Transparent Garbage Collection for C++ (Revised)
Document Number: N2129=06-0199

Hans Boehm, HP Labs
Mike Spertus, Symantec Research Labs

Symantec Research Labs

symantec.

Agenda—Goals

- Garbage collection must be available
- Garbage collection must be optional
- Garbage collection should be transparent, generally requiring no code changes
- Optional garbage collection granularity
- The programmer must be able to indicate type-safety
- The programmer must not be required to indicate type-safety
- Garbage collection requires standardization

Agenda—Proposal

- Reachability
- Syntax
- Impact on `operator new()`
- Finalization split off into separate proposal
- Implementation status
- Open questions

Garbage collection must be available

- The availability of garbage collections makes most programs much easier and attractive to implement with no negatives.
  - Vanilla C++ programs should be able to ignore memory management when not critical
- C++ is now increasingly ruled out as an implementation language for the many programs and developers that do not require manual memory management.
- Even for manually managed programs, legitimizes leak detectors
- Reference counting not sufficient
  - Too many data structures are not DAGs
  - Extensive programmer-support required for smart pointers

Garbage Collection must be optional

- The availability of manual memory management makes many large and specialized programs possible to implement.
  - Low-level systems programming
  - Programs that make heavy use of virtual memory
  - Programs with specialized performance requirements,
- Backwards compatibility. Although it might be technically conforming to turn “operator delete” into a “no-op,” the performance profile of some existing programs would experience unacceptable changes

Transparent garbage collection

- While smart-pointers are useful in the context of manually managed programs, they are not suitable for programs that wish to ignore memory management entirely.
- It should be possible to garbage collect most existing programs with no source changes at all, except for perhaps a single line per program (not per-module) to request automatic memory management.

Granularity

- Garbage collection vs. manual memory management should be specifiable at any level of granularity
  - Program level
  - Module level
  - Specific data types
  - Specific objects

Must be able to specify type-safety information

- Fully conservative (i.e., does not assume type safety) collection not suitable for very large programs
  - Large programs may consume a high-percentage of (32-bit) address space, causing unused objects to be retained.
  - Programs manipulating large pointer-sparse data structures (e.g., mpeg files) are common.
    - Scanning these for pointers is time consuming
    - Scanning these for pointers can cause disk thrashing
    - Scanning these for pointers can cause unused objects to be retained

Must not be required to specify type-safety

- Some programs are not type-safe
  - Should still work all right by default
  - Typical programmers should not need to worry about annotations
- The vast majority of vanilla programs do not require asserting type-safety for good results
- If libraries (e.g., standard libraries) are annotated, even very large programs should automatically get the benefit of type-aware garbage collection without any programmer input required

Standardization is required

- GC libraries have been used for many years, but…
  - Can’t access type information
  - Library vendors (including standard libraries) can’t use
  - Many users waiting for stamp of approval
  - Most people believe that C++ is not an option if they don’t want to manually manage memory

symantec.

Reachability

- An object is reachable if it is accessible via a pointer chain from the “roots”. Interior pointers are allowed (e.g., to support multiple inheritance).
- Strict reachability
  - Only consider pointer types.
  - Don’t consider type of pointer to avoid problems with `void *`, inheritance, etc.
  - Unions are based on last store
- Relaxed reachability
  - Pointers may be stored in any datatype large enough to hold them
    - E.g., Windows programmers frequently store pointers in DWORDs
- Compilers must not break reachability
  - See Boehm, “Simple Garbage-Collector Safety”

Syntax

- `gc_forbidden`
  - This code cannot be used in garbage collected programs
- `gc_required`
  - This code assumes the presence of a garbage collector
  - A diagnostic is required if this is combined with `gc(forbidden)` code (possibly at link time).
- In the absence of `gc_forbidden` or `gc_required`, the code is compatible with either the presence or absence of garbage collection

Type information syntax

- `gc_strict`
  - All occurrences of primitive non-pointer types are assumed not to contain pointers.
  - Collectors may make use of this information but are not required to.
- `gc_relaxed`
  - Primitive non-pointer types here may contain pointers
  - The default
- If alignment added to the standard, will add an additional one
  - Current proposal assumes natural alignment for pointers

symantec.

Some examples

- Program that assumes garbage collection
  - `gc_required`

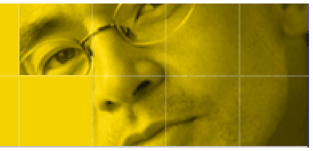
    ```
    main()
    …
    ```
  - Nothing else necessary. No need to free memory
- Modularity is good
  - ```
    gc_strict class A {
        A *next;
        B b;
        int data[1000000];
    };
    ```
  - Scan `next` and b for pointers, but no need to scan `data`.
  - This is even true for A objects created in non-strict code (because such code would explicitly refer to class A, not `int[1000000]`).
    ```

Some examples—Continued

- ```
  class mpeg {
      gc_strict mpeg(size_t s) {
          mpegData = new char[s];
      }
      ...
      char *mpegData;
  };
  ```
- `mpeg` class can be used anywhere without unnecessarily scanning `mpegData` for pointers.
- ```
  gc_strict {
     typedef int binop;
     ...
  }
  ```
- `binop` cannot contain a pointer.

Impact on `operator new`

- Allocation of garbage collected objects will not go through `operator new`
  - Many garbage collector are inextricably linked to allocation
  - `operator new` signature not sufficient for effective communication of type information
- Programs that redefine `::operator new` will work but will not benefit from garbage collection
- Classes with class-specific allocators will work but will not garbage collected
  - Their memory will still be scanned for pointers (respecting strictness annotations)
  - The underlying pools may be garbage collected as a whole
  - STL containers will only be collected if they use the default allocator



Finalization proposal split off

- Finalization split into separate proposal
  - With or without finalization, GC remains very valuable
- Enough to talk about to merit separate discussion
  - Compiler optimizations commonly cause an object to become unreachable while resources released by the finalizers are still in the use, leading to premature finalization.
  - Requires annotation by the programmer on when it is safe to call finalizers.
  - Java has been bitten badly by this
  - Treating destructors as finalizers is not an option
    - e.g., Deadlocks/data corruption can result from synchronization context

Implementation status

- On track
- Conservative collectors are stable and mature and will probably be the choice for most early implementations.
  - Implementation risks are well-mitigated
  - However, we do not restrict the choice of algorithm
    - Moving collectors must maintain `std::less<T *>`, e.g., to avoid breaking `Set<T *>`.
- Expect to have a modified g++ to support front-end syntax by next meeting

Discussion