use `delete`; indeed, the safe subset prohibits its use, since it is inherently unsafe. But there are two important reasons for allowing `delete` of collected objects.

First, programmers adapting old code may want to use garbage collection as a backup to catch existing storage leaks, while making as few changes as possible. This suggests that `delete` applied to a collected object should, at a minimum, invoke the object's destructors immediately.

Second, programmers writing new garbage-collected code may want to use `delete` as a performance hint for the collector, suggesting that some particular objects can be deleted immediately. Depending on the implementation, this may significantly reduce the load on the collector. Implementing immediate deletion is easy with mark-and-sweep algorithms, but we don't know how copying algorithms might take advantage of the deletion hints.

Allowing explicit deletion of collected objects lets programmers optimize resource-critical sections of their systems. Often, large systems have small, circumscribed sections that are responsible for large fractions of total storage allocated, and while it may be hard to identify *all* objects that can be safely deleted, it's often easy to identify many or most objects which can. Programmers can use explicit deletion to make the easy safe optimizations, relying on garbage collection to catch any leftover objects that were missed or hard to delete safely.

In both these scenarios, `delete` is inherently unsafe and its use requires care to avoid bugs that can't be detected at compile time or run-time. The program may prematurely delete an object, creating dangling pointers. Also, when adapting old code, the programmer must realize that the collector will invoke the destructors of collected objects asynchronously and that the old code may not be prepared for that.

Despite these problems, we think that allowing deletion of collected objects will be sufficiently useful that it shouldn't be outlawed. It's very easy to implement (collectors can simply ignore the deletion hint). It also follows the spirit of C++, providing programmers with a dangerous power tool; those programmers who don't want to cut their hands off can use our safe subset, which prohibits the use of `delete`.

Finally, declaring a class to be `gc` in effect supplies the class with the garbage collector's allocation and deallocation methods. Thus, there's no reason for a `gc` class also to have an overloaded `new` or `delete`, and any attempt to do so must be a programming mistake.

5. Object clean-up and weak pointers

Two closely related facilities, *object clean-up* and *weak pointers*, allow programs to track when unreachable objects are freed by the garbage collector. Object clean-up lets the programmer specify actions to be taken when an object is no longer accessible and about to be garbage collected. Weak pointers enable the construction of caches of objects that don't require clients to indicate when they are finished using cached objects.

5.1. Object clean-up

When an object is no longer used by the program and is about to be freed, it is often necessary to clean up the object by releasing resources it holds or removing it from a global data structure. For example, if an object contains an open file handle, the object's clean-up might close the file. Or if an object contains a window handle, the clean-up might release the handle back to the window system. In general, if an object contains some resource controlled by another program, the operating system, or a non-collected library, a clean-up action can release the resource when the object is garbage collected.

C++ supports clean-up with destructors—when an object of class `T` is about to be freed, the destructors of `T` and its base classes are applied to the object. Automatic objects are freed when their scope is exited; static objects are freed when the program terminates; and heap-allocated objects are freed when `delete` is called on the object. (Note that storage allocated by an overloaded `new` should be released by an overloaded `delete`, not by a destructor.)

In our proposal, garbage-collected objects are also cleaned up using destructors. If the programmer wants objects of `gc` class `T` to be cleaned up when they are about to be garbage collected, he writes a destructor `~T()` that performs the clean-up. When the collector determines that an object is unreachable by the program, it calls the object's destructor before freeing it. The collector considers an object unreachable if it can't be accessed by following a path of pointers starting from static variables or automatic variables of active functions.

If collected object `B` is reachable from collected object `A`, then `A`'s destructor will be invoked before `B`'s destructor. This ensures that `A`'s destructors will see a fully formed, non-cleaned up `B`. `B` will be cleaned up only after `A` has been collected. (If `A` and `B` form a cycle, neither will be cleaned up or collected.)

An explicit clean-up function `f` can be registered for an object `t` of type `T` using the standard interface `CleanUp`:

```
CleanUp<T, void>::Set(t, f)
```

(The default clean-up function for an object simply calls its destructor.) Clean-up for a particular object can be disabled entirely by setting the clean-up function to null:

```
CleanUp<T, void>::Set(t, 0)
```

Programmers can force an object's clean-up to be invoked immediately either by calling

```
CleanUp<T, void>::Call(t)
```

or `delete t` (whose implementation calls `CleanUp::Call`). An object's clean-up is called at most once, unless it is explicitly re-enabled by calling `CleanUp::Set`.

There is no guarantee that the collector will detect every unreachable object and invoke its destructors. Conservative collection algorithms find almost all unreachable objects, but not all of them, and any algorithm likely to be used for C++ in the next several years will almost certainly use a conservative scan of stacks and registers. Thus, programmers should treat object clean-up as a mechanism for improving resource usage, and they should not rely on having clean-ups applied to 100% of their objects.

Finally, if programmers need some action to occur on heap objects when the program exits, they should use a termination service like that described by Stroustrup [91, page 466].

Termination actions are not the same as destructors—the destructor of a heap-allocated object won't necessarily be called when the program exits.

The precise semantics of object clean-up and the `CleanUp` interface are presented in appendix F.

5.2. Clean-up asynchrony

Because the collector may run at arbitrary times, a collected object's destructor may be invoked asynchronously with respect to the main thread of execution. This isn't a problem if the destructor side-effects data that is reachable only from the object, since by definition when the destructor is invoked, no other parts of the program can access the object. But sometimes a destructor must access global data or other objects that are still accessible to the rest of the program, and in these cases such access must be synchronized to avoid races.

In multi-threaded environments, synchronizing concurrent access is straightforward using well-understood techniques. In general, programmers must synchronize access to all global data, so synchronizing destructors takes little extra effort.

But in a traditional single-threaded environment, programmers usually assume there is no concurrency, and a naively programmed destructor could access inconsistent data. In these environments, destructors can synchronize using queues provided by the standard interface `CleanUp`. You can declare a clean-up queue for instances of a type `T`:

```
CleanUp<T, void>::Queue q;
```

Calling `q.Set(t)` tells the collector that when `t` becomes unreachable it should enqueue it on `q` instead of calling its destructor. The program can poll `q` periodically at safe points by calling `q.Call()`. Each such call removes the first object from the queue and calls the object's destructor; `q.Call()` does nothing if `q` is empty. An example of this technique is presented in section 5.4.

5.3. Clean-up rationale

Advocates and critics of C++ garbage collection have been gnashing teeth over object clean-up and destructors. Two issues arise: should there be object clean-up at all, and if so, how should clean-ups be specified syntactically?

Should the collector provide object clean-up? There's been over a decade of experience using garbage-collected languages such as Lisp, Cedar, Smalltalk, Clu, and Modula-2+ to build long-lived applications, servers, and operating systems. In these environments, programmers have found object clean-up indispensable for managing in-memory caches of objects and releasing resources provided by other programs, servers, operating systems, and non-collected libraries [Hayes 92]. There's every reason to expect that object clean-up would be equally useful in systems built with C++.

In the second edition of *The C++ Programming Language* [Stroustrup 91, page 466], Stroustrup provides widely quoted arguments against collector-based clean-up. First, he argues that garbage collection simulates an infinite memory from which objects never get deleted; since the objects are never deleted from the (simulated) infinite memory, the collector shouldn't invoke their destructors. This argues by analogy without considering whether the analogy is valid, and like most such arguments it fails to address the basic question: What is most useful for building systems? In fact, most programmers using collector-based languages view the collector as automating calls to `delete`, and under this analogy, it's quite sensible for those calls to invoke clean-ups.

Stroustrup suggests that destructors (clean-ups) should only be invoked as the result of explicit calls to `delete`. But in general this requires programmers to determine when an object is no longer being used before invoking their clean-ups. Such a requirement defeats the major purpose of garbage collection, removing that burden from programmers. (Note that our design allows the programmer to force immediate invocation of destructors by calling `delete`.)

Stroustrup then implies that program-termination actions can replace most collector-driven clean-up actions. (Termination actions are registered functions that get applied to an object when the program terminates.) But of course, termination actions are different from timely clean-up—they have different purposes, and programmers want both. Clean-up actions release resources in a timely manner during program execution, whereas termination actions ensure some action is taken *only* when the program exits. (Note that programs such as servers never exit.)

Many critics are discomfited by asynchronous clean-ups. Stroustrup argues that asynchronous collector-driven clean-up is "hard to program correctly and less useful than is sometimes imagined". But we show in sections 5.2 and 5.4 how to program asynchronous clean-ups correctly with no fuss or muss, even in traditional single-threaded C++ environments. And programmers in those other collector-based languages would certainly dispute the implication that clean-up isn't very useful.

Note that destructors may be applied at unexpected times even in standard C++. In the *ARM*, the exact point at which destructors of temporary objects are called is implementation-dependent. The ANSI standards committee is currently wrestling with the problem of when destructors of automatic objects should be invoked, and no matter what they decide, this issue will continue to hold subtle surprises for the unsuspecting programmer.

Interestingly, most examples illustrating problems with C++ destructors involve freeing storage. The most common use of destructors is to free storage, and garbage collection eliminates the need for such destructors. Based on experience with other collected languages, very few collected classes will need explicit clean-up actions. Thus in practice, introducing garbage collection will eliminate most uses of destructors and simplify their use.

How should the syntax of clean-ups be specified? There are two choices for specifying clean-ups: functions explicitly registered with the collector, or C++ destructors. We think destructors are somewhat better, though explicitly registered functions would be adequate.

In languages such as Cedar and Modula-3, programmers write clean-up functions and register them with the collector. Since this requires no syntactic support from the language, some critics suggest this is the best way of specifying clean-ups.

But registered clean-up functions require non-trivial programmer conventions to support modularity and class derivation. Consider a class `U` derived from `T`, with both classes desiring to clean up their private members. Any convention must allow `U` and `T` to register clean-up functions independently in their respective constructors, while ensuring that `U`'s clean-up is called before `T`'s. Further, each class's clean-up function must also remember to call its members destructors.

In light of this, perhaps the best convention would put the clean-up actions for a class in its destructor and register a clean-up function that simply calls the destructor:

```
class T: S {
    T() {
        CleanUp<T, void>::Set(this, CallDestructor);
        ...};
    static void CallDestructor(void* d, T* t) {
        t->T::~T(); };
    ~T() {   /* clean-up actions for T */
        ...}
    ....
    };
```

As long as the base classes of `T` also follow this convention, their clean-up actions will be invoked in the proper order, after `T`'s clean-up actions. Even if the base classes also register clean-ups, all objects of class `T` will end up with the clean-up function `T::CallDestructor`, since `T`'s constructor runs after its base-class constructors. `T::CallDestructor` calls `T::~T()`, which calls the destructors of the base classes after executing the body of `~T()`.

Compared to our design that registers destructors automatically, this convention has several minor disadvantages. First, programmers can make clerical mistakes, forgetting to register a clean-up function in every constructor of class. Such mistakes may be more likely in classes with many constructors or when new constructors are added to an old class by a programmer who isn't the original author. Second, `CleanUp::Set` may be called several times during an object's construction, whereas one call is sufficient. In a typical implementation, the runtime cost of each extra call could be non-trivial, roughly the same as the cost of the allocation itself. Third, the construction of static, automatic, or non-collected heap instances of such classes will call `CleanUp::Set` unnecessarily (`CleanUp::Set` would do nothing if passed non-collected objects). Finally, the convention is more verbose, and programmers would get annoyed that the language doesn't register the destructors automatically. Destructors get called automatically for static, automatic, and non-collected heap objects, so why exclude collected objects?

Some critics (reportedly including Stroustrup) say that having the garbage collector invoke destructors asynchronously changes the semantics of the language, and thus existing applications would break. This argument assumes a different model for adding garbage collection to the language: `new` would be redefined to allocate *all* objects from the collected heap. Under this design, existing code would indeed sometimes break, since such code often depends on having destructors invoked synchronously at somewhat well-defined points.

However, our design is different: We've argued in section 2.2 that, for several reasons, coexistence with existing libraries requires both collected and non-collected heaps, and that by default `new` continues to allocate from the non-collected heap. Thus, the semantics of destructors for non-collected objects is *not* changed, and existing code continues to execute correctly side-by-side with new code written to use garbage collection. We've optimized our design for writing new code to use garbage collection, while retaining strict compatibility with old code.

Note that if a collected class `B` is derived from a non-collected class `A`, `A`'s destructor could get invoked asynchronously when instances of `B` are collected, and the destructor may not be prepared for that. In this case, the behavior of non-collected objects hasn't been affected, though the behavior of the collected instances of `B` is less than ideal. We weren't willing to flatly prohibit deriving a collected class from a non-collected class with a destructor, since we thought that in many cases that could be useful, and that this situation would arise only when trying to allocate collected instances of a non-collected class exported by an existing library. The compiler can of course give a warning, and the programmer can use clean-up queues to control the invocation points of the destructors or he can override the destructor with his own clean-up function.

5.4. Weak pointers

Weak pointers allow a class to track which objects are being used by other parts of the program. The collector ignores weak pointers when tracing reachable objects, so a weak pointer to an object won't prevent the object from getting collected. The most common use of weak pointers is to build caches of objects in which cached objects are automatically deleted by the collector when clients no longer reference them. In contrast to the traditional way of implementing such caches, clients of weak-pointer caches need not tell the caches when they are finished using an object.

For example, suppose a window server contains a cache of in-memory fonts, keyed by font name. Fonts consume a lot of memory, so when clients of the window server no longer reference a font, it should be deleted automatically from the cache, without requiring notification from clients.

Weak pointers are defined by the `WeakPointer` template class (appendix F); no special language support is needed. A weak pointer is constructed from a normal pointer `t` of type `T` using the constructor:

```
WeakPointer<T> wp(t);
```

The `Pointer` method translates a weak pointer back to a normal pointer:

```
T* t1 = wp.Pointer();
```

`Pointer` returns the original pointer, unless the weak pointer `wp` has been *deactivated*, in which case it returns null. The collector deactivates a weak pointer when it garbage-collects the referenced object; that is, when the object becomes unreachable by paths of normal pointers from static variables and automatic variables of active functions.

In our example of a window server's font cache, the server could implement the cache as a table of pairs <font name, weak pointer to font>. Because the table uses only weak pointers to reference its fonts, it won't cause those fonts to be retained in memory by the collector—a font will remain uncollected only if a client still references it.

Here's a sketch of the `FontCache`:

```
class Font;

class FontCache {
    Table<char*, WeakPointer<Font> > table;
public:
    Font* Get(char* fontName); }

FontCache fontCache;
```

The `Get` method returns a font named `fontName`, reading it into memory from disk if it isn't in the cache. Its implementation looks like:

```
FontCache::Get(char* fontName) {
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font; }
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    return font;}
```

`Get` looks in the table for an entry keyed by `fontName`. If there is such an entry, and the font referenced by the entry's weak pointer hasn't yet been garbage collected, `wp.Pointer()` will return a non-null `Font*`, which is returned to the client. If `wp.Pointer()` returns null, that means that clients no longer reference the corresponding font and it has been garbage collected. In this case, and in the case of no entry at all for `fontName`, `Get` reads the font from disk and installs it in the table before returning it.

Note that, over time, the table could fill up with entries whose fonts have been garbage collected. Often, programmers can ignore this problem, since the table entries themselves are small and there often aren't that many entries. If it is a problem, though, there are a couple of straightforward solutions.

First, `Get` could scan the table whenever it fills up, deleting entries whose weak pointers have been deactivated (`wp.Pointer() == 0`). Assuming the table is a dynamically growing hash table, this would increase the time cost of the hash table by a small constant factor.

Alternatively, the table can use object clean-up of fonts. A font's destructor can delete the corresponding entry from the cache:

```
Font::~Font() {fontCache.Delete(this);}
```

However, the collector may call the destructor during the execution of some other operation on `fontCache`, creating a race condition. So we must use a clean-up queue (section 5.2) for synchronization:

```
class FontCache {
    CleanUp<Font, void>::Queue q;
    ...};
```

When `Get` adds a font to the cache, it calls `q.Set(font)`, telling the collector that when `font` becomes unreachable by normal pointers, it should enqueue it on `q` rather than invoking its destructor. In addition, `Get` calls `q.Call()` before doing its cache look-up, invoking the destructors of any inaccessible fonts enqueued on `q`, thereby removing them from the table. The enhanced `Get` looks like:

```
FontCache::Get(char* fontName) {
    while (q.Call());    /* call ~Font() in a safe place */
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font;}
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    q.Set(font);      /* Set font's clean-up queue */
    return font;}
FontCache fontCache;
```

The precise semantics of weak pointers and their interaction with object clean-up is specified in appendix F.

6. Safety

Any proposal for C++ garbage collection must define the language rules programs must obey to ensure the correct use of garbage collection. A program following these rules is *GC-safe.* Programmers and C++ implementors need a precise language definition of GC-safety to ensure that programs can run on any conforming C++ implementation (subject to memory availability). That is, the GC-safe rules provide a standard interface between the program and the garbage collector, allowing many different programs to run with many different collector implementations.

Our definition of GC-safety is broad enough to encompass all the major families of collector algorithms, yet simple enough for working programmers. To write a portable GC-safe program, the programmer need only follow the usual C++ portability rules plus one additional restriction.

6.1. Definition of GC-Safety

We've adopted Owicki's approach to defining GC-safety [Owicki 81]. The specification of GC-safety is a promise to the program: if all the program's actions are “legitimate”, then the garbage collector will remain “invisible” to the program. A garbage collector is invisible if it doesn't free any objects still in use by the program and it doesn't make invalid changes to those objects. The rest of this section defines “legitimate” program actions.

Legitimate program actions must maintain three general properties: separation of memory, visibility of collected pointers, and visibility of assignments. (In what follows, a “collected pointer” is a pointer to a collected object. Also, by “pointer” we mean both C++ pointers and C++ references.)

Separation of memory. The program shouldn't change memory locations belonging to the collector. If it did, say by overwriting structures used to maintain the collected heap, the collector might accidentally reuse the storage of an object still in use, or it might relocate just part of an object.

Further, the program should create new collected objects or new non-collected objects containing collected pointers only by declarations or by invoking `new`. If the program acquired new memory through some mechanism unknown to the collector and then stored collected pointers in that memory, the collector might not be able to find those pointers, and it might prematurely free an object.

Visibility of collected pointers. All collected pointers should be visible at all times. That is, they should be stored in variables, members, or array elements declared with type “pointer”, or they should be the results of expressions whose type is “pointer”. The locations of pointers within an object are fixed by its declared type (the declaration or the argument to `new`).

A garbage collector needs to know exactly where all collected pointers are located at all times, so that it can determine which objects are still in use and possibly relocate those objects. If a pointer is hidden, say, by casting it to an integer or storing it in a location not declared to be a pointer, then the collector may be fooled into thinking its referent object is no longer in use and freeing it prematurely. Even if some other visible pointer also points at the object, a relocating collector couldn't correctly update the hidden pointer after moving the object.

Visibility of assignments. All assignments of collected pointers must be visible to the garbage collector. That is, if a variable, member, or array element currently contains a collected pointer, or if its value is to be changed to a collected pointer, then it must be changed either by initialization to a pointer value or by an assignment expression in which both the lvalue and rvalue have type “pointer”. Some incremental collector algorithms rely on the compiler to generate special code for pointer assignments. Changing collected pointers through some mechanism other than a pointer-typed initialization or assignment (for example, `memcpy`) would hide the assignment from the collector, perhaps causing it to prematurely free the object referenced by the rvalue.

For the purposes of determining safety, we assume that all overloaded assignment and `new` operators have been expanded to their definitions and that all assignments of whole objects have been expanded into their equivalent member-wise assignments.

6.2.Writing portable GC-safe programs

The purpose of GC-safety is to provide a set of rules that let programmers write portable programs that run correctly on many different collector implementations. Despite the apparent complexity of the definition of GC-safety, it's straightforward to write a portable GC-safe C++ program. The programmer need only follow the usual C++ portability rules plus one additional restriction.

A program is guaranteed to be GC-safe if it follows these rules:

> It doesn't execute any of the constructs listed below that the *ARM* labels “undefined” or “implementation-dependent”.
>
> It doesn't cast an integer to a pointer, unless the integer resulted from casting a non-collected pointer and the referent of the pointer is still allocated at the time the integer is cast back to a pointer.

(A program that doesn't follow these rules may still be GC-safe on particular implementations.)

The following undefined or implementation-dependent constructs could, perhaps in combination, violate GC-safety on some implementations. The constructs are labeled with the corresponding section of the *ARM*:

> accessing an uninitialized variable, member, or array element (8.4)
> accessing a union member after a value has been stored in a different member of the union (5.2.4)
> dereferencing a null pointer
> accessing a dangling pointer or reference (5.3.4)
> applying `delete` to a pointer not obtained from `new` (5.3.4)
> illegal pointer arithmetic—adding to a pointer not referencing an array element, or arithmetic resulting in a pointer outside the bounds of the array (except for one past the last element) (5.7)
> a subscript expression whose equivalent in pointer arithmetic is undefined (5.2.1)
> all casts to types containing pointers, references, and functions, except legal widening casts (“up-casts”), legal narrowing casts (“down-casts”), casts between pointer types and `void*`, and the casts from integers to pointers described above (5.4)
> casting a pointer to an integer (5.4)
> exiting a value-returning function without an explicit `return` or `throw` (6.6.3)
> using variadic functions (ellipsis) incorrectly (8.3)

By definition, it's impossible to know the behavior of undefined or implementation-dependent constructs without reference to a particular implementation. Thus, a programmer writing a truly portable program must avoid such constructs regardless of whether he's using garbage collection.

Obviously, some of the constructs have well-defined behavior on some implementations. To decide whether they are GC-safe on a particular implementation, a programmer would have to refer to the general definition of GC-safety and any “specifications” provided by the implementation's vendor. For example, with a fully conservative mark-and-sweep collector, a program could safely hide collected pointers by casting them to integers and it could copy pointers using `memcpy`.

Casting an integer to a pointer could, in general, violate pointer visibility by hiding the pointer as an integer. If the resulting pointer is invalid, dereferencing it could overwrite memory locations belonging to the collector, thus violating separation of memory.

Casting a pointer to an integer yields an implementation-dependent result (*ARM* section 5.4), but programmers often assume that repeated casting of the same pointer will yield the same integer. Of course, this is no longer true with a relocating collector, so programmers wishing to write portable programs should avoid depending on the results of such casts. Though technically, these casts don't violate GC-safety, they violate the spirit of "invisibility" of garbage collection.

6.3. Pointer validity

Our definition of GC-safety allows pointer variables to contain invalid values such as dangling pointers created by `delete` or pointers fabricated by illegal casting. This definition requires collectors to check the validity of every pointer discovered in a variable or object as the collector traces out all live objects. Also, invalid pointers may cause excess storage to be retained by the collector if they happen to point at storage reused for collected objects.

Some have suggested a stronger requirement, that all pointer-valued variables, members, elements, and expressions should evaluate to valid pointers to allocated objects. Most previous garbage-collected languages have required this stronger pointer validity. Pointer validity has some appeal, since if collectors could assume every pointer followed is valid, they might avoid some validity checks and the structures needed to support them, and invalid pointers couldn't accidentally retain excess storage.

There are several counter-arguments to the stronger requirement of pointer validity. First, pointer validity won't in fact save the overhead of validity checks. As we argue in section 2.2, a practical language proposal must allow for arbitrary interior pointers with no special type declarations, and it cannot use type declarations to distinguish between collected and non-collected pointer values. The data structures needed to handle interior pointers and to distinguish between collected and non-collected objects suffice for checking pointer validity, and the validity checks will cost at most an extra instruction per pointer followed (compared to the ten or more instructions needed to map interior pointers to base pointers—see section 10.1).

Second, pointer validity won't cause less excess storage to be retained. Regardless of whether pointer validity is required, programmers must null out all pointers to unused collected objects to ensure the objects will be freed. If the programmer forgets to null out pointers, excess storage will be retained.

Third, pointer validity would prohibit coexistence with a fair amount of existing C++ code that leaves large numbers of invalid pointers lying around, unused. For example, some libraries use overloaded `new` and `delete` to deallocate a huge collection of objects simultaneously, thus creating large numbers of dangling pointers that are never subsequently dereferenced. As another example, low-level code often creates pointers that don't appear to reference allocated objects.

Fourth, pointer validity requires the argument to `delete` be the last remaining pointer to the object being deleted. Maintaining this property would complicate destructors for non-collected circularly linked structures, requiring re-engineering of existing code and awkward constructions in future code.

Dangling pointers to collected objects wouldn't be created if `delete` didn't allow the storage occupied by collected objects to be reused immediately. However, we think garbage collection will be much more palatable to product engineers if they have the option of optimizing resource-critical parts of their systems via explicit `delete`'s that free storage immediately (see section 4.2).

6.4. Type validity

Our definition of GC-safety allows a collected pointer of type `T*` to be stored in a variable of another type `U*`, even if `T` is not derived from `U`. GC-safety requires only that collected pointers be stored in variables, members, and elements declared to have some pointer type.

This weak type validity imposes one requirement on collectors: collected storage returned by `new` must always be aligned to the largest possible alignment of any type (typically 32- or 64-bit alignment), regardless of the actual type being allocated. This requirement ensures that if an