as quickly as possible, at least this much time must elapse between successive legitimate error records. It makes sense to consider the first error as a "true" error and the remainder as spurious.

The most important non-technical issue we faced was a consequence of our mere intent to use logged data in research to be reported to the CS education community. This meant that an institutionally-approved human subjects protocol was required before data collection could even begin [8]. If we had decided to use the error logs only for local instructional purposes (e.g., to identify for special attention students having the most trouble on their programming assignments), then no such requirement would have been imposed.

Our approved protocol included a plan for collecting baseline data without the students' knowledge: a "deception" of the students, in the parlance of such protocols. The reason is that we did not want to preclude being able to study later whether the fact that students know that their errors are being logged might affect their behavior. However, even without a deception, an approved protocol is required for such research.

It also is essential to maintain confidentiality so individual students are not identifiable from any data that could be published. We did not want to preclude longitudinal studies of individual student behavior. Hence, our logging software records encrypted student login names, so if a log file were inadvertently compromised there would be no way for anyone to connect an error record with a particular student. Moreover, only one of the investigators at each institution even has "read" permission for the data log files, in order to further limit the likelihood of such a security breach. These provisions proved acceptable to our Institutional Review Boards.

5. DATA SUMMARY

We logged all student pointer errors at OSU over a one-year period, then filtered the data as explained in Section 4, and also limited the focus to students doing assignments for CS2. Three programming assignments were involved: a closed lab (done in pairs) to implement a stack class, given a queue class as a model; and two open labs (done individually) to implement a singly linked list class and a doubly linked list class. We logged 2765 pointer errors made by 139 distinct students.

Figure 4 lists each possible error message seen by these students, along with two pieces of information for each: the percentage of students making at least one error overall (i.e., 139) who made that particular error at least once, and the percentage of all errors (i.e., 2765) accounted for by that particular error.

Observations from the data highlighted in Figure 4 are:

- About 5/6 of all errors (i.e., all except dereferencing a null pointer) might not have been detected at all without checked pointers; and if they appeared, they would have resulted in later mysterious behavior of the program rather than straightforward error messages.
- Most students who made any error created a memory leak.

- Most students who made any errors at all dereferenced a dead pointer (far more than dereferenced a null pointer); indeed, a third of all errors were of this kind.
- Some errors apparently were easier than other errors for students to correct. Many students made some kinds of errors at least once, yet these errors still account for a relatively small fraction of all errors.

Error	Students Making Error	Percentage of All Errors
Creating memory leak by pointer leaving scope	**74%**	21%
Creating memory leak by using = (i.e., assignment)	**61%**	18%
Creating memory leak by using = NULL (i.e., assignment)	4%	1%
Creating memory leak by using New	1%	0%
Deleting dead pointer	19%	2%
Dereferencing dead pointer by using * or ->	**70%**	**33%**
Dereferencing null pointer by using * or ->	**57%**	**16%**
Using dead pointer with != (i.e., inequality checking)	**30%**	5%
Using dead pointer with != NULL (i.e., inequality checking)	10%	1%
Using dead pointer with == (i.e., equality checking)	13%	2%
Using dead pointer with == NULL (i.e., equality checking)	9%	1%

Figure 4: Results of Off-line Data Analysis

6. CONCLUSION

The answer to the title question is unequivocally: "all of them!" But, of course, it is helpful for an instructor to know more detail about student difficulties. The infrastructure described in this paper can support long-term empirical studies to help provide this detail. Some possible questions include, e.g., investigations of conceptual errors vs. technical ones, examination of the value of pictures such as Figure 2 to explain the model, etc. Other studies are suggested by data in Figure 4. For example, consider the first bullet point above. As instructors, we have little doubt that immediate detection and reporting of pointer errors facilitates student debugging as compared to built-in C++ pointers. This claim could be tested by building multiple versions of checked pointers that provide different error diagnostics to students, while still logging all the data of the current version. One version could allow the program to continue after every error to perform exactly like built-in C++ pointers, providing a control group for the study. Another version could report each error upon detection and stop the program, but always with the same general message such as "pointer error". Another version could provide detailed error messages as shown in Figure 4. Some questions to be answered include: How does student debugging behavior differ under such circumstances? Do students end up making significantly fewer total errors when presented with immediate error detection, and/or when given detailed error messages?

Pedagogical innovations are also facilitated. The infrastructure could be adapted to alert the instructor to students who are making many pointer errors, to those who are making a single error many times in a short period, etc. Such students could be singled out for special attention. Or, an instructor could bring to class a chart showing how many students made each type of error while doing the previous assignment, and call on students to explain the circumstances that led to errors of certain kinds. There are many creative ways in which error data—now, of course, with logging not concealed from students—could be used to focus classroom and/or individual attention on problems students are actually having, rather than on problems the instructor might have expected them to have.

7. ACKNOWLEDGMENTS

We appreciate the many contributions of our students, who have provided helpful feedback on the comprehensibility of the model and taxonomy described in Section 2, and who have provided us with a wealth of log data.

8. REFERENCES

[1] Anderson, P., and Anderson, G. *Navigating C++ and Object-Oriented Design*. Prentice Hall, Upper Saddle River, N.J. ,1998.

[2] Austin, T.M., Breach, S.E., and Sohi, G.S. Efficient Detection of All Pointer and Array Access Errors. *Proceedings SIGPLAN '94*, ACM, 1994, 290-301.

[3] Cordy, J.A., Carmichael, I.H., and Halliday, R. *The TXL Programming Language, Version 10.4*. Viewed 6 Sept 2006 at http://www.txl.ca/docs/TXL104ProgLang.pdf.

[4] Kumar, A.N. A Reified Interface for a Tutor on Program Debugging. *Proceedings Third IEEE International Conference on Advanced Learning Technologies*, IEEE, 2003, 190-194.

[5] Pike, S.M., Weide, B.W., and Hollingsworth, J.E. Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers. *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2000, 352-356.

[6] Postner, L., and Turns, J. Using Facet-Based Assessment to Understand Introductory Programming Students' Knowledge. *Proceedings 32nd ASEE/IEEE Frontiers in Education Conference*, IEEE, 2002, T4G7-T4G11.

[7] Stroustrup, B. *The C++ Programming Language (3rd edition)*. Addison Wesley Longman, Reading, MA, 1997.

[8] U.S. Department of Health and Human Services. Office for Human Research Protections (OHRP). Viewed 6 Sept 2006 at http://www.hhs.gov/ohrp.

[9] Van Scoy, F.L. *The Reid List of the First Course Language for Computer Science Majors*. Viewed 6 Sept 2006 at http://www.csee.wvu.edu/~vanscoy/reid.htm.