10. Interior pointers

Unlike most other languages with garbage collection, this proposal for C++ garbage collection allows *interior pointers,* pointers addressing the middle of objects. (Taking the address of an array element or an object's member yields an interior pointer.)

Language designers typically assume that interior pointers make garbage collection significantly more expensive. If interior pointers are allowed, a collector cannot assume that every pointer addresses the base of an object, and it must map every pointer it follows to its corresponding base. Since that mapping occurs in the collector's inner loops, it might cost a fair bit extra. Also, allowing interior pointers can cause conservative collectors to retain more storage.

Our initial design of the C++ safe subset prohibited interior pointers. Appendix A presents the necessary changes to the safe subset and discusses the implications for implementations. But we eventually decided that it would be better to allow interior pointers, for several reasons.

First, to appeal to the largest number of programmers, we wanted to minimize restrictions on the language. Though practical systems programming languages like Cedar, Modula-3, and Ada prohibit or discourage interior pointers, we wanted to avoid antagonizing the many C++ programmers who would surely object to their absence.

Second, multiple inheritance creates problems not faced by other languages. Given a class `C` derived from classes `A` and `B`, in current implementations widening ("up-casting") a `C*` to a `B*` can create an interior pointer to the storage for `B` inside the `C` object. Thus prohibiting all interior pointers would require radical changes to current implementations of multiple inheritance. Appendix A presents a run-time representation of objects that could efficiently handle the special case of interior pointers resulting from widening casts, but this scheme would require non-trivial changes to current implementations that we think would impede acceptance of C++ garbage collection.

Third, two recently developed families of partially conservative collection algorithms can handle interior pointers fairly cheaply [Bartlett 89, Yip 91, Boehm 91, Zorn 92], and we believe that these algorithms best meet all the other practical requirements of C++ garbage collection (see section 8). We think that allowing general interior pointers will typically cost less than a few percent extra in total program execution time. (See below.)

Samples's language proposal [Samples 92] would facilitate more efficient following of interior pointers by the collector. Implementations could wrap each collected object, member object, and sub-object with a `gc wrapper` that would let the garbage collector quickly find the outermost containing object in the heap. Most likely, the `gc` wrapper would require no extra space in objects with virtual functions if a scheme like that in appendix A is used, but objects without virtual functions would need an extra header word in their representation. The cost of following interior pointers is only reduced, not eliminated, because the proposal still allows the declaration of `embedded` objects and pointers to such objects, with the intention that embedded objects don't have `gc` wrappers; following an embedded pointer would entail the full cost of mapping an interior pointer to the corresponding object base. As discussed in section 3.5, Samples's proposal doesn't meet our criteria of minimal changes and coexistence.

10.1. The run-time cost of interior pointers

The Boehm and Bartlett family of collectors provide heap data structures that allow the collector to map interior pointers to base pointers. The Boehm collector carves the heap into pages, with all of the objects on a page being the same size ("big bag of pages") and a page table giving the size of objects on each page. The Bartlett collectors keep a bit vector per page indexed by word offset, indicating the beginnings of objects on the page. In either scheme, mapping a pointer to a base pointer involves extracting a page index, indexing into the page table, then indexing another table or a bit vector.

On the MIPS processor, this mapping takes about 15 instructions, compared to zero instructions for a language like Modula-3 that prohibits interior pointers. Pointer following is the most frequent operation of a collector, and it must pay this 15-instruction penalty for every pointer it follows.

To get a feel for the magnitude of this penalty, we measured a commercial C++ debugger modified to use version 2.3 of Boehm's collector. Running on a 40 MHz DECstation 5000/240, we drove the debugger with a test script that executed its basic facilities:

> The script executed for 11 minutes, 20 seconds.
> The heap grew to 7.08 million bytes.
> There were 22 collections of the entire heap, taking an average of 1.48 seconds per collection.
> The collector followed 2.68 million pointers to actual objects.

Assume that 10 of the 15 instructions per pointer followed are devoted to handling interior pointers (see appendix A), and pessimistically assume the processor executes 20 million instructions per second. Then only 0.2% of total execution time was spent handling interior pointers, which is about 0.061 seconds per collection or about 4% of the time spent in the collector.

Based on this one experiment, we think interior pointers are quite affordable. At worst, if the typical application spent 5 times as much time handling interior pointers, it would still cost just 20% of total collector time.

However, even if we outlawed interior pointers in the language, it would still be hard to eliminate all of the interior-pointer overhead from the Boehm and Bartlett family of collectors. As discussed in section 8, most implementations in the next few years will probably scan the stacks and registers conservatively, and such scanning needs a way to determine whether an address points at a heap object. The heap structures needed for this determination are very similar to those needed for handling general interior pointers, and the heap allocator must still maintain those structures.

Further, with mark-and-sweep collectors, it's generally more efficient to store object mark bits on the side to increase locality of reference and reduce dirtying of pages during the mark phase of collection, and it's required for concurrent VM-synchronized collectors. Storing mark bits on the side requires heap structures similar to those needed for interior pointers, and the instruction sequence for following a pointer is about the same regardless of whether interior pointers are allowed or not—a page index must be extracted from the pointer and used to index a page table. For example, version 2.3 of Boehm's collector can be configured to disallow interior pointers, yet the sequence for following a pointer is only one instruction shorter.

10.2. The space cost of interior pointers

Interior pointers have no effect on the space efficiency of a totally precise collector, but they can cause conservative collectors to retain more storage. When interior pointers are allowed with a conservative collector, a word is considered to point at an object if it points anywhere within the object, not just at its first byte. This increases the probability that a random bit pattern in a word will be misinterpreted as a valid pointer, and thus more objects may be unnecessarily retained. To the extent that a collector uses precise scanning, fewer words will be misinterpreted as interior pointers.

Fortunately, unnecessary retention isn't a serious problem with totally conservative collectors on newer architectures, especially with Boehm's techniques for improving space efficiency [Boehm 93]. The more limited experience reported for the Bartlett family of mostly copying collectors, which also allow interior pointers but scan the collected heap precisely, indicates that unnecessary retention is not a big problem with those collectors either [Detlefs 90, Yip 91].

11. Code-generator safety

Existing compilers may sometimes generate code incompatible with the correct operation of garbage collectors. Given an expression dereferencing the last use of a pointer `p`, many compilers may generate code that overwrites `p`'s register with a temporary address that points outside the bounds of the referenced object. If a concurrent collection occurs at that point, the object may get collected prematurely before the dereferencing expression completes.

A garbage collector determines which objects are no longer being used by tracing out the graph of reachable objects, using the registers, stacks, and static data segments as roots of the trace. Any object not traced is garbage and can be reused. The algorithms discussed in section 8 consider an address to point at an object if it points at the base of the object or anywhere inside it.

As long as the program thinks an object is live, the code generator should ensure there is at least one reachable pointer pointing at it; otherwise the collector may collect it prematurely. Consider this code fragment:

```
char* a = new char[10];
int i = 20, j = 19;
...a[i - j]...
```

If the expression `a[i - j]` is the last use of `a` (`a` is dead after the expression), then a code generator might decide to generate the following MIPS R3000 code:

```
    # a is in register $a, i in $i, j in $j
    # result will be placed in $r
addu $a, $a, $i
subu $a, $j
lb   $r, 0($a)
```

After the first instruction, register `$a` points at `a + 20`, that is, beyond the end of the object `a`. If there is no other pointer addressing the object, and if a garbage collection occurs at this point, the object would get garbage collected prematurely. (Concurrent garbage collectors can run at any time.)

Reduction of strength of loop induction variables can cause similar problems. For example, an optimizing compiler might transform this source:

```
char* a = new char[10];
for (i = 10; i < 20; i++) {
    ...a[i - j]...};
/* a is dead here */
```

into the equivalent of:

```
char* a = new char[10];
char* p = a + 10;
char* end = p + 20;
/* a is dead here */
for (; p < end; p++) {
    ...*(p - j)...}
```

Inside the loop, `p` always points past the end of the array `a`. The code generator or the calling sequence may cause the register containing `a` to be reused, leaving no pointer that points at or into the object and allowing it to be collected prematurely.

This problem is not academic—many commercial compilers perform these sorts of optimizations. Users of the current Boehm and Bartlett collectors are responsible for avoiding such situations themselves, and some conservative programmers disable optimization entirely. Though the problem appears to be rare in practice, it needs a robust solution.

11.1. A solution

In this section, we'll sketch a solution to code-generator safety that's appropriate for collectors that scan the stacks and registers conservatively, interpreting each word as a possible interior pointer.

Let `p` be a pointer-valued source expression, and let `e` be a dereferencing expression of the form `p[i]`, `*p`, or `p->f`. The compiler must ensure that throughout the evaluation of `e`, some reachable pointer (possibly in a register) points at or into `*p`. The compiler can meet this constraint by extending the lifetime of `p`'s value in the generated object code to include every load and store to an address derived from `p`. For example, if `p` is in a register, the compiler couldn't reuse that register until a load from an address derived from `p` completes.

Let's look at how a traditional compiler might implement this, assuming it generates code for one function at a time and does no inter-procedural optimization. Define a *base pointer* in the compiler's intermediate code to be a pointer-valued variable with no reaching definitions or the pointer-valued result of a function call or load, and a *derived pointer* to be any pointer resulting from address arithmetic on a base pointer or another derived pointer. Intermediate address, load, and store operations can be annotated with the base pointers from which their arguments are derived. For example, consider this intermediate code for the source expression `a[i - j]`:

```
t1 = a + i, {a}
t2 = t1 - j, {a}
x  = load t2, {a}
```

The address temporaries `t1` and `t2` and the load are derived from the base pointer `a`. The annotations must in general be sets, since a derived pointer may have multiple reaching definitions:

```
if e
    t1 = a, {a}
else
    t1 = b, {b}
t2 = t1 + i, {a, b}
x  = load t2, {a, b}
```

The definition of `t2` has two reaching definitions for `t1`, so it is annotated with `{a, b}`.

The compiler's live-variable analysis can treat the annotated base pointers of loads and stores as uses of those pointers, effectively extending the live ranges of the base pointers to include the loads and stores. Once the loads and stores have been annotated with their bases, the annotations of address temporaries can be discarded, and the annotations of the loads and stores will remain valid even after traditional optimizations. The compiler must be careful to ensure that the extended live ranges of base pointers are observed by the later phases of register allocation, instruction selection, scheduling, and peephole optimization.

Suppose the source program creates a derived pointer pointing outside of its referenced object:

```
char* a = new char[10];
a = a + 12;
...
c = a[-12];
```

The behavior of such programs is explicitly undefined under the ANSI C standard and the draft C++ standard, so the compiler is not obligated to maintain code-generator safety. It can assume that a pointer-valued source expression points at or into the object referenced by the expression's original base pointer.

Extending the lifetime of a base pointer `p` isn't required if all pointers derived from it are guaranteed to point into `*p`. Consider this example:

```
struct S {int i; char a[10];} *s;
...s->a[i]...
/* s is dead here */
```

A valid (but suboptimal) code sequence for the R3000 is:

```
addiu $s, $s, 4     /* add the offset of a */
addu  $s, $s, $i    /* index into a with i */
lb    $r, 0($s)     /* load the byte */
```

This sequence is safe, since `$s` always points at or into the object `*s`.

Extending the live ranges of base pointers has little impact on the quality of code generated for modern RISC architectures. In the common cases, either `p` is live at the end of all its dereferencing expressions, or else the temporary pointers created by the expressions point into `*p`. Only when `p` is dead at the end of a dereferencing expression must the code generator consider extending its live range. For a single expression based on `p`, this may cause `p`'s register to be retained for a few extra instructions, and only in rare situations would this cause a register spill. For an optimized loop, `p`'s register could either be retained for the duration of the loop or, if there aren't enough registers, stored in a stack temporary at the beginning of the loop.

Safety is easy to implement in a code generator written from scratch. We used this approach several years ago to add GC-safety to a better-than-pcc-quality Modula-2+ compiler implemented at SRC, and it took less than 50 lines of additional code. Obviously, retrofitting an existing optimizing compiler could be harder, though it should be only a small part of the total cost of adding garbage collection to a C++ implementation.

Our approach to code-generator safety is based on that described by Boehm [92], but somewhat simpler. Their main concern was how to use C as an intermediate code for other compilers, so they worried about handling source not conforming to the ANSI standard. They also assumed that target garbage collectors might not handle interior pointers, requiring more careful handling of base pointers by the compiler.

Diwan et al. describe an approach suitable for totally precise copying collectors [Diwan 92]. Their scheme is considerably more complicated to implement in the presence of optimization, but it allows collectors to relocate any object. The approach described here, since it considers the stacks and registers to contain ambiguous roots, requires the use of partially conservative collectors such as Bartlett's mostly copying algorithm.

Finally, researchers working on smart pointers have tried to provide code-generator safety purely at the source level without modifying the compiler [Detlefs 92]. In general, this isn't possible without increasing the natural size of pointers, relying on details about a particular implementation's compiler, or relying on the vaguely specified, inefficient `volatile` type attribute.

11.2. Unsafe libraries

Our proposal for C++ garbage collection emphasizes coexistence with existing non-collected libraries written in C++ and other languages. In particular, we feel it's very important to allow collected objects to be passed to such libraries.

Unfortunately, it is unlikely that all or even most such libraries will be compiled with safe code generators in the near future. A programmer who writes in the safe subset and uses a safe C++ compiler may still have problems if he must use a library compiled with an unsafe compiler.

Luckily, experience with the Boehm collectors indicates that in practice, lack of safely compiled libraries may not be a serious problem. Objects being manipulated by a function almost always have a base pointer stored somewhere, either in the heap or in a caller of the current function. Further, since objects created and managed by the existing libraries will be in the non-collected heap, the only vulnerable collected objects are those created by clients and passed to the libraries as uninterpreted “client data”. Since the libraries view such objects as `void*` pointers, they won't be dereferencing them, and unsafe addressing expressions won't be executed.

Obviously, conservative programmers won't be completely satisfied with assurances that problems will be “rare”. Ideally, we would give them 100% confidence. Until garbage collection is accepted as an indispensable tool, however, many vendors will see little need to provide safe compilers and safely compiled libraries, especially for languages other than C++. But even without safely compiled libraries, we think garbage collection will greatly decrease the total cost of storage bugs; we think most programmers would rather deal with extremely infrequent bugs caused by unsafe libraries than with the very frequent storage bugs and design problems they see today without garbage collection.

12. Standardization

Before C++ garbage collection can be widely used, there must be a standard that vendors and programmers can rely on. Programmers want assurance they're not tied to a particular vendor or platform, and vendors want assurance that programmers won't view garbage collection as a vendor-specific language extension. Initially, the standard will most likely be a de facto agreement among the few adventurous vendors who might risk providing garbage collection to their users. But eventually, garbage collection will have to be included in the ANSI standard.

12.1. What should be included

The standard should include the following components:

- the language interface (section 4);
- the specification of object clean-up, including the standard interface `WeakPointer.h` (section 5 and appendix F);
- the definition of GC-safety (section 6).

The safe subset is not required to use garbage collection, but for those programmers who want to use the safe subset, its definition should be standardized to allow portability. The safe-subset standard should include:

- the definition of the safe subset (section 7);
- the standard interfaces `Array`, `Text`, and `Variant` (appendices C, D, and E).

12.2. What should be excluded

As part of our system-level approach to adding collection to C++, we've discussed quite a few implementation issues. While implementation issues are important to consider when designing a programming language, the final language design should leave implementors as much freedom as possible to best meet the needs of their customers.

The following implementation issues should be explicitly excluded from a standard:

- requirements for specific algorithms;
- how C++ garbage collection interacts with other languages;
- performance specifications of garbage collection;
- how to implement code-generator safety.

The standard should not specify particular collector algorithms. No single algorithm is ideal for all programs, and we want to allow vendors and researchers as much flexibility as possible for designing new algorithms. The GC-safety rules let programmers write portable programs without relying on particular collector implementations.

In general, a C++ language standard can't specify how C++ should interact with other languages, since it doesn't have control over those languages. Those languages weren't designed or implemented with collection in mind, so there is no way to ensure that implementations of those languages will be compatible with particular implementations of collection. Our proposal allows straightforward collector implementations that should be compatible with many common language implementations. But the degree of compatibility with other languages must necessarily depend on which collector algorithm has been chosen by the C++ vendor and on the implementations of the other languages. C++ vendors (particularly those who also offer C compilers) may choose to sacrifice some general compatibility in favor of more efficient collection algorithms.

Performance specifications of garbage collection (that is, space and time usage) should not be included in the standard, just as the current draft standard does not specify the performance of `new` and `delete` (or any other part of the language, for that matter). To date, no one knows

how to write such a specification that is helpful to programmers, feasible to implement, and flexible enough to allow different implementations of storage management.

Finally, as with all issues of compiler implementation, how code-generator safety is implemented should not be included in the standard.

13. Summary of implementation changes

Here's a summary of the changes that must be made to current C++ implementations to support this proposal.

The language interface requires:

providing the `gc` keyword,
changing `new T` to call the collector's allocator when `T` is a `gc` type,
changing `delete e` to call the collector when `e` points at a collected object.

The safe subset requires:

providing `#pragma safe`,
adding a "safe" boolean attribute to names in the compiler's symbol table,
10 compile-time checks,
6 run-time checks,
2 compiler restrictions.

Both the language interface and the safe subset can be implemented as localized changes to the compiler's front end.

The compiler must be changed to provide code-generator safety.

The sources to the Boehm and Bartlett collectors are freely available for unrestricted commercial use.

14. Conclusion

For better or worse, use of C++ will surely increase over the next many years. Of all the different ways studied by researchers for improving programmer productivity, adding garbage collection to C++ could give a big bang for a small buck.

Researchers have been touting the virtues of garbage collection for three decades, and it's time for us to put up or shut up. To that end, we've presented a two-part proposal. The first part adds garbage collection to C++, and the second part defines an optional safe subset that enforces correct, portable use of garbage collection and precludes storage bugs. Though both parts are important, garbage collection by itself is so valuable that we think near-term efforts should focus on it, rather than the safe subset. Once collection is accepted, it will be easy enough to provide a safe subset for those who want it.

We've only started the long chicken-and-egg process of providing C++ programmers with safe garbage collection. C and C++ programmers tend to be conservative, and experience based on other programming languages doesn't impress them. (It took over a decade for static type safety to be accepted by C programmers.) Most commercial programmers won't use a new language tool until it's widely available on several platforms, but most vendors are reluctant to offer new tools until there is a demonstrated demand for them and there is a corresponding standard. Standards committees are reluctant to standardize technology that isn't yet in wide use. There is no free lunch.

Acknowledgments

Al Dosser and Joel Bartlett helped us early on with the language interface.
Kelvin Don Nilsen and Jerry Schwarz prodded us into thinking more carefully about GC-safety.
Hans Boehm was a patient sounding board for new ideas.
Thomas Breuel sharpened our arguments with lively debate.
Dain Samples helped us understand his alternative proposal.
Bjarne Stroustrup provided helpful feedback on the likely evolution of C++.
Cynthia Hibbard edited the paper.