Minimal Support for Garbage Collection and Reachability-Based Leak Detection

WG21/N2585 = J16/08-0095

2008-03-16

Hans Boehm

Mike Spertus

Goal

- Support
  - Conservative Garbage Collection
  - Reachability-based leak detection
    - Which becomes more critical with `quick_exit()`
- By
  - Giving undefined behavior to programs that hide pointers.
  - Providing a small API to “unhide” pointers.
  - Providing a small API to make collection less conservative.

Charter

- Kona motion 1:

  WG21 Resolves that for this revision of the C++ standard (aka "C++0x") the scope of the memory management extensions shall be constrained as follows:

  - Include making some uses of disguised pointers undefined, and providing a small set of functions to exempt specific objects from this restriction and to designate pointer-free regions of memory (where these functions would have trivial implementations in a non-collected conforming implementation).
  - Exclude explicit syntax or functions for garbage collection or related features such as finalization.

Hiding pointers

- The real issue is dereferencing previously disguised pointers:

```
T *p = new ...;
intptr_t x = (intptr_t)(p) ^ 0x555;
a: T *q = (T *)(x ^ 0x555);
T y = *q;
```

- *p is reachable everywhere.
- But if collection occurs at a, *p may be reclaimed, since p is dead.

Hidden pointers

- Proposed wording classifies pointers as
  - Reconstituted, or
  - Safely derived
- This is a property of how the pointer is computed, not the bit representation of the pointer.
  - In the preceding example:
    - p is safely derived.
    - q is reconstituted
    - they are likely to be bitwise identical

New constraint

- Reconstituted pointers may not be dereferenced.
- More precisely:

  A pointer to storage obtained from an allocation function shall be dereferenced or passed to a deallocation function only if it was either safely derived, or the referenced object was previously *declared reachable* (see [library:declare_reachable])

This does not:

- Preserve correctness of all current C++ programs.
  - We really can’t if we want to move usefully closer to GC support.
  - Code that encodes pointers either has to break or leak.
- Require GC support in the implementation.
  - Vendors can trivially provide implementations that conform to the standard and don’t break old code.

Issues

- Where can we store pointers without making them “reconstituted”?
  - Currently in T*, `intptr_t`, and sufficiently aligned sections of `char` arrays.
- Might it be OK to dereference a reconstituted pointer if a safely derived pointer is stored in a non-stack location?
  - Eliminates need for part of API, but has optimization consequences for GC-based implementations.
  - We’re leaning against.

Issues contd.

- Do the rules apply to malloc’ed memory, as opposed to just memory allocated with default operator new (and the default STL allocator)?
  - Pro: More useful.
  - Con: Low-level OS code sometimes hides pointers. Would need fixing for GC implementation.
  - Con: Arguably infringes on WG14 territory.
  - Currently: No.

“Unhiding” API

```
void declare_reachable( void* p )
  throw(std::bad_alloc)
```

- p is a safely derived pointer.
- Allows reconstituted copies of p to be dereferenced.

```
template < typename T > T*
  undeclare_reachable( T* p ) throw()
```

- Undoes declare_reachable.
- Returns safely derived pointer.

Intended usage

- Calls bracket code that hides pointers.
- E.g. `declare_reachable()` before inserting node into xor-list, `undeclare_reachable()` on removal.
- Note that we need a safely derived pointer to the node after removal.

- Implementation:
  - – Insert into global/thread-local multiset.

Issues

- `void *` vs. template: inconsistency is ugly.
- Should undeclare_reachable return safely derived pointer, or make argument safely derived?
- We allow non-heap pointers, disallow null. Is this right?

Pointer-location API

```
void declare_no_pointers( char* p,
  size_t n ) throw()
```

– Declares [p, p+n) to contain no pointers. (Pointers stored there become reconstituted.)

```
void undeclare_no_pointers( char* p,
  size_t n ) throw()
```

– Undoes the effect of the above call. Arguments must match exactly. Calls on the same arguments don’t nest.

`Declare_no_pointers()` purpose

- Prevent the collector from needlessly tracing data known to not contain pointers.
  - Can significantly reduce extra memory retention by conservative collector
    - Especially in dense address spaces.
  - Can sometimes substantially reduce tracing time.

Implementation: A bit tricky, but we believe we can get it to a dozen or so memory operations for small regions.

Issues

- Combined `declare_no_pointers()` + `operator new` call?
  - More efficiently implementable.
  - Supported by existing collectors.
  - Con: Widens API.
- What’s the lifetime of a `declare_no_pointers()` call?
  - Currently until inverse call or object collection.