only alternative is to have the compiler generate code that initializes to null all pointer arrays created in safe blocks. A pointer array contained within an automatic variable must be zeroed on entry to the variable's block, and a pointer array contained within an object created by `new` must be zeroed by `new`. (The language already ensures that static variables are zeroed.)

Requiring compilers to initialize pointer arrays is allowed by the C++ definition, which merely states that the initial values of variables are undefined.

7.8. Safe arrays

Safe arrays are provided by three standard template classes defining arrays of fixed size, arrays of varying size allocated in the heap, and sub-arrays of the first two kinds. The overloaded subscript operator of these classes checks at run-time that its index is within bounds, signaling a run-time error if not. The interface to these classes is declared safe, meaning any use by clients cannot violate safety, but their implementations must use unsafe language features. Appendix C provides a completely specified interface and implementation.

These classes are intended to provide a safe, efficient alternative to the basic functionality of built-in C++ arrays, not the full generality of higher-level array abstractions. They would be specified and included as part of the standard classes available with any implementation of the safe subset. Programmers wanting to build their own flavor of safe arrays can use these both as a model and as building blocks.

An `Array<T, n>` is an array of `n` elements of type `T` that behaves like a built-in C++ array. For example,

```
Array<int, 10> a;
```

declares `a` to be an array of 10 integers, `a[0]` through `a[9]`. An `Array` is represented using the built-in C++ array type:

```
template<class T, int n> class Array {
    T a[n];
    ...};
```

A dynamic array `DynArray<T>` is a pointer to a heap-allocated array of elements of type `T` whose size is chosen when the array is created. For example,

```
DynArray<int> d1, d2;
...
d1 = DynArray<int>(100);
d2 = DynArray<int>(200);
```

`d1` refers to an array of 100 integers, `d[0]` through `d[99]`. A `DynArray` is represented as a single pointer to a heap-allocated array:

```
template<class T> class DynArray {
    T* a;
    ...};
```

The size of a `DynArray` is stored by `new` with the array.

A `SubArray<T>` acts as a reference to a sub-sequence of elements in another `Array`, `DynArray`, or `SubArray`. `SubArrays` are constructed using the `Sub` methods of the three classes. For example, if `a` is an `Array<T, n>`, `SubArray<T>`, or `DynArray<T>`, then

```
SubArray<T> s = a.Sub(start, length)
```

initializes `s` to be a sub-array of elements `a[start]` through `a[start + length - 1]`. The elements of `s` share storage with the elements of `a`; that is, `s[i]` is the same lvalue as `a[start + i]`.

`SubArrays` can be used to pass varying-sized arrays by reference. Consider this function declaration:

```
void F(SubArray<int> s);
```

Callers of F can pass an Array, DynArray, or SubArray to F, and the formal parameter s will reference the elements of the actual parameter. The run-time checking of these classes ensure that F cannot index non-existent elements of its actual parameter.

For convenience, Array and DynArray define default conversions to SubArray. For example, given Array<int, 10> a, then F(a) is equivalent to F(a.Sub(0, 9)).

A SubArray is represented as a reference to its elements and a length:

```
template<class T> SubArray {
    T (&t)[];
    int number;
    ...};
```

The array classes support several common operations on an array a :

a[i] accesses the i-th element of a.

a.Number() returns the number of elements in a.

a.Sub(start, length) creates a SubArray referencing elements a[start] through a[start + length - 1].

a.Copy(s) copies the elements of SubArray s into the elements of a.

a == s returns true if a and the SubArray s refer to the same element lvalues, that is, if a.Number() == s.Number() and &a[0] == &s[0].

a.Equal(s) returns true if a and SubArray s are the same size and all of their elements are equal using operator ==.

The operations check their arguments at run-time, preventing out-of-bounds array accesses.

The Array assignment a1 = a2 copies the elements of a2 into a1. DynArrays act like pointers, so the DynArray assignment d1 = d2 makes d1 reference the same elements as d2. SubArrays act like references and thus can't be assigned. The elements of all three kinds can be copied en masse using Copy.

Arrays can be nested to create multi-dimensional arrays. For example,

```
Array<Array<int, 3>, 2> a;
```

declares a 2 x 3 array of integers, similar to the C++ array int a[2][3]. Arrays can be nested inside DynArrays and SubArrays:

```
DynArray<Array<int, 3> > d(5);
SubArray<Array<int, 3> > s = d.Sub(3, 2);
```

This initializes d to a 5 x 3 array and s to be a 2 x 3 sub-array referencing d[3][0] ... d[4][2].

The array classes provide no direct method for statically initializing an array, but programmers can declare SubArrays of initialized C++ arrays using the SUB macro:

```
static int a[] = {55, 66, 73, 99};
SubArray<int> s = SUB(int, a);
```

The SubArray s refers to all the elements of a. The SUB macro expands to a call of a SubArray constructor and its use is always safe. SUB2, SUB3, and SUB4 can be used for two, three, and four-dimensional arrays.

In general, these safe array classes provide fully functional replacements for built-in C++ arrays. Further, SubArrays combined with the Copy and Equal methods provide extremely concise, efficient notations for copying and comparing arrays, mostly eliminating the one

remaining reason some programmers think they “need” pointer arithmetic. (With modern compilers, pointer arithmetic is irrelevant for producing good object code.)

When these classes are used with a good compiler like DEC's *cxx*, the only significant overhead relative to built-in arrays comes from the index checks, which of course can be disabled with a compile-time switch. The classes impose no overhead on modern optimizers, since their overloaded operators expand inline to use the built-in operators. For example, given an `Array<int, 10> a`, the expression `a[i]` is implemented by the expression `aImp[i]`, where `aImp` is declared `int aImp[10]`.

Several people have suggested that these classes should inherit their specification from an abstract base class `Array`. Unfortunately, that would require the use of virtual functions and impose a word of overhead for every array. That word of overhead would be intolerable for many uses of C++ built-in arrays, which these arrays are intended to replace in the safe subset.

7.9. Safe interfaces

The safe subset won't be widely used until most class libraries have safe interfaces. Unfortunately, programmers will be reluctant to provide safe libraries until the safe subset is widely used. A combination of three strategies can help deal with this problem incrementally, without requiring an edict from on high.

First, many common unsafe library functions are subsumed by the standard interfaces `Array` and `Text`. Most applications using the safe subset will have little need to use old interfaces like `string.h`.

Second, it's straightforward to provide safe veneers for most of the other standard library functions. The most common unsafe construct is a function with a `char*` argument representing a string of unknown size; such arguments can be papered over in the veneer with `SubArray<char>`. For example, here's a safe veneer for `read`:

```
int safe_read(int file, SubArray<char> s) {
    return read(file, &s[0], s.Number());}
```

In general, the veneers' interfaces will be safe while the implementations must be written in the unsafe language. Experience with Modula-2+ and Modula-3 shows that it's not hard to provide safe interfaces to the most common standard services [McJones 87, Nelson 91].

Providing safe veneers for complicated interfaces like X or Microsoft Windows is a fair bit more work, but not at all unthinkable for one programmer to do in a few weeks. Many veneers already exist for those interfaces, for example to provide an object-oriented C++ interface or an Ada interface.

Finally, well-written applications tend to isolate their use of low-level system libraries behind a small number of their own interfaces. For example, an application doing binary i/o often hides the calls to the low-level i/o services behind its own application-specific abstractions. The programmer can provide safe interfaces to these abstractions, encapsulating the low-level, unsafe code in one or two modules. If a module needs to call only one or two unsafe functions, rather than defining a whole new interface, the programmer can wrap just those calls with `#pragma unsafe`.

Experience with Cedar, Modula-2+, and Modula-3 shows that this mixed approach is feasible and is not a major roadblock to using a safe language. But we don't want to minimize the amount of care and effort that will be needed to gradually build up a solid core of safe interfaces to the standard libraries.

7.10. Signaling checked run-time errors

By definition, a checked run-time error occurs because of a programming bug such as an out-of-bounds subscript. Precisely how the error is signaled is up to the implementation, but the program should normally halt immediately with enough information for a debugger to find out

where the error occurred. Implementations might choose to print an error message and call `abort()` or, more likely, to raise a private exception.

Programmers should not expect to catch such exceptions by name. Long experience with languages such as Mesa and Modula-2+ shows that allowing programs to handle run-time errors in a programmatic way will often mask true bugs.

For example, suppose the `Array` interface raised a public exception `SubscriptError` for out-of-bound subscripts. A programmer might write a loop indexing through an array, using a `try` statement to terminate the loop when the index got too big:

```
try {
    for (i = 0;; i++) {
        ...a[i]...
        f();
        ...}}
catch (SubscriptError) {}
```

If some other statement in the loop body raised `SubscriptError` as the result of a true programming error (for example, the function `f` might be buggy), the resulting exception would be caught by the `catch`, the loop would terminate prematurely, and the program would continue execution with no indication that a bug had occurred. This defeats the purpose of the safe subset's run-time checks: to detect bugs as soon as they occur.

Another complication arises if programs can catch exceptions raised as the result of checked run-time errors. For efficiency, programmers often want to turn off run-time checking in fully tested production code. The safe subset specifically allows checking to be disabled to encourage its use in the development of production programs. Letting programs catch run-time errors would open up additional avenues for complexity and mistakes, since the programmer mustn't ever disable run-time checks in any code that explicitly handles the resulting run-time errors.

In situations like the example above, it is almost always clearer and more efficient for the programmer to use explicit tests instead of catching run-time errors. Using explicit tests, the programmer is free to disable all run-time checks in a correct program. But if a program relied on catching some of the run-time checks, it wouldn't be practical for the programmer to disable all but the few checks his program relies on. More likely, he would leave all checks in a file enabled, even though most of them would be unnecessary for correct execution.

Programs may catch run-time errors in one special circumstance: to notify the user that an unexpected bug has occurred and to shut down as gracefully as possible. The top level of a program might look like:

```
try {
    ...main body of program...}
catch(...) {
    ...notify user of a bug and shutdown...}
```

Note that shutting down “gracefully” is fraught with peril. By definition, a run-time error means that a bug has occurred and that the program has violated its invariants in an unexpected way. Its state could be arbitrarily corrupted, so the program shouldn't update any persistent objects or files with its state.

7.11. Evaluating the safe subset

Some critics have accused us of trying to ram Modula-3 (or Ada, or Pascal) down the throats of C++ programmers. On the contrary, our sole goal is to improve productivity by reducing the cost of storage bugs, not to impose our broader subjective views about programming languages. Researchers and practitioners might reasonably disagree whether the benefits of the safe subset are worth the costs, but surely this issue requires a dispassionate, detailed examination, not emotional cries of “Fascist programming languages!” or “This isn't C!”.

We've tried hard to minimize the restrictions of the safe subset. We don't know how to pare down the list further without sacrificing GC-safety or imposing much larger run-time costs on programs. We've resisted the temptation to add restrictions that might “improve” the language in ways unrelated to GC-safety.

The benefit of the safe subset is obvious: freedom from storage bugs. But there are two costs, restrictions on the expressiveness of the language and the run-time checks.

Expressiveness. There are four major restrictions on the language:

First, the use of built-in arrays is restricted, but the safe array classes provide equally concise and efficient replacements.

Second, unsafe casting and union types containing pointers, references, or functions are prohibited. These constructs are useful mainly in low-level code dealing with hardware devices and external data formats. Many uses of unions are better expressed with derived classes, and often the space savings of a union are insignificant.

Third, functions taking a variable number of arguments are prohibited. The most common use of variadic functions is to call C's `printf` and friends, and the standard C++ stream library provides an adequately concise and efficient replacement. Default arguments and function overloading subsume most other common uses of variadic functions in C.

Fourth, pointers to automatic variables are prohibited, as is storing references to automatic variables into heap objects. By far, the most common use of taking the address of an automatic variable is to pass an argument “by reference” to a function. But reference parameters and `SubArray`s allow this to be done safely, and safe veneer interfaces provide access to existing library functions like `read`.

Judging from this list, we believe that most programming styles can be readily adapted to use the safe subset with almost no effort and no sacrifice in performance. Further, based on over a decade of experience with languages such as Cedar, Ada, and Modula-2+, we believe that most parts of most applications can be written in the safe subset, with the small unsafe parts neatly encapsulated behind safe interfaces.

Run-time checks. The overhead of the run-time checking is not insignificant. In similar safe languages, the overhead can range from 1 to 15% for programs that don't use arrays heavily, up to as much as 100% for programs that do. (Well-known compiler optimizations can dramatically reduce the overhead of array-subscript checks.) But this overhead is much better than CenterLine's factor of 50 or Purify's factor of two to four, and unlike those tools, the safe subset requires no additional memory.

The relatively low overhead of the checks allows them to be used throughout the development process, even in embedded or CPU-intensive applications where larger performance penalties aren't practical. The checks can be left enabled throughout testing and beta release, and some programmers may well choose to leave checks enabled in released programs where safety is of extreme importance. (Many Ada programs are shipped with run-time checks enabled.)

8. Suitable collection algorithms

The language interface to garbage collection (sections 4 and 5) is compatible with all the major families of collection algorithms. But as a practical matter, some algorithms will be more suited for C++ garbage collection than others.

Practical collection algorithms for C++ must satisfy traditional requirements of garbage collectors:

> *Latency:* interruptions of the program should be short, less than 0.1 second for interactive programs.
>
> *Efficiency:* the collector should be time- and space-efficient, competitive with current implementations of `malloc`/`new`, and should be able to handle very large heaps.
>
> *Concurrency:* the collector should support multi-threaded programs running on multi-processors.

By “competitive”, we mean only that programmers will find that they can achieve their goals more cheaply by using garbage collection. Even if collection were more expensive than explicit deallocation (and recent measurements indicate it often isn't [Zorn 92]), many programmers would gladly pay that cost to increase productivity and decrease bugs.

Compared to previous collected languages like Lisp or Cedar, our design for C++ garbage collection imposes some additional, tougher requirements:

> *Interior pointers:* pointers can address the interior of objects. Traditional collectors support pointers only to the base of objects.
>
> *Cross-heap pointers:* non-collected objects can point at collected objects, and vice versa. Traditional collectors assume the language prohibits storing collected-heap pointers in the non-collected heap.
>
> *Unions containing collector pointers:* pointers to collected objects can be stored in union members. Unlike previous collected languages, C++ doesn't tag its unions.
>
> *Multi-lingual compatibility:* pointers to collected objects can be freely passed to libraries written in other languages, and those libraries' objects can point at collected C++ objects.
>
> *Minimal changes to compilers:* the more special support needed from the compiler, the less likely C++ garbage collection will be accepted and provided by vendors. Some algorithms, such as reference counting, require special language, compiler, or hardware support that wouldn't be practical in today's commercial multi-lingual environments.

In the last several years, researchers have evolved a class of collection algorithms meeting these requirements. Boehm et al. have developed mark-and-sweep collectors [Boehm 91], and Bartlett et al. have developed copying collectors [Bartlett 89, Detlefs 90, Yip 91], and the performance of these collectors is good enough for many C++ applications [Yip 91, Zorn 92]. Both families of collectors use similar technology for satisfying our requirements.

Low latency (short interruptions) is achieved using virtual-memory synchronization to implement generational and concurrent collection [Shaw 87, Ellis 88].

Interior pointers are handled in the Boehm collectors by ensuring that all objects on a page have the same size and by using a table to map page numbers to object sizes. The Bartlett collectors use a bitmap per page indicating the starting offsets of objects.

Cross-heap pointers from the non-collected heap to the collected heap can be handled efficiently with the same VM-synchronization techniques used for generational collection. The non-collected heap is considered part of the root-pointer set of a generational collection, along with global data and the old-space pages of the collected heap. At the start of each generational collection, all the pages in the root set are write-protected, and the subsequent write-protection

faults caused by the program tell the collector which of those pages have been modified; the collector needs to scan just those modified pages, not the entire root set. The cost of this approach for a non-collected heap of *N* bytes and a collected heap of *C* bytes is about no worse than for a totally collected heap of *N* + *C* bytes. This hasn't yet been implemented in either a Boehm or a Bartlett collector, though it would be straightforward and take almost no extra code.

Most of today's major commercial platforms provide the basics for supporting virtual-memory synchronization, including Windows, Windows NT, OS/2, Macintosh, the various flavors of Unix, and almost all of the newer hardware for those systems. The collectors must be able to protect and unprotect data pages, intercept the resulting page-protection faults, and resume the program. The Boehm mark-and-sweep collectors only need write protection, whereas the Bartlett copying collectors need both write and no-access protection. The actual facilities provided by the various platforms may well be an imperfect match for the collectors' needs, and in particular, they may not be very well tuned [Appel 91], but experience to date indicates they are adequate for initial use by C++ collectors. Once paying customers start using C++ collectors, we can expect operating-system vendors to pay more attention to the requisite underlying facilities.

Multi-lingual compatibility and minimizing changes to C++ compilers dictate that the collector must be at least partially conservative. A totally conservative collector treats each word in a register, stack frame, or object as a possible pointer, whereas a totally precise collector knows exactly where each pointer is stored at all times. The Boehm mark-and-sweep collectors are totally conservative, while the Bartlett copying collectors scan the registers and stacks conservatively and the heap precisely. (Copying collectors must scan the heap precisely.)

Conservative scanning requires no compiler support and very little run-time support; the collector just needs access to the stacks and registers. With conservative collectors, libraries written in C and other languages can allocate collected objects without requiring changes to those compilers. Because each word in the registers, stacks, and heaps is treated as a pointer even if it isn't, conservative scanning sometimes retains unused objects accidentally, increasing memory usage.

Precise scanning requires a fair bit of compiler and run-time support (but no language changes). The compiler generates “type maps” describing the locations of pointers within objects, and `new` tags each object with a type map.

Most researchers believe precise scanning of the heap is more time- and space-efficient than conservative scanning, since it examines fewer words, does less work to identify true pointers, and doesn't accidentally retain unused objects. On the other hand, interpretation of type maps entails a fair bit of overhead. No one has yet published good comparisons of the two techniques. However, measurements of the Boehm collectors show that fully conservative scanning yields practical collectors [Boehm 91, Zorn 92]. Boehm has recently developed simple techniques for dramatically reducing accidental retention of objects by conservative scanning [Boehm 93].

While it is technically possible to scan stacks and registers precisely, doing so without compromising code quality is complicated [Diwan 92], and it would be infeasible to modify all the other major language compilers as well as C++ compilers. The performance benefit of precise stack scanning is not large, since the average size of stacks is small and scanning them takes only a small fraction of total collection time.

There's been quite a bit of positive experience with algorithms that scan the stacks and registers conservatively and scan the heap precisely [Rovner 85a, DeTreville 90b, Bartlett 89]. However, those collectors didn't allow interior pointers; once interior pointers are allowed, there's a greater chance that a random word on the stack could be mistakenly interpreted as a true pointer and more unused storage retained. In practice, this doesn't appear to be a significant problem [Boehm 93].

The best C++ collectors might scan the stacks conservatively and use both precise and conservative scanning of the heaps. Objects allocated by C++'s `new` would be tagged with their type map and scanned precisely; objects allocated by C's `malloc` would be untagged and scanned conservatively.

As discussed in section 9, the language interface explicitly allows unions to contain collected pointers. This requires a collector without hardware support to scan pointer-valued union members conservatively, even if the collector uses otherwise totally precise scanning. Handling unions is straightforward for both conservative mark-and-sweep and mostly copying algorithms, which already have the necessary mechanisms to handle ambiguous pointers in the root set.

Detlefs hypothesized that it isn't possible to scan unions conservatively in a concurrent, mostly copying collector [Detlefs 90]. But we now believe that is wrong. A collector could maintain the set of all objects having pointer-containing unions (the set could be implemented as a threaded list). Before scanning the heap, the collector marks all objects that are referents of those pointer-containing unions. If the collector later discovers that a marked object is actually reachable from the root set, it promotes the object rather than copying it (since it may be referenced by an ambiguous pointer). In a VM-synchronized collector, it's easy to do the initial marking from pointer-containing unions concurrently.

9. Unions

C++ unions cause problems for garbage collection. Some algorithms simply can't handle unions containing collected pointers, while those that can don't always perform well. We were faced with a tough choice: If we prohibited unions containing collected pointers, we'd be adding another restriction to the language and sacrificing some compatibility with existing code and programming practice. But if we allowed such unions, we'd be imposing additional constraints on collector implementations and their performance.

In the end, we decided in favor of fewer restrictions on the language. The definition of GC-safety presented in section 6 explicitly allows unions containing collected pointers. But programmers must be warned that such unions may cause particular collector implementations to perform sub-optimally, and they should avoid such unions wherever possible. Obviously, telling programmers that a feature is legal but should be avoided is not ideal. But the alternatives seem worse.

The root of the problem is that, unlike the variant records of Pascal or Modula, C++ unions are untagged, containing no indication of their current contents. Consider the union

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

A collector can't tell whether `u` contains an integer or a pointer, since on most implementations the representations of integers and pointers are indistinguishable. These *ambiguous pointers* cause two problems for collectors. First, conservative collectors simply assume that the union contains a pointer, whether it does or not, and retain the object that is apparently referenced by the ambiguous pointer value. This can cause excess unused storage to be accidentally retained; if pointer-containing unions are used extensively, the excess storage may be unacceptably large [Detlefs 90]. Second, pure copying collectors are in a bind: If the union contains a pointer, its referent must be moved and the pointer updated, but if it contains an integer, its value must be left unchanged. Without special hardware, untagged unions seem to preclude pure copying collectors.

Some have suggested that compilers could add implicit tags to unions identifying their current contents to the collector. But appendix B shows it isn't feasible to tag unions while maintaining C++ semantics. Even if it were, the tagging would increase the size of unions, making their representation incompatible with data structures defined externally by hardware, file formats, and modules written in other languages like C. This would defeat our original purpose for allowing unions, maintaining compatibility with existing code and programming practice.

Prohibiting unions containing collected pointers would sidestep the algorithmic issues, but at the cost of compatibility. For example, clients would be prohibited from passing collected pointers to an existing library that stored those pointers in unions.

We finally decided that minimizing language restrictions and maximizing compatibility were most important, and that collector implementations must handle pointer-containing unions. In practice, this probably means that an otherwise precise collector must scan pointer-containing unions conservatively, as is done in Boehm's mark-and-sweep collectors [Boehm 91] and Bartlett's mostly copying collectors [Bartlett 89]. Note, however, that a collector with otherwise totally precise scanning will incur no performance penalty if the program avoids unions with ambiguous pointers; in particular, a mostly copying collector that scans the root set precisely will behave like a pure copying collector.

Though the proposal requires collectors to work correctly with pointer-containing unions, we recognize that they may be much less efficient if there are many ambiguous pointers. The language interface thus includes a standard `Variant` template class that provides programmers with the functionality of tagged unions:

```
template <class T1, class T2> Variant2 {
      int tag;
      union {T1 t1; T2 t2;} u;
      // public interface to Variant class...
     }
```

For example, `Variant2<int, char*> v` declares `v` to be a tagged union of an `int` and a pointer. A `Variant`'s tag is automatically set when a value is assigned to it, and it is optionally checked when the `Variant`'s value is accessed. See appendix E for the full interface. By using `Variant`s wherever possible, programmers can avoid the collector inefficiencies of untagged unions.

A compiler/collector implementation may be built with particular knowledge about `Variant`s. When it encounters a `Variant`, it can use the tag field to determine its current value and thus scan it precisely. Of course, conservative collectors can handle `Variant`s correctly without any extra implementation work, though they won't receive the performance benefit.

Finally, note that `Variant`s impose no further requirements for GC-safety. Since `Variant`s are implemented with unions, the rules for unions apply equally to `Variant`s. In particular, a GC-safe program must access only the currently assigned member of a `Variant`.