Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers

Scott M. Pike and Bruce W. Weide
Computer and Information Science
The Ohio State University
Columbus, OH 43210

Joseph E. Hollingsworth
Computer Science
Indiana University Southeast
New Albany, IN 47150

{pike,weide}@cis.ohio-state.edu
jholly@ius.indiana.edu

Abstract

Pointer errors are stumbling blocks for student and veteran programmers alike. Although languages such as Java use references to protect programmers from pointer pitfalls, the use of garbage collection dictates that languages like C++ will still be used for real-time mission-critical applications. Pointers will stay in the classroom as long as they're used in industry, so as educators, we must find better ways to teach them. This paper presents *checked pointers*, a simple wrapper for C++ pointers that prevents pointer arithmetic and other common sources of pointer errors, and detects all dereferencing and deallocation errors, including memory leaks. The syntax of checked pointers is highly faithful to raw C++ pointers, but provides run-time error detection and debugging information. After debugging, changing one **#include** is all that is required to substitute a non-checking implementation that is as fast as raw C++.

1 Introduction

This is not a paper about chess. But it is a paper about strategy and complexity. The complexity is how to develop and debug programs that use dynamically allocated memory. The strategy employs a simple wrapper for C++ pointers, which we call ***checked pointers***, that detects and reports common bugs such as illegal dereferences. Sometimes this kind of programming mistake surfaces as a cryptic error message such as *segmentation fault* or *bus error*. But sometimes it goes undetected unless it happens to ramify into unexpected run-time behavior. In either case, it is frequently difficult to identify and/or reproduce the root cause of the resulting core dump or snowball behavior. Checked pointers also detect memory leaks, which often remain latent bugs until the most inopportune moments.

War stories of pointer errors are legion, but a couple anecdotes suffice to show they are a real problem in practice.

- The *Toronto Globe and Mail* reported in April of 1997 that the legacy software system for the Toronto Stock Exchange crashed due to a pointer error that had failed to surface in over twenty years of operation. After a large gold discovery in Indonesia, the volume of the stock for Bre-X Minerals Ltd skyrocketed a couple orders of magnitude over normal trading levels. The TSE software read into main memory the entire book of buy and sell orders for a given stock in order to execute a single order. As it turns out, the memory for the book was leaked instead of being properly deallocated if an order was cancelled. The high trading volume of Bre-X led to a higher-than-average number of cancellations, which eventually strangled the system due to a lack of available memory [9].
- The *Wall Street Journal* reported in June of 1998 that a program that reads to children would occasionally toss in foul words. The product would grab text from a screen and send it to a voice synthesizer. It also included a filter that would check for four-letter words and block them. Unfortunately, a pointer-twisting bug occasionally swapped the swear-word list with the list of words to be spoken, and voila, the voice synthesizer had the mouth of a drunken sailor on nickel night [10].

This paper explains what we should teach students about how to use C++ pointers, and how teaching a disciplined approach to using C++ pointers can be supported with simple and free software.

2 Background

The problem of denotation is ubiquitous. In the general case, it is a widespread philosophical problem pertaining to any language and its domain of discourse. In our special case, the domain of discourse is computation, and the problem is how programs attach to the world of hardware computing devices (e.g., memory locations).

It is a folk theorem of computer science that most problems can be solved by introducing a level of indirection. Fortran programmers achieve indirection by using arrays and integer subscripts to model memory locations and their addresses. Other languages (in our case C++) support indirection by providing dynamic memory management and special con-

structs that help compilers detect certain type-related errors. Of course, the C++ programmer using pointers can accomplish the same byzantine manipulations as the Fortran programmer using arrays and integers: namely, doing arithmetic with pointers and treating a pointer as the base address of an array segment. C++ also leaves the programmer responsible for allocating and deallocating storage, but unlike Fortran, it provides built-in implementations of operators **new** and **delete** to help with this.

Java distills the notion of indirection without opening the same can of worms as Fortran or C++. To the Java programmer, nearly everything is a reference and pointers are nowhere to be seen. A reference in Java is not the base address of an array; it acts like a pointer whose value cannot be changed using pointer arithmetic. Moreover, a reference is automatically dereferenced without explicit syntax for doing so. The Java programmer is still responsible for making a reference refer to something (by using **new** or assignment), but is no longer responsible for deallocating the storage it refers to because Java has a garbage collector.

To be sure, then, C++ pointers are a disturbing ilk because they are so easy to misuse, especially when compared to Java references. Table 2 lists the common problems that can arise in both student and professional C++ programs with pointers, some of which cannot arise in Java programs with references.

Problem #1 in Table 2 is an inherent consequence of using indirection to solve the denotation problem. But other than this problem, Java seems reasonable. Why not just move everything and everyone to Java? The downside—to be sure, one that some people hope to remedy someday—is that one of the improvements offered by Java ("solving" problem #7, memory leaks) comes at the cost of garbage collection. This effectively rules out Java for real-time mission-critical applications. The license agreement for Sun's Java Developer's Kit includes the following disclaimer [6]:

> Software [JDK] is not designed or licensed for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. You warrant that you will not use Software for these purposes.

For obvious reasons, you'd hate to have garbage collection kick in when your 747 is about to land. There are incremental garbage collection algorithms that attempt to amortize the cost of reclaiming unused storage, but explicit memory allocation and deallocation will continue be the responsibility of real-time programmers for many years to come. So there is still some point in teaching C++. But as educators teaching C++ we must find better ways to show students how to use C++ pointers wisely and safely.

The remainder of this paper explains how checked pointers can support teaching a disciplined use of C++ pointers. Table 2's last column shows that the C++ pointer problems listed above are addressed—not quite up to Java standards, but probably as well as can be expected if the programmer still has explicit control over storage allocation and deallocation.

3 Design Desiderata

Others have developed commercially-available and prototype tools to help C++ programmers detect pointer errors. Some (e.g., Purify [5]) are intended to deal with legacy code, i.e., C++ source code as it is, not as we think it should be. Purify, in fact, works by instrumenting object code; C++ source is not even required. Other approaches (e.g., LCLint [2]) use static analysis to check programmer-provided annotations regarding the intended uses of pointer variables. Finally, there have been various proposals for "smart" or "safe" pointers (e.g., [3]) that involve programmers living with various degrees of change to pointer syntax. C++ designer Bjarne Stroustrup explains how to code some smart-pointer techniques in *The C++ Programming Language* [8].

In contrast to tools like Purify and LCLint, our approach is not designed for legacy code, and no annotation or static analysis is required. We limit what programmers can do with C++ pointers to essentially what they can do with Java references, *plus* explicit control of storage management. Our approach is similar to "smart" pointers, but with a few twists, as outlined below.

Our desiderata for checked pointers included the following:

- The syntax of checked pointers and raw C++ pointers should be as similar as possible.

It might be possible to have no syntactic differences at all by using just a little more "macro magic" than we used, but is seems this would require replacing the built-in **new** and **delete** operators, about which Stroustrup [8, p. 421] warns:

> ... replacing the global ***operator new ()*** and ***operator delete ()*** is not for the fainthearted.

Not being particularly fainthearted, but also not being C++ gurus, we opted for syntax that is only slightly different (except in declarations of pointer variables) from the syntax of raw C++ pointers. We created a class template, *Pointer*, that is parameterized by the type pointed to. In particular, the only syntactic differences are listed below:

C++ normal pointer example	**C++ checked pointer example**
`int* p;`	`Pointer<int> p;`
`p = new int;`	`New (p);`
`delete p;`	`Delete (p);`

Table 1: Syntax differences

Problem	Problem with raw C++ pointers?	Problem with Java references?	Problem with checked C++ pointers?
1. Difficulty reasoning about side effects due to aliasing	YES	YES	YES
2. Treating a pointer/reference as if it were an integer	YES	NO	NO
3. Treating a pointer/reference as if it were an array base	YES	NO	NO
4. Dereferencing a null pointer/reference	YES	NO*	NO*
5. Dereferencing a dangling pointer/reference	YES	NO	NO*
6. Deleting a dangling pointer/reference	YES	NO	NO*
7. Failure to reclaim storage no longer needed	YES	NO	NO**

* Not detected by compiler, but detected at run-time upon occurrence

** Not detected by compiler or at run-time upon occurrence, but reported on request

Table 2: Problems with raw C++ pointers vs. Java references vs. checked C++ pointers

One advantage of slightly different syntax for **new**/*New*, is that we can log, and later report, important information to facilitate debugging memory leaks (see below). This would not be possible without introducing some syntactic difference for **new**.

- As many errors as possible should be detected at compile-time.

Compile-time prevention of programmer errors is the best strategy, of course. We are able to eliminate problems #2 and #3 simply by not defining arithmetic operators or operator [] for *Pointer* variables.

- Errors not detected at compile-time should be reported immediately at run-time.

When the internal state of a program becomes invalid at run-time, perhaps the best thing it can do is terminate execution immediately and output a meaningful error message. Programmers make mistakes; detecting such mistakes can prevent bugs from propagating through the program execution into compounded or undetected errors. We are able to report problems #4, #5, and #6 at run-time, upon occurrence, by keeping track of the set of memory addresses that have been allocated but not yet deallocated. We detect dereferences of null and dangling pointers by recognizing that requested memory addresses are not in this set. This turns out to be a bit trickier than it sounds, as explained briefly in the next section, although it is conceptually straightforward.

- Errors not detected at run-time upon occurrence should be reported before program termination.

If all else fails, an error report at program termination—"the program had an error, and here's some information about where it was"—is better than nothing. We approach problem #7 in this way by providing a procedure that reports, for each memory address that has been allocated but not deallocated, the file name and line number where its allocation (e.g., "*New* (*p*)") occurred. If this procedure, *Report_Storage_Allocation*(), is called at the end of the program, then every memory address it reports is a memory leak in the program.

- The checking features should be easy to turn off, so that even if checked pointers are too slow to be incorporated in released industrial-strength software, it should be easy to replace the checking version with very fast unchecked pointers without changing more than an include file.

We achieve this by having two implementations of *Pointer*, one that does checking and one that doesn't. The performance of the unchecked implementation—just as Stroustrup claims [7, p. 290]—is very efficient; with inlining, it is essentially indistinguishable from the performance of raw C++ pointers.

- The checking-pointer software should be simple and inexpensive for educators and students.

Our software, including both the checked and unchecked implementations, is small and easy to understand. It is free for anyone who wants it from:

```
http://www.cis.ohio-
state.edu/rsrg/checkmate
```

4 Implementation Considerations

Below is the header file Pointer.h as seen by a client programmer. Private members for our two different implementations—checked and unchecked—have been elided.

```
#define New(p) p.Allocate (__LINE__,
__FILE__)
#define Delete(p) p.Deallocate (__LINE__,
\
    __FILE__)
template <class T>
class Pointer
{
public:
    Pointer<T> ();
    Pointer<T> (void* a);
    Pointer<T>& operator= (const void* a);
    bool operator== (const Pointer<T>& a);
    bool operator== (const void* a);
    bool operator!= (const Pointer<T>& a);
    bool operator!= (const void* a);
    void Allocate (long l, char* f);
    void Deallocate (long l, char* f);
    T* operator-> ();
    T& operator* ();
};
void Report_Storage_Allocation ();
```

Pointer does not supply member functions for arithmetic manipulations or array access. Consequently, any implementation of the *Pointer* class rules out problems #2 and #3 above by design. There are a copy constructor and an assignment operator from **void***, which allows initialization of a *Pointer* to 0 (null) and assignment of 0 to a *Pointer*. By the rules of C++ [8, p. 835], a constant expression that evaluates to 0—but not to any other integral value—can be implicitly converted to **void***, so attempting to set a *Pointer* equal to a non-zero value results in a compile-time error. Similarly, equality and inequality testing is permitted only between two *Pointer* variables of the same type or between a *Pointer* and 0.

Problems #4, #5, and #6 are concerned with deleting and dereferencing null and dangling pointers. As a side note, it is legal to delete null pointers in C++, so that is not an "error" we need to catch. To detect the other three cases, we need to keep track of where each pointer is pointing, and which memory locations are currently allocated. We check at run-time that a pointer points to a legal address before deleting or dereferencing it. If the check fails, execution halts and the error is reported immediately. We accomplish this by implementing a straightforward (but not trivial) memory-location naming scheme along with some run-time allocation accounting.

Finally, we come to problem #7: detecting storage leaks. The header file Pointer.h declares the global operation *Report_Storage_Allocation*. This operation prints out the current contents of the pointer table. Thus, a call to this operation at the end of any client program reports every address that was allocated but never reclaimed, along with the file name and line number of each allocation for debugging purposes. A single call to *Report_Storage_Allocation* reports all memory allocated for *Pointer* variables of all types.

The need for and method of creating a special memory-location naming scheme, and other implementation details, are not covered in this paper due to space limitations. They are explained in a document available at the above URL.

Although checked pointers are slow relative to unchecked pointers, they serve their purpose well in code development, as explained in the next section. After you've debugged your pointer program, if you replace the checked version by the unchecked version, then your resulting code is just as fast as built-in C++ pointers. Moreover, replacing the checked implementation with the unchecked implementation of *Pointer* is as easy as changing one include file; no other source code needs to be edited.

5 Classroom Experience

We previously have found the use of "checking components" in general to be a valuable tool for debugging support, both in classroom and industrial use [1]. It is, therefore, not surprising that we have found checked pointers to be similarly helpful to students.

Two sections of a data structures course at IUS used Microsoft's Visual C++ as the programming environment, generating Win32 Console Applications. Since these are Win32 applications, an alternative implementation of checked pointers was used. This implementation detects errors in exactly the same way as described above, but instead of halting the program with a rational error message about pointer abuse, it displays a message in a debugger's output window that describes the offense (using the Win32 OS call *OutputDebugString*). It then transfers control immediately to the debugger (using *DebugBreak*) [7]. Once in the debugger, the student can read the error message, and immediately find the section of code where the pointer problem occurred.

We observed that with these two pieces of information in hand, tracking down and eliminating the source of the offense in student programs was usually very fast. In fact, it took students a bit of time to get used to such swift feedback concerning their program errors. Initially, they would think: "my program is working, but when I run it, I keep getting this message saying 'user breakpoint called', and I can't figure it out." The 'user breakpoint called' happens when control is transferred to the debugger. Individual and in-class consultation made the students aware that their programs were not defect-free (at least with respect to pointer usage). As can be expected, the more experienced students became with 'user breakpoint called' the more adept they became at finding their pointer-related defects. Consequently, they preferred having checked pointers to help them.

At OSU, we ran a small-scale study in which we divided a class of 34 CS2 students into two groups, and presented

each with identical implementations of a random-access sequence component, except that one used checked pointers while the other used raw C++ pointers. Both implementations contained the same set of six pointer bugs. We timed the members of each group as they worked together in two-person teams to debug the faulty code. Our objective was to assess whether checked pointers were useful in debugging and, if so, in what ways. The actual differences in time-to-completion turned out to be statistically insignificant. But the exit surveys were quite revealing. Students in the control group admitted that testing resulted in bus errors and segmentation faults which forced them to read through the code line-by-line, trolling for bugs. The students in the test group, however, were able to use the debugging information provided by the checked pointers to pinpoint sections of faulty code.

The latter approach to bug detection and identification scales much better with larger projects. The study mentioned above contained fewer than 50 lines of code distributed across six methods. This made the line-by-line analysis of the code feasible enough for the control-group students to ignore information provided by testing. They tended to stop testing after the first bus error or segmentation fault, and proceeded directly to trying to understand the implementation code as a whole. With larger programs and/or a lower bug density, this approach would be much harder to carry out and would be painfully inefficient, even if possible.

Next term we will conduct another study in which students are asked to debug pointer code where the bug density is much lower and therefore more realistic. Results will be posted to the URL in Section 3 when available.

6 Conclusions

This paper contributes ideas that can be instrumental to the improvement of pointer education. On the concrete level, we have presented a real component that is freely available for immediate use as well as for potential refinement. More importantly, however, we have focussed on key insights that are easily accessible, and on design trade-offs and implementation considerations that can be realized in many ways. The outcome is an approach to checkable pointers that is faithful to C++ syntax without loss of performance.

7 Acknowledgment

We gratefully acknowledge financial support from the National Science Foundation under grants DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project P116B60717, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or Microsoft.

References

[1] Edwards, S.H., Shakir, G., Sitaraman, M., Weide, B.W., and Hollingsworth, J.E. A framework for detecting interface violations in component-based software. In *Proc. 5th Intl. Conf. on Software Reuse*, IEEE, June 1998, 46-55.

[2] Evans, D. Static detection of dynamic memory errors. In *Proc. PLDI '96*, ACM, 1996, 44-53.

[3] Ginter, A. *Cooperative Garbage Collectors Using Smart Pointers in the C++ Programming Language*. Technical report 1991-461-45, Department of Computer Science, University of Calgary, 1991.

[4] Harms, D E., and Weide, B W. Copying and swapping: influences on the design of reusable software components. *IEEE Transactions on Software Engineering 17*, 5 (May 1991), 424-435.

[5] Hastings, R. and Joyce, R. Purify: fast detection of memory leaks and access errors. In *Proc. Winter Usenix Conf.*, Usenix Association, 1992.

[6] Java JDK License Agreement (http://java.sun.com/products/jdk/1.1/LICENSE), downloaded 10 Sep 1999.

[7] Lowrey, F. Examining the WIN32 debug API. *Dr. Dobb's Journal*, Nov. 1998, 58-64.

[8] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[9] *Toronto Globe and Mail*, 12 Apr 1997 (http://www.globeandmail.ca/) (Referenced from http://catless.ncl.ac.uk/Risks/19.09.html#subj1.1).

[10] *Wall Street Journal*, 17 Jun 1998 (Referenced from http://catless.ncl.ac.uk/Risks/19.82.html#subj2.1).