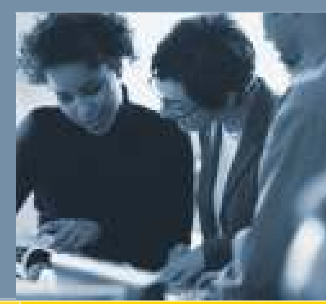
# Programmer Directed GC for C++



*Michael Spertus*

*N2286=07-0146*

*April 16, 2007*

# Garbage Collection

- Automatically deallocates memory of objects that are no longer in use.
- For many popular languages, garbage collection is the only way to reclaim memory
- Non-memory resources typically need to be released explicitly
- Has interesting tradeoffs with explicit memory management
  - Speed
  - Space
  - Latency
  - Virtual memory
  - etc.

# Garbage Collection for C++—Motivation

- For many data structures, object lifetime is difficult to manage statically
  - Some sort of dynamic technique is often required
- C++ is now increasingly ruled out as an implementation language for the many programs and developers that do not require manual memory management
  - Vanilla C++ programs should have the option of ignoring memory management when not critical
- Even for explicitly managed programs, accurately identifying leaked objects is valuable
  - Leak detectors
  - “Litter collection”
- C++ garbage collection technology is mature and ripe for standardization
  - Has been used extensively in a wide variety of scenarios for over a decade
  - Our proposal is closely tied to what has been shown by experience to work

# Optional garbage collection

- We definitely do not propose turning C++ into a pure garbage collected language
- Explicit memory management is critical for many classes of programs
  - Systems programming
  - Programs that make heavy use of virtual memory
  - Programs with specialized performance requirements
- Our proposal allows garbage collection to be freely mixed with explicit memory management

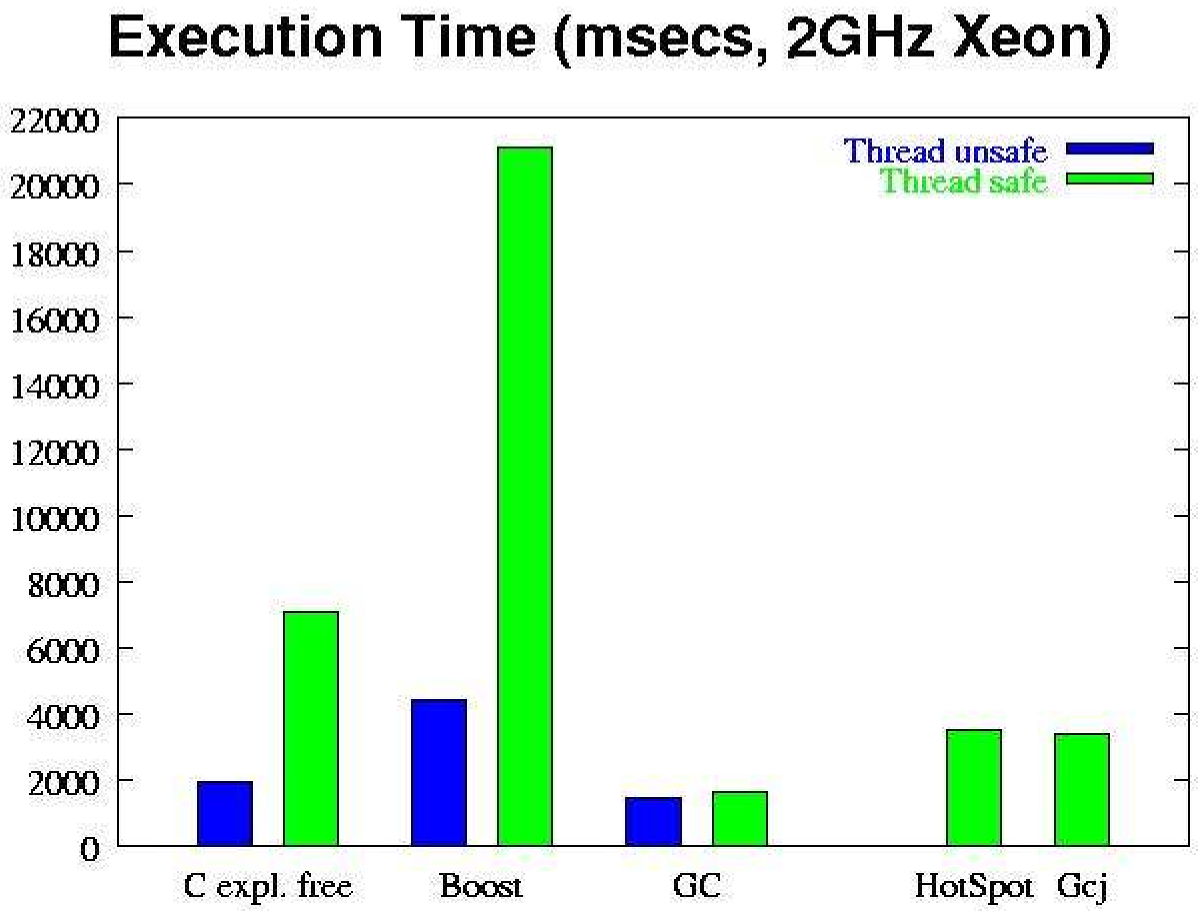
## Basic Use

- Source code changes are minimal
  - To use garbage collection, put "`gc_required;`" somewhere in your program
  - Existing object libraries can generally be used without recompilation
- If garbage collection is enabled, memory can be reclaimed either by explicit deletion or by the collector
  - Enabling garbage collection on an explicitly managed program is a no-op
  - …unless it has leaks, in which case the garbage collector can protect against memory leaks ("litter collection"). This has proven very useful in practice
    - As a example, one telco had a multi-million line executable that leaked memory on a large switch, requiring a reboot every hour. This program used 200 threads and 500MB heap. After enabling litter collection, the program was able to run indefinitely

# Comparison to shared_ptr

- Complements reference counted smart pointers
- Advantages
  - Speed
  - Can reclaim data structures that aren't DAGs (reference counting fails to reclaim cycles)
  - Interoperability: can reuse the billions of lines of existing C/C++ code
  - Suitable for programming in a pure GC-style on a par with any existing garbage collected languages
  - Can litter collect
  - Easy interoperation with explicit deletion
  - Easier migration from explicitly managed code
  - Avoids some problems with destructors

## Shared_ptr performance comparison



## Single-heap model

- (Almost) all memory is subject to reclamation by either explicit deletion or garbage collection
- We don’t allow restriction of garbage collection to particular types or objects
  - This effect can be achieved by explicitly deleting other classes
  - If you designated a class like the following (but no others) as subject to garbage collection
    `class A { vector<A *> v; };`
    data allocated by the `A::v` would be leaked because `vector<>` would allocate non-garbage collected memory
  - Passing pointers from one component to another quickly becomes confusing
  - Not very friendly to generics
  - May be best handled with a separate pointer type (e.g. shared_ptr or C++/CLI)

# What about non-memory resources?

- Not reclaimed by garbage collection
- Although significant, this has not proven a showstopper in other garbage collected languages and is typically less of an issue in C++ garbage collection due to the wealth of explicit management options (e.g., see next slide)
- We are considering annotations to help detect if a class is modified to use a non-memory resource
  - This should be thought of as an opportunity to provide better GC than in other languages
- Could also be handled by finalization

## When to use `shared_ptr`?

- If you only need to automatically manage a few types or data structures
  - `shared_ptr` has cost proportional to the amount of automatically managed memory, while GC cost is proportional to total memory
- If you need to manage objects that control non-memory resources
- If you need prompt deletion
- You have strict latency requirements (although watch out for destructor cascades)
- Virtual memory performance is important
- Large objects
- Bottom-line—Like we said before, `shared_ptr` complements GC
- Can be used together

## Why standardize?

- Access to the type system is often required for high-quality garbage collection. This has often proven a limiting factor in practice.
- Need to proscribe GC-unsafe optimizations (Such optimizations threaten leak detectors as well).
  - Not really an issue yet, but could definitely change
- Allow components to more easily share automatically managed objects
- Many users require stamp of approval and believe that C++ is not an option if they don't wish to explicitly manage memory

## The proposal

- Hews closely to existing practice
  - “No untried functionality”
  - Use existing practice for both garbage collection and litter collection
- Implementable with simple syntactic sugar on existing engines
- Defines reachability
- Main syntactic change is annotations two ideas:
  - Whether this translation unit requires, forbids, or allows garbage collection
  - Whether a region of code stores pointers in non-pointer types, allowing accurate (as opposed to conservative) collection.
- A small number of APIs

# Permitting or forbidding collection

- `gc_safe`—This translation unit works in either garbage collected or explicitly managed programs. All standard libraries are required to be `gc_safe`; This is the default.
  - Using GC on existing unannotated libraries is very useful (e.g., litter collection)
  - Experience shows failures more likely from changing allocators than adding GC.
- `gc_required`—This translation unit relies on a garbage collector to reclaim some objects
- `gc_forbidden`—This translation unit cannot be used in garbage collected programs. (E.g., it hides pointers with the xor-trick)

```
Example:
    gc_required;
    // Remaining code as normal
```

- Program will fail to compile/link with inconsistent declarations

## Reachability annotations

- `gc_strict`—The annotated code is type-safe (Specifically, it does not store pointers in primitive non-pointer types.)
- `gc_relaxed`—The annotated code may store pointers in non-pointer types (e.g. storing a pointer in an integer). This is the default. For relaxed code, the collector needs to conservatively scan all primitive datatypes for pointers.

Example:

```
gc_strict {
    // Type-safe code here
    // (Typically entire program)
}
```

- Even explicitly managed programs can benefit from reachability annotations (e.g., memory diagnostic tool output should improve).

# A few more examples

- Can annotate on a finer grain if necessary

```
gc_strict class A {
  A *next;
  B b;
  int data[1000000]; // Won't contain pointers
};
```

- Note that we don't know whether `b` is strict or relaxed. That is determined where `B` was defined. This (properly) eliminates the need for non-local knowledge. Just look at the explicitly-mentioned primitive types in the annotated code.

symantec.

## APIs

- `bool std::is_garbage_collected()`
- To allocate memory that will not be subject to GC, even in a garbage collected program (This allows you to do the XOR-trick even in a garbage collected program)
  - `new(std::nogc)`
  - `nogc_allocator.allocate()`
  - `nogc_malloc()`
- `std::gc_disable_gc()/gc_enable_gc()`—Temporarily prevent garbage collection (Think of as a “critical section”).
- `bool std::gc_collect();` Now would be a good time to GC
- `std::gc_add_root();`

# What about finalization?

- Broken off into separate proposal
- 80-20 rule
  - Almost all of the value of GC is still retained without finalizers
  - Almost all of the complication comes from finalizers
- The primary complication comes from interaction with the optimizer. Basically, dead variable elimination can cause an object’s memory may become unreachable while non-memory resources managed by the object are still in use.
  - In other languages, this can result in intermittent errors that won’t be caught in QA
  - But there are some options
- Still, there are good use cases worth considering
  - E.g., distributed reference counts, weak hash tables, diagnostics for collecting objects, managing non-memory resources
- Decoupling proposals have some benefit
  - Sufficiently independent to avoid delaying/muddying GC proposal
  - Experience with standardized GC could help shape an appropriate finalization approach in the future

## Impact on `operator new()`

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

## Comparison to other language GC

- Hews closely to traditional C++ GC, where there is a lot of experience with this model
- Objects subject to explicit deletion as well as garbage collection
- C++ programs typically generate far less garbage than Java programs due to large amount of stack allocation and explicit deletion
  - Collection cycles can be much less frequent.
  - Sometimes only a few per hour, reducing GC overhead to <1%

# Implementation Status

- Underlying technology mature with over a decade of industrial use for both pure garbage collection, mixed model, and litter collection
- Reference implementation based on g++ 4.1.2 in process. Will be complete for July meeting
- Reference implementation will include standardese
- Reference implementation will improve a “best practices” programming guide

## Some Concerns

- Non-memory resources (as above)
- Making all objects subject to GC and entire heap subject to scanning (as above) as contrasted with a desire to just collect individual classes
- Risk of desired libraries being `gc_forbidden` or `gc_required` and therefore incompatible with a particular application
  - Nearly all libraries expected to be `gc_safe` for this reason
  - Similar to the situation with threads
- Difficulties in validating whether unannotated legacy libraries are really safe for collection
  - Similar to the situation with memory models
- To what extent should root sets be standardized?
- Lack of finalization (as above)
- Performance profile (e.g., VM)
- Enabling/disabling GC at link and run-time
- Lack of experience with the reachability annotations
- Distinguishing “real” leaks from expected collection

# Process status

- Voted into registration standard
- Recently received considerable discussion on the reflector describing concerns and questions of some committee members, such as those listed above
- We remain comfortable that standardizing GC along the main lines of the proposal in the C++09 timeline is both appropriate and beneficial
- Meaningful time at this week’s standards meeting will be devoted to these questions