References

[Appel 91] Andrew W. Appel and Kai Li.
Virtual memory primitives for user programs.
In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991.

[Bartlett 89] Joel F. Bartlett.
Mostly-copying garbage collection picks up generations and C++.
Western Research Laboratory Technical Report TN-12, Digital Equipment Corporation, 1989.

[Boehm 91] Hans-J. Boehm, Alan J. Demers, Scott Shenker.
Mostly parallel garbage collection.
In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* 1991.

[Boehm 92] Hans-J. Boehm and David Chase.
A proposal for garbage-collector-safe C compilation.
T*he Journal of C Language Translation* 4(2):126-141, December 1992.

[Boehm 93] Hans-Juergen Boehm.
Space Efficient Conservative Garbage Collection.
To appear in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation,* 1993.

[CenterLine 92]
CenterLine Software, Cambridge, Massachusetts.
CodeCenter, The Programming Environment, 1992.

[Codewright 93]
Codewright's Toolworks, San Pedro, CA.
Alloc -GC: The garbage collecting replacement for malloc(), 1993.

[Detlefs 90] David L. Detlefs.
Concurrent garbage collection for C++.
CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.

[Detlefs 92] David L. Detlefs.
Garbage collection and run-time typing as a C++ library.
In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[DeTreville 90a]
John DeTreville.
Heap usage in the Topaz environment.
Systems Research Center Report 63, Digital Equipment Corporation, 1990.

[DeTreville 90b]
John DeTreville.
Experience with concurrent garbage collectors for Modula-2+.
Systems Research Center Report 64, Digital Equipment Corporation, 1990.

[Diwan 92] Amer Diwan, Eliot Moss, and Richard Hudson.
Compiler support for garbage collection in a statically typed language.
In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[Dix 93] Trevor I. Dix and Tam T. Lien.
Safe-C for introductory undergraduate programming.
To appear in *The 16th Australian Computer Science Conference*, 1993.

[Edelson 91] D. R. Edelson and I. Pohl.
Smart pointers: They're smart but they're not pointers.
In *Proceedings of the 1991 Usenix C++ Conference*, April, 1991.

[Edelson 92] Daniel R. Edelson.
Precompiling C++ for Garbage Collection.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92.* Springer-Verlag, 1992.

[Ellis 88] John R. Ellis, Kai Li, and Andrew W. Appel.
Real-time concurrent collection on stock multiprocessors.
Systems Research Center Report 25, Digital Equipment Corporation, 1988.

[Ellis 91] Margaret A. Ellis and Bjarne Stroustrup.
The Annotated C++ Reference Manual.
Addison Wesley, 1990.

[Ginter 91] Andrew Ginter.
Design alternatives for a cooperative garbage collector for the C++ programming language.
Research Report No. 91/417/01, Department of Computer Science, University of Calgaray, 1992.

[Hayes 92] Barry Hayes.
Finalization in the collector interface.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92.* Springer-Verlag, 1992.

[McJones 87] Paul R. McJones and Garret F. Swart.
Evolving the UNIX system interface to support multithreaded programs.
Systems Research Center Report 21, Digital Equipment Corporation, 1987.

[Pure 92] Pure Software, Los Altos, California.
Purify Version 1.1 Beta A, 1992.

[Nelson 91] Greg Nelson, editor.
Systems Programming with Modula-3.
Prentice Hall, 1991.

[Owicki 81] Susan Owicki.
Making the world safe for garbage collection.
In *Eighth Annual ACM Symposium on Principles of Programming Languages,* 1991.

[Rovner 85a] Paul Rovner.
On adding garbage collection and runtime types to a strongly typed, statically checked, concurrent language.
Xerox Palo Alto Research Center report CSL-84-7, 1985.

[Rovner 85b] Paul Rovner, Roy Levin, and John Wick.
On extending Modula-2 for building large, integrated systems.
Systems Research Center Report 3, Digital Equipment Corporation, 1985.

[Samples 92] A. Dain Samples.
GC-cooperative C++.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92*. Springer-Verlag, 1992.

[Shaw 87] Robert A. Shaw.
Improving garbage collector performance in virtual memory.
Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, 1987.

[Steffen 92] Joseph L. Steffen.
Adding run-time checking to the Portable C Compiler.
Software—Practice and Experience 22(4):305-316, April 1992..

[Stroustrup 91] Bjarne Stroustrup.
The C++ Programming Language, Second Edition.
Addison-Wesley, reprinted with corrections December 1991.

[Stroustrup 92] Bjarne Stroustrup and Dmitry Lenkov.
Run-time type identification for C++ (revised).
In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[Yip 91] G. May Yip.
Incremental, generational, mostly-copying garbage collection in uncooperative environments.
Western Research Laboratory Research Report 91/8, Digital Equipment Corporation, 1991.

[Zorn 92] Benjamin Zorn.
The measured cost of conservative garbage collection.
Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado at Boulder, 1992.

Appendix A: Restricting interior pointers

Our proposal's language interface and safe subset allow programs to create arbitrary interior pointers (pointers into the middle of objects). In early designs, we prohibited interior pointers, believing that would significantly improve the performance of garbage collection. But as discussed in section 10.1, we now believe the performance improvement to be negligible, compared to the significant costs of outlawing interior pointers and changing existing implementations to achieve the corresponding performance improvement.

This appendix presents our original design for prohibiting interior pointers and taking advantage of the prohibition.

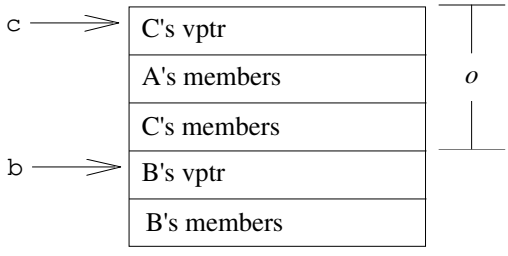
A.1. Base pointers

In a GC-safe program, all pointers to collected objects must be *base pointers*. A base pointer is the result of `new` or a legal widening cast.

Unfortunately, this definition of base pointer can still result in pointers that address the middle of objects. Consider a class `C` derived from `A` and `B`:

```
class A {...};
class B {...};
class C: A, B {...};
C* c = ...;
B* b = (B*) c;
```

Assuming `A`, `B`, and `C` have virtual functions, a typical implementation will lay out instances of `C` as follows:



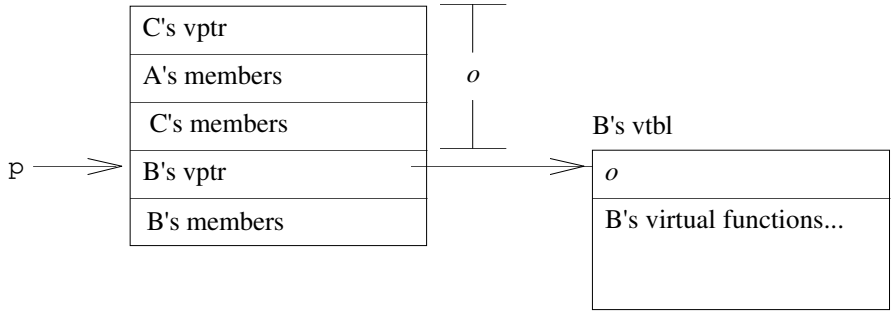
(A *vptr* is a pointer to a virtual-function table.) The pointer `c` addresses the beginning of the object, and `b`, the result of the widening cast `(B*) c`, points at offset *o* in the middle of the object.

In general, a base pointer addresses the base of an object or a sub-object of another object.

A.2. A run-time representation for objects

This definition of "base pointer" could make it harder for the collector to follow pointers—unlike traditional languages like Modula-3 and Lisp, a base pointer does not necessarily address the beginning of an object in the collected heap. But a simple implementation technique allows the collector to follow any base pointer efficiently. The compiler ensures that all `gc` classes are represented with virtual-function-table pointers (vptrs), regardless of whether the classes actually have virtual functions.

The compiler stores in each virtual-function table (vtbl) the offset from the beginning of the object to the slot containing the pointer to that table (offset *o* in the example above).



The collector is thus assured that, in a GC-safe program, every base pointer points to a vptr slot in some object. Given a base pointer `p`, the collector finds the beginning of the object by extracting the offset from the vtbl and adding it to the pointer:

```
p + **(int**) p
```

On the MIPS processor, this takes 5 cycles, compared to the 15 cycles needed to handle general interior pointers.

A couple of details remain.

First, a class `B` may be a base class of many different classes, and the storage for its members may be placed at different offsets in those derived classes. So there needs to be a separate copy of `B`'s virtual function table for each such derived class. (For related reasons, *The Annotated Reference Manual* also suggests a different copy of the virtual function table for each combination of base and derived class.)

Second, `gc` classes with no virtual functions and non-class types allocated in the heap must have vptrs to empty vtbls containing the offset 0.

Third, we'd like the representation of ordinary structs (classes with no base classes) to have the same representation as C structs, which would allow such structs to be passed to C functions. Storing a vptr in the first word would ordinarily make the representation of a struct incompatible. But we can finesse that problem by changing the C++ implementation so that a pointer to an object points at the word immediately following the vptr. Thus, a C++ pointer to a struct will be interpreted correctly by C functions.

Only the collected-heap instances of structs and non-class types need the dummy vptr. To maintain representational compatibility with C, instances that are static, automatic, members of other objects, or elements of arrays can't have a dummy vptr. Consider this example:

```
gc struct S {...};
struct T {S s; ...};
typedef S A[c];
```

To maintain compatibility with C, the instances of `S` that are members of `T` and elements of `A` can't have vptrs. Because the safe-use rules for the collector (enforced by the safe subset) prohibit pointers to those instances of `S` (since they would be interior pointers), the vptrs aren't needed for correct operation of the collector.

A.3. Extending the safe subset to prohibit interior pointers

The safe subset must be extended to prohibit creation of interior pointers. Three additional checks would suffice:

> *Use of the address-of operator* `&` *is prohibited.* Otherwise, `&` applied to a member or array element would yield an interior pointer.
>
> *The operator* `new` *applied to reference-containing types is disallowed.* Otherwise, a reference in a heap object might address an object's member or an array element via an interior pointer.

> *A run-time check prohibits explicit use of* `this` *except when it is a base pointer to a heap object or the operand of* `*` *or* `->`. Otherwise, when `this` gets bound to the address of a member object or an array element, it would be an interior pointer that could then be stored in a heap object.

Since a base pointer can address a sub-object within a containing object, the run-time check needs a quick way of determining whether an object offset corresponds to a valid offset of a sub-object. The compiler can represent the set of valid offsets as a bit vector stored in the class's vtbl.

Using the same mechanism used for conservative scanning of the stacks, the run-time check first determines whether a pointer `p` references a heap object, and if it does, it finds the beginning address `b` of the object. The check then tests whether the bit for offset `p - b` is set in the vector of valid offsets stored in the object's vtbl. Using heap structures like those used in the Boehm collector, the check would take about 18 cycles on the MIPS processor.

Appendix B: Why tagged unions aren't practical

Unions containing collected pointers pose special problems for copying collectors. For example, consider:

```
union {int i; char* p;} u;
```

How does the collector know whether `u` contains an integer or a pointer? A pointer must be relocated by the collector, and an integer must not. In other languages such as Cedar and Modula-2+, variant records have tags indicating their current contents, but in traditional C++ implementations, unions are untagged.

Tagging unions in C++ isn't practical for two reasons: it can't be done efficiently while conforming to C++ semantics, and it would sacrifice representational compatibility with external data structures.

Suppose that the compiler adds an implicit tag to each union and generates code to change the tag when a member is assigned. This works fine for simple assignments to members, but there doesn't seem to be any efficient method of updating the tag when a member is changed by assignment through an alias (a pointer or reference to the member). Consider this fragment, which conforms to the ANSI C and ARM union semantics:

```
union U {int i; char* p;};
U u;
char** ptrToP;

u.i = 0;              /* u contains an integer i */
ptrToP = &u.p;        /* create an alias to u.p */
... = u.i;            /* u still contains integer i */
*ptrToP = ...;        /* assign u.p via the alias */
... = u.p;            /* u now contains pointer p */
```

The union member `u.p` gets changed via assignment through the alias `*ptrToP`. In general, since almost any pointer could be an alias for a union member, there isn't any way a compiler could generate efficient code to maintain union tags.

Even if tagging could be done efficiently, the tags would increase the size of unions. Unions are frequently used to access data defined by hardware devices, external file formats, and modules written in other languages (especially C), and increasing the size of unions would make them incompatible with these external representations.

Appendix C: Array.h

This appendix presents the standard interface `Array.h`, which provides safe arrays. The interface has been tested with DEC's *cxx* compiler.

```
#ifndef   _Array_h_
#define   _Array_h_

/*************************************************************************
```

Safe Arrays

The three template classes in this interface provide safe arrays, which replace built-in C++ arrays in the safe subset. The subscript operator `[]` of these classes checks its index at run-time, causing a checked run-time error if it is out-of-bounds.

An `Array<type, size>` is an array of `size` elements of `type`; it behaves like a built-in C++ array.

A `DynArray<type>` is a heap-allocated array whose size is chosen when it is created. A `DynArray` behaves like a pointer to its heap array.

A `SubArray<type>` references a contiguous sub-sequence of elements in another `Array`, `DynArray`, or `SubArray`. A `SubArray` shares elements with the other array and behaves like a reference combined with a length.

Common operations

The three classes provide the following common set of operations on an array `a` with element type `T`:

`a.Number()`
: Returns the number of elements of `a`.

`a[i]`
: Accesses the `i`-th element of `a`, causing a checked run-time error if `i < 0` or `a.Number() <= i`.

`a.Sub(start, number)`
: Returns a `SubArray<T> s` referencing the elements `a[start]` through `a[start + number - 1]`; that is, `s.Number() == number` and `&s[i] == &a[start + i]`, for `0 <= i < number`. Causes a checked run-time error if `start < 0`, `a.Number() < start + number`, or `number < 0`.

`a.Sub(start)`
: => `a.Sub(start, a.Number() - start)`

`a.Sub()`
: => `a.Sub(0, a.Number())`

`a.Copy(s)`
: Copies the elements of `SubArray s` into the elements of `a`; that is, assigning `a[i] = s[i]`, for `0 <= i < a.Number()`. It is a checked run-time error if `a.Number() != s.Number()`.

`a == s`
: Returns true if `a` and `SubArray s` refer to the same elements; that is, `a.Number() == s.Number()` and `&a[i] == &b[i]`, for `0 <= i < a.Number()`.

`a != s`
: equivalent to `! (a == s)`

`a.Equal(s)`
Returns true if the elements of `a` are equal to the elements of `SubArray s`; that is, `if a.Number() == s.Number()` and `a[i] == s[i]`, for `0 <= i < a.Number()`.

The classes provide implicit conversion of `Arrays` and `DynArrays` to `SubArrays`—an `Array` or `DynArray a` is converted by evaluating `a.Sub()`.

Array-specific operations

`Array<T, n>(s)`
Given `SubArray s`, constructs a new `Array a` of `n` elements and copies the elements of `s` into `a`, using `a.Copy(s)`.

`a1 = a2`
Copies the elements of a2 into a1.

DynArray-specific operations

A `DynArray` can be null, meaning it has no storage associated with it. A null `DynArray` is not the same as a `DynArray` with 0 elements.

`DynArray<T>();`
Constructs a null `DynArray`.

`DynArray<T>(n);`
Creates a non-null `DynArray` of `n` elements.

`DynArray<T>(s);`
Given `SubArray s`, creates a non-null `DynArray d` of `s.Number()` elements and copies `s` into `d`, using `d.Copy(s)`.

`d.IsNull()`
Returns true if `d` is null.

`d1 = d2`
Makes `d1` refer to the same elements as `d2`. After the assignment, `d1.Number() == d2.Number() && &d1[i] == &d2[i], 0 <= i < d2.Number()`.

`d1 == d2`
Returns true if `d1` and `d2` refer to the same elements or both are null; that is, `(d1.IsNull() && d2.IsNull()) || (d1.Number() == d2.Number() && &d1[i]== &d2[i]`, for `0 <= i < d1.Number())`.

The following array operations invoked on a null `DynArray` will cause a checked run-time error:

`Number, [], Sub, Copy, Equal`

SubArray-specific operations

A `SubArray` is implemented as a reference so that it can refer to local arrays in the safe subset. Thus, `SubArrays` can't be assigned or stored in heap objects. The safe subset will detect dangling `SubArrays` to local arrays.

`SubArray()`
Creates a `SubArray` of 0 elements (`s.Number() == 0`).

Initialized built-in arrays can be accessed in the safe subset only by using the special form `SUB(type, array)` to create a `SubArray` accessing the built-in array's elements: