Doc No.: Pxxxx
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Project: Programming Language C++
Audience: Evolution
Date: 2019-04-10

Proposal of Handles

Introduction

We propose the addition to C++ of a new kind of entity called *handles*. Handles have similarities to classes, pointers and references. Handles have copy-by-reference semantics, and may or may not be destroyed automatically by the implementation at an unspecified time after they become unreachable.

Motivating Example

Suppose we are creating a random dungeon generator.

Let us define a *dungeon* as a set of *rooms*, where some pairs of rooms are directly connected by *doors*, and there is a path between every pair of rooms (that is, there is a series of doors we could walk through to get from any room A to any other room B in the dungeon).

We want to generate random such dungeons. The algorithm we will use is:

1. We start with an NxN grid.
2. Each cell is assigned randomly as either a room or impassible with some probability P.
3. The starting room is (N/2,N/2), which is always assigned as a room.
4. There is a door between each pair of rooms that share a side.

Let's start with the complete code and then walk through it:

```
struct auto Room {
    Room(size_t id) : id(id) {}

    // a unique id number for this room
    size_t id;

    // Door i of this room leads to the Room doors[i]
    std::vector<Room> doors;
```

```
};

Room generate_dungeon(size_t N, float P) {
    std::mt19937 gen(std::random_device{});
    std::uniform_real_distribution<float> dis;

    // STEP 1
    std::vector<Room> rooms_mat(N*N);
    auto rooms = [&](size_t x, size_t y) -> Room& {
       return rooms_mat[x*N+y];
    };

    size_t next_id = 1;

    // STEP 2
    for (auto& row : rooms)
        for (auto& cell : row)
            if (dis(gen) < P)
                cell = new Room(next_id++);

    // STEP 3
    rooms(N/2,N/2) = new Room(0);

    // STEP 4
    auto connect_rooms = [](Room a, Room b) {
        if (a != nullhnd && b != nullhnd) {
           a.doors.push_back(b);
           b.doors.push_back(a);
        }
    };

    for (size_t x = 0; x < N-1; x++)
        for (size_t y = 0; y < N; y++)
           connect_rooms(rooms(x,y), rooms(x+1,y));

    for (size_t x = 0; x < N; x++)
        for (size_t y = 0; y < N-1; y++)
           connect_rooms(rooms(x,y), rooms(x,y+1));

    return rooms(N/2,N/2);
}

int main() {
```

```
    Room current_room = generate_dungeon(100, 0.7);

    std::mt19937 gen(std::random_device{});
    std::uniform_real_distribution<size_t> dis;

    for (size_t i = 0; i < 1000; i++) {
        size_t door = dis(gen) % current_room.doors.size();
        Room next_room = current_room.doors[door];
        std::cout << "Moving from room "
                  << current_room.id
                  << " to room "
                  << next_room.id
                  << std::endl;
        current_room = next_room;
    }
}
```

Let's work through the interesting lines:

```
struct auto Room {
```

Starting a class definition with `class auto` or `struct auto`, declares a handle type. A handle definition actually introduces a pair of related types. There is the handle type which is given the name (Room). Then there is the payload type, which is an anonymous class type (like how a lambda generates an anonymous class type). Handle types have similarities to pointer and reference types. In particular a handle types default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator and destructor have built-in semantics (like pointers and reference types) and are not overloadable.

```
    size_t id;
```

Here we see a normal member of Room. Like a reference type, a Room handle can be used with the dot operator to refer to its members. So given a Room `r`, we can refer to this member with `r.id`.

```
    std::vector<Room> doors;
```

The standard containers are specialized for handle types in such a way that the ownership relationship between the elements of doors and the enclosing handle type is visible to the implementation. We will see why this is important later.

```
    std::vector<Room> rooms_mat(N*N);
```

Here we see a std::vector<Room> initialized with a size_t which causes the default constructor of Room to be called N*N times. Notice that the class definition of Room does not have a default constructor, and even if it did, it would not be called here. The default constructor of Room is the built-in handle default constructor, which causes a handle to be initialized to a special value called a null handle.

```
    for (auto& row : rooms)
        for (auto& cell : row)
            if (dis(gen) < P)
              cell = new Room(next_id++);
```

Here we see a new expression of handle type. A new expression of handle type is special. In particular it is the only way to call the payload constructors that are declared in the class definition. A new expression of a handle type returns an object (by value) of handle type (rather than a pointer to the handle type). Here the new expression is calling the Room(size_t) constructor, and returning a Room handle. This Room handle is then assigned to cell (which is a reference to a handle in the std::vector<Room> rooms). The assignment is the built-in handle move assignment operator and overwrites the null handle with the newly created one.

```
    auto connect_rooms = [](Room a, Room b) {
        if (a != nullhnd && b != nullhnd) {
           a.doors.push_back(b);
           b.doors.push_back(a);
        }
    };
```

Here we see a new keyword `nullhnd`. There is likewise a type called `std::nullhnd_t`. The relationship and built-in operations between nullptr, std::nullptr_t and pointer types, is the same as the relationship between nullhnd, std::nullhnd_t and handle types.

```
    return rooms(N/2,N/2);
}
```

Here is where the real magic happens. Notice we are returning just one element of the std::vector<Room>. The local std::vector<Room> variable is destroyed as the function returns, destroying the handles it contains. The returned room continues to be referenced by the return value, likewise the rooms it is connected to by its doors member, and recursively the rooms those are connected to, forming the complete dungeon. The rooms that are not connected via a path from the starting room (and hence not part of our generated dungeon) become unreachable. Note that there still exist handle objects to these unreachable rooms (from doors between rooms), however there is no longer a way to refer to them. At this point these unreachable rooms become available for destruction. Formally, they may (or may not) be destroyed by the implementation at some point after the vector is destroyed and before the end

of the program. In practice, a high quality implementation of handles will destroy them automatically shortly after the function returns.

Notice that if we had of used any of the current memory management mechanisms available in C++ it wouldn't have worked correctly. Notably std::shared_ptr would have created cycles between the unreachable rooms causing a permanent memory leak. std::unique_ptr and single ownership doesn't help because there are multiple references to rooms. A memory pool is likewise leaky. The only choice would be to manually manage objects by marking reachable rooms and then delete the unreachable ones - or to complicate the algorithm in other ways.

Furthermore, handles are more performant than std::shared_ptr. std::shared_ptr requires expensive atomic operations to mutate the reference count. Handles don't need that.

Background / Motivation

A basic understanding of automatic memory management algorithms is a necessary background for comprehension of this proposal. We recommend the following two wikipedia articles as a starting point:

https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

https://en.wikipedia.org/wiki/Tracing_garbage_collection

We will refer to the family of high-performance precise tracing garbage collection algorithms (or better), collectively, as *garbage collection*. Ie The kinds of garbage collection that every popular contemporary language uses (apart from C and C++). In particular we are not referring to std::shared_ptr reference counting or global conservative collectors as garbage collection.

The underlying motivation of this proposal is to add to C++ a way to use garbage collection alongside manual memory management. Garbage collection is a useful proven tool that, for some use cases, has well-understood advantages.

Design Goals

We worked under the following constraints:

1. BACKWARD COMPATIBLE: No breaking changes to C++
2. ZERO COST: The garbage collector shouldn't cost performance, if it isn't used.
3. CONVENIENT: The garbage collector should be available out of the box as part of the language, and enabled implicitly by syntactic use.
4. ENCAPSULATED: Garbage collected objects should be able to co-exist robustly and elegantly with manually allocated objects in the same program.

5. IMPLEMENTABLE: A minimally compliant garbage collector should be easy to implement.
6. IMPLEMENTOR FREEDOM: A high-quality compliant garbage collector should be possible to implement.
7. FAMILIAR: All things being equal, usage should be familiar to users of garbage collected languages.
8. SIMPLE: The garbage collector should be easy to use and intuitive.

Design Process

The best way to achieve the design goals is to introduce a new kind of entity, that has, where possible, similar semantics to a class type, but has differences, as needed, to accommodate the design goals. Objects of existing class types are not garbage collected. Objects of the new kind of class type are exclusively garbage collected. The class designer chooses, at class-design-time, whether objects of their class shall be garbage collected or not.

This is not the only possible design. One can imagine a design whereby a class can have some objects garbage-collected and other objects of the same type not. Such a design is inferior with respect to the design goals. In particular, design goals 4,5,6,7 and 8 - are all better served with the proposed design.

Existing standard and non-standard garbage collection systems (both library and core language) available to C++ have been studied. We claim the proposed design is superior as evaluated against the design goals.

Therefore, syntactically a way is added to mark class definitions according to this property:

```
class C { /* not garbage collected */ };
struct C { /* not garbage collected */ };
class auto C { /* garbage collected */ };
struct auto C { /* garbage collected */ };
[TODO: unions?]
```

The auto in this context is short for AUTOmatically memory managed.

As these marked types are exclusively garbage collected, we are able to encapsulate usage of objects of the type in such a way that they are only handled by their address on some logical garbage collected heap.

[Note: The logical graph that is formed by these garbage collected objects (vertices) and their addresses (edges), form the data structure over which the garbage collector operates. In order

to maintain the integrity of this graph, the exclusive handling by these addresses enables robust tracking of the graph]

Due to the desired exclusivity of usage by these addresses, we syntactically remove the other ways to manage objects of these types. In particular given a normal C++ class type C one can manage C by value (C), by reference (C&) or by pointer (C*). We don't want garbage collected objects to be handleable by any of these methods to preserve the integrity of the object graph. In order to achieve this we simply make the "real" class type anonymous and have the name C refer to the address type. In accordance with design goal 7 these are the same semantics that garbage collected languages uses. To distinguish these garbage collected addresses from pointers and references we coin a new name for them. To highlight that they have reference-semantics we call them *handles*.

Informal Specification

A handle definition of H:

```
class auto H { /*...*/ };
```

defines one name, and two types. An anonymous class type we call the payload type, and then the handle type, which has a built-in compound type "handle of __payload_H" where __payload_H is a name for exposition-only of the payload type. The name H names the handle type, not the payload type. [Note: There is no syntax, given a class type C, to form "handle to C". Likewise, given "handle to C", there is no way to extract C.]

The injected class name of the payload type is the handle type, not the payload type. However constructors and destructor of the payload type syntactically use the handle type name:

```
class auto H {
 public:
  H(); // default constructor of payload type
  H(H); // clone constructor of payload type
        // from an object of handle type
}
```

The expression `this` within a NSMF definition of a payload type is a const object of handle type.

The handle type is a built-in composite type that has the following basic functionality:

- Default-constructible to a null handle

- Copy-constructible from another handle
- Copy-assignable from another handle

When used in an expression of the form E1.E2, it is as if the handle is a reference to the payload. That is, it dispatches using the usual reference type semantics:

```
H h = /*...*/;
h.f(); // Calls function f from the handle definition of H
```

A new expression of a handle type constructs a new payload object and returns a handle to it:

```
H h1 = new H;  // calls payload default constructor
H h2(h1); // no new payload object, this is just a handle copy
H h3 = new H(h1); // creates a clone of h1 payload
                  // and returns handle to clone into h3
```

The handle type is equality comparable with std::nullhnd_t:

```
H h;
assert(h == nullhnd);
```

The default equality and comparison operators of a handle type are based on handle ordering, like pointers. These defaults can be overriden by payload-based ones in the handle definition. There is also standard library functions std::handle_equal(h1,h2) and std::handle_less(h1,h2) for cases where they are overridden but the underlying handle ordering is still required.

Each handle object is either null or *attached* to a payload object. A new expression creates a handle object that is attached to the new payload object:

```
H h = new H; // The handle object h is attached
             // to a newly created payload object
```

Handle copy construction and copy assignment makes a new handle object that is attached to the same payload object:

```
H h = new H; // 1 handle attached
H h2 = h; // 2 handles attached
H h3;
H3 = h; // 3 handles attached
```

The destructor of a handle detaches the handle from its payload:

```
{
```

```
    H h = new H; // 1 handle attached
    {
       H h2 = h; // 2 handles attached
    }
    // 1 handles attached
}
// 0 handles attached
```

We say: a payload object has zero or more handle objects attached to it.

Each object (any C++ object, not just payload object) *owns* zero or more handle object.

An object owns any handles that are direct or indirect subobjects of it:

```
struct auto H1 {};
struct S { H1 h; }
struct auto H2 : H1 {
    H1 h;
    H1 a[10];
    S s;
};

int main() {
    H2 h2 = new H2;
    // The h2 payload object owns 13 handles:
    //    1 from the base handle
    //    1 from h2.h
    //    10 from h2.a (h2.a[0], h2.a[1], ..., h2.a[9])
    //    1 from h2.s.h
}
```

In addition to the automatic subobject ownership relationship, when constructing a new handle object (not payload object) it is possible to explicitly create an ownership relationship to a containing object. This is for use in classes that use either normal new or placement new to create objects that they “contain”. For example std::vector, std::map, std::optional, std::unique_ptr. When the type T they are polymorphic on is a handle type (branched at compile-time), the equivalent new or placement new occurs on the handle type (not the payload type), and the new handle object is owned by the nominated object. This is achieved with special overloads of operator new for library authors of the form:

```
new (std::owned_handle_allocator, ...) T // where T is
                                         // a handle type
```

Let us define the *handle object graph* as a directed bipartite graph between handle objects and payload objects. There is an edge from a handle (source) to a payload (destination) if the handle is attached to the payload. There is an edge from a payload (source) to a handle (destination) if the payload object owns the handle (directly or indirectly, through subobjects or explicit ownership assignment). Notice that a handle can have at most one incoming edge.

For each handle that has no incoming edge, we call it a root handle. A root handle is one that is not owned by a payload object. That is, it is not a direct or indirect subobject of a payload object, and does not have an explicit ownership relationship with an object that is a direct or indirect subobject of a payload object.

A payload object is *reachable* if there exists a path in the graph beginning at a root handle and ending in the payload object.

A payload object that is not reachable is *unreachable*.

Unreachable payload objects may or may not be destroyed by the implementation. They may leak until the end of the program.

Unreachable payload objects that are destroyed by the implementation may be destroyed at anytime after they become unreachable, in any thread, with any sequencing or concurrently.

Appendix A: 2014 Discussion

This proposal was first informally proposed on std-proposals in 2014. Here is a transcript of the discussion and comments:

[std-proposals] Precise Per-Type Cyclic Garbage Collection (DRAFT 1)

43 messages

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 12:03 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Hey guys, this is a design I've been toying with (in the abstract for some time actually). It needs a bunch of work, but I would appreciate your feedback on this short draft. Also, if you are aware of any overlapping past proposals that would be great.

Thanks,
Andrew.

Precise Per-Type Cyclic Garbage Collection (DRAFT 1)

Introduction / Summary

We propose a core language feature that allows objects of user-selected class types to be cyclically garbage collected. Constraints on the usage of class types so selected, and pointers to such class types, are imposed to enable the implementation of fast safe precise collection.

Motivation

Many C++ programs can be decomposed into (a) low-level components for which the performance and timing control of manual memory management and value layout are of great benefit; and (b) higher-level organizational components for which the benefit is dominated and negligble - and the convenience of safe automatic cyclic garbage collection would be worth the tradeoff.

It would be great to be able to get the best of both worlds in one program. That is, to be able to specify certain classes as being garbage collected and others to be manually managed - and to only pay for what you use.

Comparison

With Boehm Demers Weiser Collector

The Boehm Collector can only be used either program-wide or not-at-all. What we propose isolates garbage collection only to certain user-selected types. Also, what we propose is *precise* garbage collection like in the managed languages, as opposed to *conservative* collection. The reachability graph is tracked explicitly through instrumenting the type system. That is, rather than scanning entire memory areas for all potential pointers to any dynamic memory, only the pointers to collected types are tracked - and they are tracked as they are initialized, assigned and destroyed. This gives it a radically different performance profile, and makes it proportional only to the use of collected types in the program, and not proportional to all dynamic memory use.

With shared_ptr<T>

Shared pointers cannot deal with cycles. weak_ptr often does not have a sensible place where it can be applied to break cycles, and when it does it is awkward to use. Shared pointer are also awkward to use with this through enable_shared_from_this. What we propose is much cleaner and easier to use, at the cost of the added implementation complexity of compiler support. Under the proposed feature, the user can just use regular pointer syntax to work with collected types, and doesn't need to worry about any of these issues.

With ownership and memory pools

Ownership schemes are many times artificial. In many object models, objects do not have natural owners, and it can be challenging to impose one. When an owner is selected the programmer must be careful to make sure the lifetime of the owner

encloses uses of the owned object. This implementation overhead and constraint is many times not worth the effort compared to automatic memory management.

Memory pools, which are just artificial owners, are not feasible in many long-running programs. In many such programs the memory pool can never be cleared, so they are no different than simply leaking into the heap and waste and exhaust memory.

Specification

Introduce a context-sensitive keyword **gc** that can appear in the head of a class specifier:

```
class foo gc
{
    ...
};
```

If a class type is marked with gc, it is called a *collected type*. A subclass of a collected type is also a collected type, whether or not marked with gc. A collected type may not have any base classes that are not also collected types. It follows that all base classes and all subclasses of a collected type are collected types.

An object of collected type is called a collected object. A collected object may only be a complete object or a base class subobject, it may not be a member subobject or an array element:

```
foo x[10]; // ill-formed
struct bar { foo x; } // ill-formed
```

You can use pointers instead:

```
foo* x[10]; // ok
struct bar { foo* x; } // ok
```

A collected type may only be allocated with dynamic storage duration. It may not be allocated with automatic, static or thread local storage duration. Again, you can use pointers instead:

```
auto s = new foo;

thread_local auto t = new foo;

void f()
{
    auto a = new foo;
}
```

An object of type pointer to a collected type, is called a collecting pointer. A collecting pointer cannot participate in pointer arithmetic. That is, there is no builtin meaning for addition, subtraction, increment or decrement of a collected pointer. It may also not be converted or cast to or from void*, and it may not be the subject or result of a reinterpret cast. It may only be initialized or assigned a null pointer constant or another collected pointer (possibly dynamic or static cast from a base class or subclass).

If a complete collected object is destroyed, any pointers to it or its base class subobjects are assigned the null pointer value by the implementation.

Given a time point at run-time of the program, we will describe a directed graph as follows. There is a root node. For each complete object of collected type there is a node. For each non-null collecting pointer that is not a member subobject of an object of collected type, there is an edge from the root

node to the complete object of the subject of the pointer. For each remaining non-null collecting pointer, there is an edge from the collected object of which it is a member to the complete object that is the subject of the pointer. If there is no path from the root node to a collected objects node, we say the collected object is unreachable.

The implementation shall automatically destroy collected objects, at some point between when it first was unreachable and the end of the program, or at the end of the program if they never become unreachable. (As a quality of implementation issue this should be as soon as reasonably possible given reasonable resources.)

Implementation

If a program contains a collected type, a garbage collector is linked into the program by the implementation. The constructor and destructor of both collected types and collecting pointers are generated to talk to the garbage collector. The garbage collector uses this information to track the graph. Periodically the garbage collector searches the graph using a generational or other garbage collection algorithm, deleting objects as appropriate.

Outstanding Issues

How do references to collected types work? References are much like constrained pointers so the specification of a reference to collected type would be similar to collecting pointers.

Are the restrictions on storage duration necessary? Couldn't collected objects of non-dynamic storage duration simply be ignored by the collector? What about the subobject restriction? It was initially felt

that this would simplify usage and make it safer - as well as easing implementation.

Is the assignment of null to pointers on delete necessary or helpful? Again this was a safety feature. We wanted collecting pointers if non-null to always be pointing to a collected object. If they are deleted manually, which would be unusual - we thought this would be because of destructor timing, and not resources - given that the performance profile of these high-level objects is most likely not paramount.

--

You received this message because you are subscribed to the Google Groups "ISO C++ Standard - Future Proposals" group.
To unsubscribe from this group and stop receiving emails from it, send an email to std-proposals+unsubscribe@isocpp.org.
To post to this group, send email to std-proposals@isocpp.org.
Visit this group at http://groups.google.com/a/isocpp.org/group/std-proposals/.

Andrew Sandoval <sandoval@netwaysglobal.com> Wed, Feb 12, 2014 at 12:43 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

[Quoted text hidden]

So, what is the point to having garbage collection in C++? My problem with gc in general is two fold: 1) It tends to laziness -- that is developers use it and then don't ever really understand object lifetime, etc. and 2) it is non-deterministic. The control over lifetime of an object is left up to the implementation of the collector -- not to the developer who should fully understand the required lifetime.

I'm also not sure you've made the case for why shared_ptr is insufficient. It, like unique_ptr, gives you exact control over the lifetime of pointers. They are tied to scope and reference counts as they are actually used. I may be slow, but I don't understand what you mean by "Shared pointers cannot deal with cycles." What cycles?

One of my complaints about managed languages is that everything becomes a pointer -- everything is made with new. Why bring that flaw into C++?

I also suspect that the idea of restricting reinterpret_cast is going to be problematic. One of the best things about C++ is that your hands are NOT tied -- if you want to leak memory, it is expected that you have a good reason for it. If you want to cast something to void *, it is again expected that you have a good reason for doing it. Everyone knows better than to reinterpret_cast without a very good reason, right?

Sorry to come down sounding negative, I personally just don't see the need and don't want C++ to pickup what I think are mistakes in managed languages.

Best wishes anyway.

[Quoted text hidden]

Bjorn Reese <breese@mail1.stofanet.dk> Wed, Feb 12, 2014 at 1:34 AM

To: andrewtomazos@gmail.com

> On 02/11/2014 03:03 PM, Andrew Tomazos
> wrote:
>
> > With shared_ptr<T>
> >
> > Shared pointers cannot deal with cycles.
> > weak_ptr often does not have a
> > sensible place where it can be applied to break
> > cycles, and when it does
> > it is awkward to use. Shared pointer are also
> > awkward to use with this
>
> This reminds me of a Boost proposal:
>
> https://svn.boost.org/svn/boost/sandbox/block_ptr/
> libs/smart_ptr/doc/index.html

Matthew Woehlke <mw_triad@users.sourceforge.net> Wed, Feb 12, 2014 at 3:12 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On 2014-02-11 09:43, Andrew Sandoval wrote:
> I'm also not sure you've made the case for why
> shared_ptr is
> insufficient. It, like unique_ptr, gives you exact control
> over the
> lifetime of pointers. They are tied to scope and
> reference counts as they
> are actually used. I may be slow, but I don't
> understand what you mean by "Shared
> pointers cannot deal with cycles." What cycles?

Let's say that A and B are classes that each keep a record of other objects referencing them... Now say we have:

```
pa = make_shared<A>();
pb = make_shared<b>();
// both pa and pb have refcount 1
pa->connect(pb);
// *pa now has an sptr ref to *pb, likewise *pb to *pa
// refcount of pa, pb is 2
```

Now if pa and pb go out of scope, the objects are inaccessible to the program but still reference each other, and so are not freed by shared_ptr. The above may be a contrived example, but that's the idea of a self-referential cycle. Any useful GC needs to be able to detect such cycles.

Basically, "reachable memory" is not a function of refcounts but of what memory is accessible, directly or indirectly, via all variables currently in scope.

--
Matthew
[Quoted text hidden]

Andrew Sandoval <sandoval@netwaysglobal.com> Wed, Feb 12, 2014 at 3:55 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org
Cc: mw_triad@users.sourceforge.net

[Quoted text hidden]

I still don't see it. if pa and pb go out of scope, the reference count should drop to zero. connect() should either have a shared_ptr at it's scope or use one at the class scope and either way when the destructor fires (at scope exit) the reference count should drop. If not I would think there is a design flaw.

More importantly, the whole thing can probably be simplified to start with so that one or both do not need to be dynamically created. I might be alone in this, but I find that I very rarely ever use new outside of singletons. Almost everything is best kept owned within a particular scope. On the rare occasions that new is needed, the results always go to a shared_ptr, either at local scope or in container at class scope, etc.

I'm not trying to pick on the idea, I just don't see it. And maybe I'm overstating this a little bit, but I really think that the languages that use GC have fundamental flaws that hurt the quality of code written with them. Way back when Java first came out, GC was the salvation from pointers. The problem is that they just made a mess of something that was never that bad to start with. So, instead of leaks you get crashes due to lifetime issues. It's a lot easier to catch a leak than to solve a free-after-use crash. Objective-C, C#, they all have that problem still, and they sort of ruin the whole meaning of scope. RAII is a much better way IMO.

-Andrew

[Quoted text hidden]

David Rodríguez Ibeas <dibeas@ieee.org> Wed, Feb 12, 2014 at 4:13 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

The problem, Andrew, is that if you build a data structure with a cycle, say a circular list, using shared_ptr the first node has a count of 2 (original pointer to the list, and last element in the list) while the rest of the nodes might have a count of 1. Now the list goes out of scope, and the external reference is dropped, the count of the head of the list drops from 2 to 1, but it cannot be released yet as the tail of the list still holds a valid shared_ptr. At this point each node in the list holds the next node alive even though the program cannot access any of the elements.
[Quoted text hidden]

Nevin Liber <nevin@eviloverlord.com> Wed, Feb 12, 2014 at 4:33 AM

Reply-To: std-proposals@isocpp.org
To: "std-proposals@isocpp.org" <std-proposals@isocpp.org>

On 11 February 2014 11:55, Andrew Sandoval <sandoval@netwaysglobal.com> wrote:

> I still don't see it. if pa and pb go out of scope, the reference count should drop to zero.

Here is a trivial example:

```
struct A {
        shared_ptr<A> p;
};

int main() {
        auto a = make_shared<A>();  // refcount == 1
        a->p = a;  // refcount == 2
}  // refcount == 1
```

The refcount never goes to zero, so you have a leak.

More details at <http://www.boost.org/doc/libs/1_55_0/libs/smart_ptr/shared_ptr.htm>.

While people do use GC as a crutch, there are some data structures (such as lock free) which are significantly easier to implement and reason about if you have GC. Check out <http://www.drdobbs.com/lock-free-data-structures/184401865> for a more in-depth description.

--
Nevin ":-)" Liber <mailto:nevin@eviloverlord.com> (847) 691-1404

[Quoted text hidden]

Klaim - Joël Lamotte <mjklaim@gmail.com> Wed, Feb 12, 2014 at 4:34 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Hi, here are a few questions about this proposal:

1. This proposal is about a language extension, but seems (if I didn't miss anything) to ignore C++11 minimal garbage collection hooks
as explained by Stroustrup there:
http://www.stroustrup.com/C++11FAQ.html#gc-abi
Did you take this into account?

2. Acronyms are harder to interpret, even in this case.
Instead of 'gc', I would suggest 'collected' (which is the adjective you use to describe what the keyword does to the type)

3. How do you expect generic algorithm developers to work with types which can't be manipulated through iterators/ranges?

4. Did you consider attaching the garbage collecting logic to specific instances instead of types?

[Quoted text hidden]

---

xavi <gratal@gmail.com> Wed, Feb 12, 2014 at 5:13 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals <std-proposals@isocpp.org>

My main concern is whether a language extension is really necessary or it could be implemented as a library. Is it possible to achieve a similar effect with something like boost::intrusive_ptr (which removes all the awkwardness of enable_shared_from_this), where all reference-counted objects inherit from a ref-counter class, and add some mechanism to detect cycles?
There might be certain things missing in the language:
  - Being able to forbid automatic storage for certain types.
  - Having some mechanism so that objects can only be created inside a smart pointer. Without inheritance, it's easy, by making the constructors private and make_ptr (or something similar) a friend. With inheritance things get much more complicated, so the language might need to be changed there.
  - Maybe tweak virtual inheritance so that it's possible to inherit from two ref-counted classes without significant overhead.

If such a solution is possible, the language changes to allow it will also be useful in other situations, and it will make it easy to develop custom garbage-collection mechanisms in user code.

2014-02-11 Klaim - Joël Lamotte <mjklaim@gmail.com>:
[Quoted text hidden]

[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 5:25 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Tuesday, February 11, 2014 7:34:52 PM UTC+1, Klaim - Joël Lamotte wrote:

> 1. This proposal is about a language extension, but seems (if I didn't miss anything) to ignore C++11 minimal garbage collection hooks
>
> as explained by Stroustrup there:
> http://www.stroustrup.com/C++11FAQ.html#gc-abi
>
> Did you take this into account?

Yes. The safely-derived pointer concept is designed for program-wide conservative collection such as the Boehm collector. This proposal is for nominating specific types for collection by a precise collector. Collecting pointers are even more constrained than safely-derived pointers, and they are better as the constraints on collecting pointers are enforced at compile-time.

> 2. Acronyms are harder to interpret, even in this case.
>
> Instead of 'gc', I would suggest 'collected' (which is the adjective you use to describe what the keyword does to the type)

Noted. To be considered.

> 3. How do you expect generic algorithm developers to work with types which can't be manipulated through iterators/ranges?

Integration with collections and algorithms needs some study, but the initial idea is that you use pointer to T rather than T itself. So for example:

```
vector<foo*> v = ...;

for (auto x : v)
    x->do_something();
```

Clearly this doesn't work easily with user-defined equality, hashing and comparison - this is no different than the situation today with pointers, unique_ptrs or shared_ptrs.

But having said that - since posting the proposal I have come up with a pretty radical idea of how to deal with this, but it is quite difficult to explain and I need to work through the ramifications. To try to summarize it (and fail), a collected type will be a "handle" type. It will store its data members within an unnamed struct, and it will have one "hidden" implicit member that is a pointer to that struct. That way, what we call in the original proposal a collecting pointer (foo*) will now be the collected type itself (foo), and what was the collected type (foo) is now an unnamed struct. Within its member functions, the this pointer is set to the implicit pointer member for looking up data members. So from within the class specifier it will look like you are defining a normal type and you can use it mostly as normal, but when you copy construct it, it will be a handle to the same instance. The collection graph is then traced the same way, except using these implicit pointer members and unnamed structs, rather than the collecting pointers and collected types. I have to double check that this isn't completely insane, and if you couldn't follow what I just said, I don't blame you.

> 4. Did you consider attaching the garbage collecting logic to specific instances instead of types?

Yes. It doesn't seem to work as well. If the entire type is nominated, things seem to work out much cleaner.

[Quoted text hidden]

Matthew Woehlke <mw_triad@users.sourceforge.net> Wed, Feb 12, 2014 at 5:54 AM

Reply-To: std-proposals@isocpp.org

To: std-proposals@isocpp.org

> On 2014-02-11 14:13, xavi wrote:
> My main concern is whether a language extension is
> really necessary or it
> could be implemented as a library.

I believe there are already libraries in the wild that do this. IIRC, VTK (http://vtk.org) is one...

See also http://www.aosabook.org/en/vtk.html and http://www.vtk.org/doc/release/5.10/html/classvtkGarbageCollector.html.

> There might be certain things missing in the language:
> - Being able to forbid automatic storage for certain
> types.
> - Having some mechanism so that objects can only
> be created inside a
> smart pointer. Without inheritance, it's easy, by making
> the constructors
> private and make_ptr (or something similar) a friend.

These days you probably just want to friend std::make_shared.

> With inheritance things get much more complicated, so
> the language
> might need to be changed there.

I'm not sure a technical solution to this problem is required. If someone wants to shoot themselves in the foot by bypassing a base class that is intended to only ever be constructed into a shared_ptr...

> - Maybe tweak virtual inheritance so that it's possible
> to inherit from
> two ref-counted classes without significant overhead.

Is this a problem in cases other than non-virtual inheritance from an intrusive pointer class? (Do I miss

why this would be an issue with plain old
std::shared_ptr?)

--
Matthew
[Quoted text hidden]

Andrew Sandoval <sandoval@netwaysglobal.com> Wed, Feb 12, 2014 at 6:13 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Tuesday, February 11, 2014 12:33:25 PM UTC-6, Nevin ":-)" Liber wrote:

> On 11 February 2014 11:55, Andrew Sandoval <sand...@netwaysglobal.com> wrote:
>
> I still don't see it. if pa and pb go out of scope, the reference count should drop to zero.
>
> Here is a trivial example:
>
> ```
> struct A {
>         shared_ptr<A> p;
> };
>
> int main() {
>         auto a = make_shared<A>();  // refcount == 1
>         a->p = a;  // refcount == 2
> }  // refcount == 1
> ```
>
> The refcount never goes to zero, so you have a leak.
>
> More details at <http://www.boost.org/doc/libs/1_55_0/libs/smart_ptr/shared_ptr.htm>.
>
> While people do use GC as a crutch, there are some data structures (such as lock free) which are

significantly easier to implement and reason about if you have GC. Check out <http://www.drdobbs.com/lock-free-data-structures/184401865> for a more in-depth description.

--

Nevin ":-)" Liber <mailto:ne...@eviloverlord.com> (847) 691-1404

Okay, I can see that. I just wonder if we can't find a way to solve the specific problem rather than add a language feature that is likely to be used as a crutch. Probably just prejudice on my part from seeing too many developers go after the quick and sloppy.

-Andrew

[Quoted text hidden]

---

xavi <gratal@gmail.com> Wed, Feb 12, 2014 at 6:21 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals <std-proposals@isocpp.org>

2014-02-11 Matthew Woehlke <mw_triad@users.sourceforge.net>:

> On 2014-02-11 14:13, xavi wrote:
>
> > My main concern is whether a language extension is really necessary or it could be implemented as a library.
>
> I believe there are already libraries in the wild that do this. IIRC, VTK (http://vtk.org) is one...
>
> See also http://www.aosabook.org/en/vtk.html and http://www.vtk.org/doc/release/5.10/html/classvtkGarbageCollector.html.

> > There might be certain things missing in the language:
> > - Being able to forbid automatic storage for certain types.
> > - Having some mechanism so that objects can only be created inside a smart pointer. Without inheritance, it's easy, by making the constructors private and make_ptr (or something similar) a friend.
>
> These days you probably just want to friend std::make_shared.

Intrusive reference-counting is more efficient, and makes a lot of sense for objects which are always supposed to be reference-counted. It also allows easily creating new "connected" shared

pointers from references, which seems like a reasonable thing to do. Also, befriending make_shared is a tricky issue, and it makes *all* your constructors effectively public.

> With inheritance things get much more complicated, so the language might need to be changed there.

> I'm not sure a technical solution to this problem is required. If someone wants to shoot themselves in the foot by bypassing a base class that is intended to only ever be constructed into a shared_ptr...

Every constructor would need to be protected, and then every derived class will need to make their own constructors protected and friend the pointer maker, and so on... it's easy to make mistakes, and there is some repetition.

> - Maybe tweak virtual inheritance so that it's possible to inherit from two ref-counted classes without significant overhead.

> Is this a problem in cases other than non-virtual inheritance from an intrusive pointer class? (Do I miss why this would be an issue with plain old std::shared_ptr?)

It is an issue the moment you want to enable_shared_from_this.

--

Matthew

--

--- You received this message because you are subscribed to the Google Groups "ISO C++ Standard - Future Proposals" group.
To unsubscribe from this group and stop receiving emails from it, send an email to std-proposals+unsubscribe@isocpp.org.
To post to this group, send email to std-proposals@isocpp.org.
Visit this group at http://groups.google.com/a/isocpp.org/group/std-proposals/.

[Quoted text hidden]

Dain Bray <dain.bray@gmail.com> Wed, Feb 12, 2014 at 6:26 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

What you are describing sounds similar to the way C++/CLI added managed classes. If you are not familiar with it, you might check that out.
I'm not sure if a GC is necessary, I find shared types are rare, and cyclic shared types rarer still--which weak ptr breaks well enough.. Seems like alot of complexity for little, if any gain. Perhaps this would be better as a library solution?

[Quoted text hidden]
[Quoted text hidden]

hun.nemethpeter@gmail.com
<hun.nemethpeter@gmail.com>

Wed, Feb 12, 2014 at 9:58 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

I think the real question here is can we detect shared_ptr cycles in compile time?

On Tuesday, February 11, 2014 7:33:25 PM UTC+1, Nevin ":-)" Liber wrote:

> On 11 February 2014 11:55, Andrew Sandoval <sand...@netwaysglobal.com> wrote:
>
> > I still don't see it. if pa and pb go out of scope, the reference count should drop to zero.
>
> Here is a trivial example:
>
> ```
> struct A {
>     shared_ptr<A> p;
> };
>
> int main() {
>     auto a = make_shared<A>();  // refcount == 1
>     a->p = a;  // refcount == 2
> }  // refcount == 1
> ```
>
> The refcount never goes to zero, so you have a leak.
>
> More details at <http://www.boost.org/doc/libs/1_55_0/libs/smart_ptr/shared_ptr.htm>.
>
> While people do use GC as a crutch, there are some data structures (such as lock free) which are significantly easier to implement and reason about if you have GC. Check out <http://www.drdobbs.com/lock-free-data-structures/184401865> for a more in-depth description.

--
Nevin ":-)" Liber <mailto:ne...@eviloverlord.com> (847) 691-1404
[Quoted text hidden]

Patrick Michael Niedzielski <patrickniedzielski@gmail.com>

Wed, Feb 12, 2014 at 10:52 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On mar, 2014-02-11 at 15:58 -0800, hun.nemethpeter@gmail.com wrote:
> > I think the real question here is can we detect shared_ptr cycles in
> > compile time?

Not in the general case, because you'd run into the halting problem. What a given shared_ptr<> points to is not known always known at compile-time. To know whether there is a shared_ptr cycle at any point during the duration of the program, you have to do static analysis on the program with all possible inputs. The problem with that,

though, is
that you can't know
whether a given program
will halt on some given
set
of inputs, so you can't
even guarantee that your
compilation will
finish.

(As a side note, the way
the standard deals with
the halting problem at
compile time probably
wouldn't work here. In
cases like template
recursion and
preprocessor macros,
which are often said to be
"Turing-complete", the
standard places an
implementation-defined
limit
on the maximum
recursion depth. This
makes them not
Turing-complete,
technically, although
that's generally not
important.)

In trivial cases, you may
be able to detect them,
but that won't help
much, and doesn't solve
the problem. It may be a
useful diagnostic,
though.

Cheers,
Patrick Niedzielski

Geoffrey Romer <gromer@google.com> Wed, Feb 12, 2014 at 10:59 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Tue, Feb 11, 2014 at 4:52 PM, Patrick Michael Niedzielski <patrickniedzielski@gmail.com> wrote:

> On mar, 2014-02-11 at 15:58 -0800, hun.nemethpeter@gmail.com wrote:
> > I think the real question here is can we detect shared_ptr cycles in
> > compile time?
>
> Not in the general case, because you'd run into the halting problem.
> What a given shared_ptr<> points to is not known always known at
> compile-time. To know whether there is a shared_ptr cycle at any point
> during the duration of the program, you have to do static analysis on
> the program with all possible inputs. The problem with that, though, is
> that you can't know whether a given program will halt on some given set
> of inputs, so you can't even guarantee that your compilation will
> finish.

Here's a simple example:

```
struct s {
  shared_ptr<s> ptr;
}

shared_ptr<s> f(int n) {
  static map<int, shared_ptr<s>> ptrs = {{1,
{nullptr}}};
  if (ptrs.find(n) == ptrs.end()) {
    if (n %2 == 0) {
      ptrs.emplace(n, f(n/2));
    } else {
```

```
      ptrs.emplace(n, f(3*n + 1));
    }
  }
  return ptrs[n];
}

int main() {
  int n;
  cin >> n;
  f(n);
  return 0;
}
```

If your compiler can tell you whether this program contains any reference cycles, it has just solved a problem that has defeated some of the world's greatest mathematicians, called the Collatz conjecture.

(As a side note, the way the standard deals with the halting problem at
compile time probably wouldn't work here.

There's no "probably" about it; this will not work.

In cases like template
recursion and preprocessor macros, which are often said to be
"Turing-complete", the standard places an implementation-defined limit
on the maximum recursion depth. This makes them not Turing-complete,
technically, although that's generally not important.)

In trivial cases, you may be able to detect them, but that won't help
much, and doesn't solve the problem. It may be a useful diagnostic,
though.

Cheers,
Patrick Niedzielski

[Quoted text hidden]

Patrick Michael Niedzielski <patrickniedzielski@gmail.com>

Wed, Feb 12, 2014 at 12:06 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On mar, 2014-02-11 at 16:59 -0800, Geoffrey Romer wrote:

> Here's a simple example:
>
> struct s {
> shared_ptr<s> ptr;
> }
>
> shared_ptr<s> f(int n) {
> static map<int, shared_ptr<s>> ptrs = {{1, {nullptr}}};
> if (ptrs.find(n) == ptrs.end()) {
> if (n %2 == 0) {
> ptrs.emplace(n, f(n/2));
> } else {
> ptrs.emplace(n, f(3*n + 1));
> }
> }
> return ptrs[n];
> }
>
> int main() {
> int n;
> cin >> n;
> f(n);
> return 0;
> }
>
> If your compiler can tell you whether this program contains any reference
> cycles, it has just solved a problem that has

> defeated some of the world's
> greatest mathematicians, called the Collatz conjecture.

You're right in that that's an example of an undecidable program, but there's a strategy to tell that this program will not have reference cycles. The ptr inside s can only point to nullptr, because it is only set during construction, and never changed. Every time you emplace, you are constructing a shared_ptr<s> based on another shared_ptr<s>. The only shared_ptr<s> that can originally be constructed from has a nullptr in its ptr member. Assuming std::map's emplace member function doesn't do any magic (which it shouldn't, for obvious reasons), all shared_ptr<s> in the map will point to the same object of type s, who has a null shared_ptr<s>. In other words, no reference cycles, found in a way that avoids the

halting problem
altogether.

That said, that's not an
easy thing to get a
compiler to do that, and
it's not worth it.
Furthermore, it doesn't
solve the problem in
general, so this still can't
be done.

> > (As a side note, the
way the standard deals
with the halting problem
at
> > compile time probably
wouldn't work here.
>
> There's no "probably"
about it; this will not work.

Okay, I should clarify.
For using a strategy I
hinted below your
response (i.e., doing what
template recursion and
preprocessor macros do
by placing an
implementation-defined
limit on the maximum
depth of the
construct), there is no
"probably" about, yes. It
will definitely
work. Limiting the
theoretical Turing
completeness of the
language with
an analogous

implementation-defined limit on recursion depth and looping count/depth would cause this to be solvable in O(n) time, based on the number of loops or recursive calls total (each loop/recursive function call could be checked in O(1) time, with a sufficiently large constant based on the implementation-defined limit).

The "probably" was a polite way of saying "this is obviously non-solution". At least, I hope it's obvious why.

Cheers,
Patrick

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 1:27 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On Tuesday, February 11, 2014 9:26:07 PM UTC+1, Dain Bray wrote:
> > What you are describing sounds similar to the way C++/CLI added managed classes. If you are not familiar with it, you might check that out.

I am familiar, thanks. It is similar in part, but has different goals. What Microsoft was trying to do was modify C++ so it could run on their VM and so use their VM libraries. A small part of that was precise garbage collection. What we are proposing here is some minimal clean additions purely to enable adding a precise garbage collector for a subset of user-nominated types.

> Perhaps this would be better as a library solution?

shared_ptr is basically as good as it gets as a pure library solution, and I compare the differences in the proposal. In any case, something as heavily-demanded as real garbage collection warrants core language additions if needed - and I think the "only pay for what you use" property of my proposal is the right approach.

[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 1:56 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org
Cc: hun.nemethpeter@gmail.com

On Wednesday, February 12, 2014 12:58:20 AM UTC+1, hun.nem...@gmail.com wrote:

> I think the real question here is can we detect shared_ptr cycles in compile time?

Garbage collection has been a topic of intense study in academia for decades and is on-going. If you are interested I would encourage you to read up on it. Section 7.5 to 7.8 in the dragon book 2nd edition has a good primer.

The important thing to understand is that to do better than shared_ptr, which cannot even detect cycles at run-time, much less compile-time - we need to be able to expose the full reachability graph to the garbage collection algorithm. A shared_ptr cannot see the whole graph - it can only see the inbound-side of edges, not the outbound-side. That is, each shared_ptr is an edge of the graph, and it can see who it is pointing to, but it doesn't know who it belongs to. Under the proposal the full graph is tracked by instrumenting the constructors of collected types and collecting pointers.

[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 2:36 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org
Cc: mw_triad@users.sourceforge.net

On Tuesday, February 11, 2014 8:54:42 PM UTC+1, Matthew Woehlke wrote:

> On 2014-02-11 14:13, xavi wrote:
> > My main concern is whether a language extension is really necessary or it
> > could be implemented as a library.
>
> I believe there are already libraries in the wild that do this. IIRC,
> VTK (http://vtk.org) is one...

Like in many of the native extension environments of managed languages and scripting languages, the VTK garbage collector works with the same general architecture as the proposal - however to register the outbound-side of edges you need to manually call a register function for each member collecting pointer that a collected type contains. From this information the full graph is formed.

It should be clear that such a system as a pure library solution is extremely awkward to use and unsafe. Under the proposal, this graph tracking is instrumented automatically by the compiler. Given the extremely high demand for this feature, it should be clear that a core language addition is warranted. If unconvinced by ease of use, than you should be at least be convinced by the compile-time safety aspect.

[Quoted text hidden]

---

hun.nemethpeter@gmail.com <hun.nemethpeter@gmail.com>

Wed, Feb 12, 2014 at 3:01 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Wednesday, February 12, 2014 1:59:29 AM UTC+1, Geoffrey Romer wrote:

> On Tue, Feb 11, 2014 at 4:52 PM, Patrick Michael Niedzielski <patrickni...@gmail.com> wrote:
>
> > On mar, 2014-02-11 at 15:58 -0800, hun.nem...@gmail.com wrote:
> > > I think the real question here is can we detect shared_ptr cycles in
> > > compile time?
> >
> > Not in the general case, because you'd run into the halting problem.
>
> And what about the non-general case?

```
struct s {
  shared_ptr<s> ptr;
}
```

This S struct refers to S as a shared_ptr.

Is it possible to create a safe struct pattern, where cycle is not possible? This is smaller goal then a generic one.

So my idea is introducing a new attribute, called [[cycle_free]] or [[acyclic]] or whatever that can be attached to a class.

so

```
[[cycle_free]]
struct S {
  shared_ptr<S> ptr;
};
```

will gives a warning because a cycle is possible with this.
And a cycle_free class only contains cycle_free ones.
[Quoted text hidden]

Thiago Macieira <thiago@macieira.org> Wed, Feb 12, 2014 at 3:02 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Em ter 11 fev 2014, às 20:36:57, Andrew Tomazos escreveu:
> It should be clear that such a system as a pure library solution is
> extremely awkward to use and unsafe. Under the proposal, this graph
> tracking is instrumented automatically by the compiler. Given the
> extremely high demand for this feature, it should be clear that a core
> language addition is warranted.

Can it wait for compile-time reflection support and simply use that to detect
which members are collecting pointers?

--
Thiago Macieira - thiago (AT) macieira.info - thiago (AT) kde.org
Software Architect - Intel Open Source Technology Center
PGP/GPG: 0x6EF45358; fingerprint:
E067 918B B660 DBD1 105C 966C 33F5 F005 6EF4 5358
[Quoted text hidden]

Geoffrey Romer <gromer@google.com> Wed, Feb 12, 2014 at 3:23 PM

Reply-To: std-proposals@isocpp.org

To: std-proposals@isocpp.org

On Tue, Feb 11, 2014 at 6:06 PM, Patrick Michael Niedzielski <patrickniedzielski@gmail.com> wrote:

> On mar, 2014-02-11 at 16:59 -0800, Geoffrey Romer wrote:
> > Here's a simple example:
> >
> > struct s {
> >   shared_ptr<s> ptr;
> > }
> >
> > shared_ptr<s> f(int n) {
> >   static map<int, shared_ptr<s>> ptrs = {{1, {nullptr}}};
> >   if (ptrs.find(n) == ptrs.end()) {
> >   if (n %2 == 0) {
> >   ptrs.emplace(n, f(n/2));
> >   } else {
> >   ptrs.emplace(n, f(3*n + 1));
> >   }
> >   }
> >   return ptrs[n];
> > }
> >
> > int main() {
> >   int n;
> >   cin >> n;
> >   f(n);
> >   return 0;
> > }
> >
> > If your compiler can tell you whether this program contains any reference
> > cycles, it has just solved a problem that has defeated some of the world's
> > greatest mathematicians, called the Collatz conjecture.

> You're right in that that's an example of an undecidable program,

Well, strictly speaking it's not known to be undecidable (and I wouldn't be surprised if it was decidable), it's just evidently _extremely hard_ to decide.

> but there's a strategy to tell that this program will not have reference cycles. The ptr inside s can only point to nullptr, because it is only set during construction, and never changed. Every time you emplace, you are constructing a shared_ptr<s> based on another shared_ptr<s>. The only shared_ptr<s> that can originally be constructed from has a nullptr in its ptr member. Assuming std::map's emplace member function doesn't do any magic (which it shouldn't, for obvious reasons), all shared_ptr<s> in the map will point to the same object of type s, who has a null shared_ptr<s>. In other words, no reference cycles, found in a way that avoids the halting problem altogether.

Argh, you're right, but that's a bug in my example, not a fundamental point. I think this fixes it:

```
shared_ptr<s> f(int n) {
  static map<int, shared_ptr<s>> ptrs = {{1, {nullptr}}};
  if (n != 1 && ptrs[n].ptr == nullptr) {
    if (n %2 == 0) {
      ptrs[n].ptr = f(n/2);
    } else {
      ptrs[n].ptr = f(3*n + 1);
    }
  }
  return ptrs[n];
```

}

The point of the code is that it produces a reference cycle if and only if the "hailstone sequence" starting with the input number contains a cycle other than (4, 2, 1), so determining at compile time if this code can produce a reference cycle requires deciding whether there exists a cycle other than (4, 2, 1) in the hailstone sequence of any number, which would be tantamount to solving the Collatz conjecture.

That said, that's not an easy thing to get a compiler to do that, and
it's not worth it. Furthermore, it doesn't solve the problem in
general, so this still can't be done.

> > (As a side note, the way the standard deals with the halting problem at
> > compile time probably wouldn't work here.
>
> There's no "probably" about it; this will not work.

Okay, I should clarify. For using a strategy I hinted below your
response (i.e., doing what template recursion and preprocessor macros do
by placing an implementation-defined limit on the maximum depth of the
construct), there is no "probably" about, yes. It *will definitely*
work. Limiting the theoretical Turing completeness of the language with
an analogous implementation-defined limit on recursion depth and looping
count/depth would cause this to be solvable in O(n) time, based on the
number of loops or recursive calls total (each loop/recursive function

call could be checked in O(1) time, with a
sufficiently large constant
based on the implementation-defined limit).

The "probably" was a polite way of saying "this is
obviously
non-solution". At least, I hope it's obvious why.

Cheers,
Patrick

[Quoted text hidden]

Geoffrey Romer <gromer@google.com> Wed, Feb 12, 2014 at 4:10 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Tue, Feb 11, 2014 at 9:01 PM,
<hun.nemethpeter@gmail.com> wrote:

> On Wednesday, February 12, 2014 1:59:29 AM
> UTC+1, Geoffrey Romer wrote:
>
> > On Tue, Feb 11, 2014 at 4:52 PM, Patrick
> > Michael Niedzielski <patrickni...@gmail.com>
> > wrote:
> >
> >
> >
> > > On mar, 2014-02-11 at 15:58 -0800,
> > > hun.nem...@gmail.com wrote:
> > > > I think the real question here is can we detect
> > > shared_ptr cycles in
> > > > compile time?
> > >
> > > Not in the general case, because you'd run into
> > > the halting problem.
> > And what about the non-general case?
> >
> >
> > struct s {
> > shared_ptr<s> ptr;
> > }
> >
> >
> > This S struct refers to S as a shared_ptr.
> >
> > Is it possible to create a safe struct pattern,
> > where cycle is not possible? This is smaller goal
> > then a generic one.
> >
> > So my idea is introducing a new attribute, called
> > [[cycle_free]] or [[acyclic]] or whatever that can
> > be attached to a class.
> >
> > so

```
[[cycle_free]]
struct S {
  shared_ptr<S> ptr;
};
```

will gives a warning because a cycle is possible with this. And a cycle_free class only contains cycle_free ones.

First, a meta point: the problem of cycles in reference-counting is very well-known, and has been studied by a lot of very smart people. That doesn't mean there are no good solutions left to be found, but it does mean that you should be very skeptical of any solution that didn't take a lot of effort to find, or that doesn't contain an identifiable deep insight that all those smart people could plausibly have missed.

As for this specific proposal, would shared_ptr have a [[cycle_free]] annotation? I don't believe there's any way to selectively annotate some instantiations of a template but not others, so you have to pick once and for all. If it doesn't have that annotation, then [[cycle_free]] types can't contain shared_ptrs, which makes the annotation trivial and completely useless. On the other hand, if it does have that annotation, you've basically destroyed the usefulness of shared_ptr: situations where you need objects to point to other objects of the same type are just too commonplace for this to be viable. For example, it's basically the only way to represent any sort of linked data structure, such as a list, tree, or graph.

Worse, reference cycles can involve reference types other than shared_ptr; any kind of ownership relationship can participate in a cycle, so you face the same dilemma about e.g. whether to annotate unique_ptr. You've also glossed over

how this interacts with separate compilation: when you compile something like

```
struct X {
  shared_ptr<Y> y;
  Z* z;
};
```

Y and Z may be incomplete types, in which case the compiler has no way of knowing if they're annotated or not. That might be solvable, but only by making [[cycle_free]] even more restrictive and useless.

--

You received this message because you are subscribed to the Google Groups "ISO C++ Standard - Future Proposals" group.
To unsubscribe from this group and stop receiving emails from it, send an email to std-proposals+unsubscribe@isocpp.org.
To post to this group, send email to std-proposals@isocpp.org.
Visit this group at http://groups.google.com/a/isocpp.org/group/std-proposals/.

[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Wed, Feb 12, 2014 at 4:54 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Wednesday, February 12, 2014 7:10:30 AM UTC+1, Geoffrey Romer wrote:

> First, a meta point: the problem of cycles in reference-counting is very well-known, and has been studied by a lot of very smart people. That doesn't mean there are no good solutions left to be found, but it does mean that you should be very skeptical of any solution that didn't take a lot of effort to find, or that doesn't contain an identifiable deep insight that all those smart people could plausibly have missed.

Actually, contrary to popular belief, and quite fascinatingly, reference counting garbage collection algorithms can breaking cycles quite easily. See this paper from our friends at IBM:

Concurrent Cycle Collection in Reference Counted Systems, David F. Bacon and V.T. Rajan

https://www.cs.purdue.edu/homes/hosking/690M/Bacon01Concurrent.pdf

In fact there are some that claim reference counting with such cycle-breaking is more performant than the trace / mark-and-sweep style algorithms - and there are some major VMs that are considering changing to it.

The problem for our purposes is, like tracing, these cycle-breaking reference counted algorithms need access to the full graph. shared_ptr doesn't give us the outbound-side of edges.

As for, can we detect potential cycles at compile-time? - who cares... We want to be able to have cycles, we just want them to be collected.

[Quoted text hidden]

hun.nemethpeter@gmail.com
<hun.nemethpeter@gmail.com>

Wed, Feb 12, 2014 at 6:09 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> As for this specific proposal, would shared_ptr have a [[cycle_free]] annotation?

Good question. Maybe we should mark it as a [[link]] that links a [[cycle_free]] struct.

> I don't believe there's any way to selectively annotate some instantiations of a template but not others, so you have to pick once and for all. If it doesn't have that annotation, then [[cycle_free]] types can't contain shared_ptrs, which makes the annotation trivial and completely useless. On the other hand, if it does have that annotation, you've basically destroyed the usefulness of shared_ptr: situations where you need objects to point to other objects of the same type are just too commonplace for this to be viable. For example, it's basically the only way to represent any sort of linked data structure, such as a list, tree, or graph.

I don't think so. For example a Html class that has a Header and Body, and Body has h1, ul and other elements... I think cycle is not possible in an Html document. Do we really need so generic list, tree graph data structures these days? They are basically list<T>, TreeNode<T>, GraphNode<T> nowadays and we should just state that GraphNode<T> can't link a GraphNode<T>.

> Worse, reference cycles can involve reference types other than shared_ptr; any kind of ownership relationship can participate in a cycle, so you face the same dilemma about e.g. whether to annotate unique_ptr. You've also glossed over how this interacts with separate compilation: when you compile something like

> struct X {
> shared_ptr<Y> y;
> Z* z;
> };
>
> Y and Z may be incomplete types, in which case the
> compiler has no way of knowing if they're annotated or not.
> That might be solvable, but only by making [[cycle_free]]
> even more restrictive and useless.

if
Y is struct Y { int a; };
and
Z is struct Z { char* a };
then this struct hierarchy is safe, isn't it? If this is safe we should extend this pattern to maximum level.

> This check can be performed after the compilation. In this
> case a central data file is generated, where every
> [[cycle_free]] and [[link]] is collected.
>
> This approach is start from a safe core pattern where cycle
> is not possible and should be carefully extended.
> [Quoted text hidden]

Geoffrey Romer <gromer@google.com> Thu, Feb 13, 2014 at 4:52 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Wed, Feb 12, 2014 at 12:09 AM,
<hun.nemethpeter@gmail.com> wrote:

> As for this specific proposal, would shared_ptr have a [[cycle_free]] annotation?
>
> Good question. Maybe we should mark it as a [[link]] that links a [[cycle_free]] struct.
>
> I don't believe there's any way to selectively annotate some instantiations of a template but not others, so you have to pick once and for all. If it doesn't have that annotation, then [[cycle_free]] types can't contain shared_ptrs, which makes the annotation trivial and completely useless. On the other hand, if it does have that annotation, you've basically destroyed the usefulness of shared_ptr: situations where you need objects to point to other objects of the same type are just too commonplace for this to be viable. For example, it's basically the only way to represent any sort of linked data structure, such as a list, tree, or graph.
>
> I don't think so. For example a Html class that has a Header and Body, and Body has h1, ul and other elements... I think cycle is not possible in an Html document.

A cycle is not possible in an HTML document, but that's because the nodes form a tree structure, and trees have no cycles. This is enforced by the grammatical structure of HTML as a context-free language, not by any kind of type system requirement on nodes. What you're proposing is different: you want the node *types* to form a strict hierarchy, with nodes not permitted to link to other nodes whose types have the same or a higher level. HTML definitely does not have that property; for example, you can have a <ul> that contains <li> nodes, that themselves contain <ul> nodes.

> Do we really need so generic list, tree graph data structures these days? They are basically list<T>, TreeNode<T>, GraphNode<T> nowadays and we should just state that GraphNode<T> can't link a GraphNode<T>.

How on earth do you represent a graph of more than one node using a GraphNode<T> type that can't link to other GraphNode<T>s?

More fundamentally, the fact is that some graphs have cycles. Either your node type can represent those graphs, and so can potentially contain reference cycles, or it can't represent those graphs, and so isn't a general graph node type. I don't see how you can square that circle.

> Worse, reference cycles can involve reference types other than shared_ptr; any kind of ownership relationship can participate in a cycle, so you face the same dilemma about e.g. whether to annotate unique_ptr. You've also glossed over how this interacts with separate compilation: when you compile something like
>
> ```
> struct X {
>  shared_ptr<Y> y;
>  Z* z;
> };
> ```
>
> Y and Z may be incomplete types, in which case the compiler has no way of knowing if they're annotated or not. That might be solvable, but only by making [[cycle_free]] even more restrictive and useless.

if
Y is struct Y { int a; };

and
Z is struct Z { char* a };
then this struct hierarchy is safe, isn't it? If this is safe we should extend this pattern to maximum level.

This check can be performed after the compilation. In this case a central data file is generated, where every [[cycle_free]] and [[link]] is collected.

This approach is start from a safe core pattern where cycle is not possible and should be carefully extended.

This pattern is safe, but extremely narrow, and fundamentally cannot be extended to support general computation in a useful way.
--

You received this message because you are subscribed to the Google Groups "ISO C++ Standard - Future Proposals" group.
To unsubscribe from this group and stop receiving emails from it, send an email to std-proposals+unsubscribe@isocpp.org.
To post to this group, send email to std-proposals@isocpp.org.
Visit this group at http://groups.google.com/a/isocpp.org/group/std-proposals/.

[Quoted text hidden]

---

hun.nemethpeter@gmail.com <hun.nemethpeter@gmail.com> Thu, Feb 13, 2014 at 6:02 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> A cycle is not possible in an HTML document, but that's because the nodes form a tree structure, and trees have no cycles. This is enforced by the grammatical structure of HTML as a context-free language, not by any kind of type system requirement on nodes. What you're proposing is different: you want the node *types* to form a strict hierarchy, with nodes not permitted to link to other nodes whose types have the same or a higher level. HTML definitely does not have that property; for example, you can have a <ul> that contains <li> nodes, that themselves contain <ul> nodes.

Yep.. You are right, this approach looks like a no-go. I have no idea now how to enforce tree like, no-cycle structure on type level but it would be useful.

> Do we really need so generic list, tree graph data structures these days? They are basically list<T>, TreeNode<T>, GraphNode<T> nowadays and we should just state that GraphNode<T> can't link a GraphNode<T>.

> How on earth do you represent a graph of more than one node using a GraphNode<T> type that can't link to other GraphNode<T>s?

> More fundamentally, the fact is that some graphs have cycles. Either your node type can represent those graphs, and so can potentially contain reference cycles, or it can't represent those graphs, and so isn't a general graph node type. I don't see how you can square that circle.

Looks like just a type attribute is not good enough here.

But we need a "value hierarchy level" thing. So a value can accept only new stand-alone, or lower-level values. That result in a tree like structure. But I don't know how to use it in a compile-time check.

> Worse, reference cycles can involve reference types
> other than shared_ptr; any kind of ownership relationship
> can participate in a cycle, so you face the same dilemma
> about e.g. whether to annotate unique_ptr. You've also
> glossed over how this interacts with separate
> compilation: when you compile something like
>
> struct X {
> shared_ptr<Y> y;
> Z* z;
> };
>
> Y and Z may be incomplete types, in which case the
> compiler has no way of knowing if they're annotated or
> not. That might be solvable, but only by making
> [[cycle_free]] even more restrictive and useless.

if
Y is struct Y { int a; };
and
Z is struct Z { char* a };
then this struct hierarchy is safe, isn't it? If this is safe we should extend this pattern to maximum level.

> This check can be performed after the compilation. In this
> case a central data file is generated, where every
> [[cycle_free]] and [[link]] is collected.
>
> This approach is start from a safe core pattern where
> cycle is not possible and should be carefully extended.

> This pattern is safe, but extremely narrow, and
> fundamentally cannot be extended to support general
> computation in a useful way.

I agree. I have no idea how to extend this pattern now.

[Quoted text hidden]

Mikhail Semenov <mikhailsemenov1957@gmail.com> Thu, Feb 13, 2014 at 7:39 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On the positive note, I think it's a great idea to introduce optional garbage collection.
As for the null assignment, it may be a good idea to consider this option. If you assign null to the pointer than the object that it points to will obviously be destroyed.
Now, delete: I once was playing with that idea. Imagine, you got a graph or a database, you wan to delete a node. You use delete and all the nodes (pointers) that point to it will be assigned null automatically. There is an issue of timing, of course. You start accessing those nodes during garbage collection, it's not good.
After delete there should be a forced garbage collection.

On Tuesday, February 11, 2014 2:03:27 PM UTC, Andrew Tomazos wrote:

> Hey guys, this is a design I've been toying with (in the abstract for some time actually). It needs a bunch of work, but I would appreciate your feedback on this short draft. Also, if you are aware of any overlapping past proposals that would be great.
>
> Thanks,
> Andrew.
>
> **Precise Per-Type Cyclic Garbage Collection (DRAFT 1)**
>
> [Quoted text hidden]

Sean Middleditch <sean.middleditch@gmail.com> Fri, Feb 14, 2014 at 5:47 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Thursday, February 13, 2014 1:39:42 AM UTC-8, Mikhail Semenov wrote:

> On the positive note, I think it's a great idea to introduce optional garbage collection.
> As for the null assignment, it may be a good idea to consider this option. If you assign null to the pointer than the object that it points to will obviously be destroyed.
> Now, delete: I once was playing with that idea. Imagine, you got a graph or a database, you wan to delete a node. You use delete and all the nodes (pointers) that point to it will be assigned null automatically. There is an issue of timing, of course. You start accessing those nodes during garbage collection, it's not good.
> After delete there should be a forced garbage collection.

It's not currently feasible to allow deterministic destruction of objects in a GC'd world without very severe performance consequences in far too many real-world scenarios. For larger server/HPC apps, forcing a collection across many gigabytes of heap spread out in a NUMA architecture just to delete one object/graph would be non-optimal to the say the least. Even a reasonable desktop today can have apps with several or even dozens of gigabytes of managed objects. For other devices, GC is often avoided (at great pain in managed languages; see any discussion on performance in C# or JavaScript on mobile devices or even desktop-class game development for a more thorough overview; I'm not interested in rehashing that discussion) making the need for `delete` on objects in many apps with small memory working sets relatively moot. The number of use cases where `delete` on a GC'd object would not be a severe performance issue is pretty slim.

[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Fri, Feb 14, 2014 at 6:27 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On Thursday, February 13, 2014 8:47:01 PM UTC+1, Sean Middleditch wrote:
> > On Thursday, February 13, 2014 1:39:42 AM UTC-8, Mikhail Semenov wrote:
> > > On the positive note, I think it's a great idea to introduce optional garbage collection.
> > > As for the null assignment, it may be a good idea to consider this option. If you assign null to the pointer than the object that it points to will obviously be destroyed.
> > > Now, delete: I once was playing with that idea. Imagine, you got a graph or a database, you wan to delete a node. You use delete and all the nodes (pointers) that point to it will be assigned null automatically. There is an issue of timing, of course. You start accessing those nodes during garbage collection, it's not good.
> > > After delete there should be a forced garbage collection.
>
> It's not currently feasible to allow deterministic destruction of objects in a GC'd world without very severe performance consequences in far too many real-world scenarios. For larger server/HPC apps, forcing a collection across many gigabytes of heap spread out in a NUMA architecture just to delete one object/graph would be non-optimal to the say the least. Even a reasonable desktop today can have apps with several or even dozens of gigabytes of managed objects. For other devices, GC is often avoided (at great pain in managed languages; see any discussion on performance in C# or JavaScript on mobile devices or even desktop-class game development for a more thorough overview; I'm not interested in rehashing that discussion) making the need for `delete` on objects in many apps with small memory working sets relatively

> moot. The number of use cases where `delete` on a GC'd object would not be a severe performance issue is pretty slim.

Ok, it sounds like the conclusion here is that a delete expression should be ill-formed on a collecting pointer type. If you want to delete a collected object, assign your collecting pointer to nullptr and it will be destroyed at some non-deterministic point in the future.

[Quoted text hidden]

Matthew Woehlke <mw_triad@users.sourceforge.net> Fri, Feb 14, 2014 at 6:28 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On 2014-02-13 04:39, Mikhail Semenov wrote:
> You use delete and all the nodes (pointers) that point to it will be
> assigned null automatically.

Isn't this what weak pointers are for?

--
Matthew
[Quoted text hidden]

---

inkwizytoryankes@gmail.com <inkwizytoryankes@gmail.com> Fri, Feb 14, 2014 at 6:46 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org
Cc: hun.nemethpeter@gmail.com

Its "easy" :> const variables cant create cycles. Its because to create cycle you need modify existing pointer to point new object (constructor is only exception there). This have big draw back, modify data require coping lot of data.
I once created naive Lisp implementation, everything work as excepted until I try add variables. This break constnes of data structure and introduce cycles again because variables are indented do point any data.
If noconst data stored in that structure cant reference elements of that structure its still impossible possible to create cyclic data.

On Wednesday, February 12, 2014 9:02:38 PM UTC+1, hun.nem...@gmail.com wrote:

> A cycle is not possible in an HTML document, but that's because the nodes form a tree structure, and trees have no cycles. This is enforced by the grammatical structure of HTML as a context-free language, not by any kind of type system requirement on nodes. What you're proposing is different: you want the node *types* to form a strict hierarchy, with nodes not permitted to link to other nodes whose types have the same or a higher level. HTML definitely does not have that property; for example, you can have a <ul> that contains <li> nodes, that themselves contain <ul> nodes.
>
> Yep.. You are right, this approach looks like a no-go. I have no idea now how to enforce tree like, no-cycle structure on type level but it would be useful.
>
> [Quoted text hidden]

Mikhail Semenov <mikhailsemenov1957@gmail.com> Fri, Feb 14, 2014 at 7:15 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org
Cc: mw_triad@users.sourceforge.net

How would I organize a cyclic structure with shared/weak pointers? When you use garbage collection you just use pointers, all pointers own the structure the point to.
They are all equal.
[Quoted text hidden]
[Quoted text hidden]

Matthew Woehlke <mw_triad@users.sourceforge.net> Fri, Feb 14, 2014 at 7:24 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On 2014-02-13 16:15, Mikhail Semenov wrote:
> > On Thursday, February 13, 2014 8:28:21 PM UTC, Matthew Woehlke wrote:
> > > On 2014-02-13 04:39, Mikhail Semenov wrote:
> > > > You use delete and all the nodes (pointers) that point to it will be
> > > > assigned null automatically.
> > >
> > > Isn't this what weak pointers are for?
> >
> > How would I organize a cyclic structure with shared/weak pointers? When you
> > use garbage collection you just use pointers, all pointers own the
> > structure the point to.
> > They are all equal.

I wasn't talking about std::weak_ptr specifically (which isn't helpful for this purpose, no). I was talking about the general concept of a weak pointer, which is a pointer class where each object has a pointer to a shared instance of the pointer class, which in turn has a pointer the actual object. So that when you want to delete it, the only place you need to null a pointer is on the pointer class. (The pointer class itself is just strongly ref-counted in the usual manner.)

(Hmm... actually a shared_ptr<unique_ptr<T>> might work here... you'd have to do the double dereference by hand, but you could specialize or subclass to work around that.)
[Quoted text hidden]

Andrew Tomazos <andrewtomazos@gmail.com> Sat, Feb 15, 2014 at 12:07 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> On Wednesday, February 12, 2014 6:02:32 AM UTC+1, Thiago Macieira wrote:
> Em ter 11 fev 2014, às 20:36:57, Andrew Tomazos escreveu:
> > It should be clear that such a system as a pure library solution is
> > extremely awkward to use and unsafe. Under the proposal, this graph
> > tracking is instrumented automatically by the compiler. Given the
> > extremely high demand for this feature, it should be clear that a core
> > language addition is warranted.
>
> Can it wait for compile-time reflection support and simply use that to detect
> which members are collecting pointers?

Sorry Thiago, I wasn't ignoring your good question, I just needed some time to think about it.

If we imagine a pure library solution in which there are two classes provided:

```
std::collected_type
std::collecting_ptr<T>
```

Deriving from std::collected_type marks the type as a collected type. std::collecting_ptr<T> is a collecting pointer where T must be a collected type.

First, I think that even if we could implement these, the interface may be unacceptably inferior to a core language feature. For example, "gc-unaware" raw pointers and references to collected types are still possible under this scheme, and I think we would like this ill-formed for safety. Likewise, one could multiply inherit from a collected type and a non-collected type.

Putting that aside, how would we implement these classes with reflection? In the constructor of std::collected_type we don't know what the (dynamic) derived type of the complete object we are in is, so even if we had a reflection facility that allowed us to iterate data members, we can't get a handle on the complete type. I think the instrumentation needs to take place at the kind of level in the implementation that works with generating vtables and similar, and I don't think any of the (even in the abstract) reflection mechanisms are planned to be so powerful.

[Quoted text hidden]

Thiago Macieira <thiago@macieira.org> Sat, Feb 15, 2014 at 3:27 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Em sex 14 fev 2014, às 06:07:00, Andrew Tomazos escreveu:
> In the constructor of std::collected_type we don't know what the (dynamic)
> derived type of the complete object we are in is, so even if we had a
> reflection facility that allowed us to iterate data members, we can't get a
> handle on the complete type. I think the instrumentation needs to take
> place at the kind of level in the implementation that works with generating
> vtables and similar, and I don't think any of the (even in the abstract)
> reflection mechanisms are planned to be so powerful.

Is it necessary at the time of the constructor? Or is it only necessary when
something begins collecting the type?

If I create a collectable type on the stack, it can't get be GC'ed.

--
Thiago Macieira - thiago (AT) macieira.info - thiago (AT) kde.org
Software Architect - Intel Open Source Technology Center
PGP/GPG: 0x6EF45358; fingerprint:
E067 918B B660 DBD1 105C 966C 33F5 F005 6EF4 5358

[Quoted text hidden]

Philipp Maximilian Stephani <p.stephani2@gmail.com> Sun, Feb 16, 2014 at 2:38 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Another bad thing about reference counting is that it forces atomic operations, which can kill performance in multi-threaded applications. The problem gets worse the more cores per machine we get.
The problem I see is that garbage collection in managed languages can only get better, and simplistic attempts like reference counting can only get worse. I don't have data, but I'd expect typical Java programs to outperform equivalent reference-counted C++ programs even today or in the near future.
[Quoted text hidden]

Jeffrey Yasskin <jyasskin@google.com> Sun, Feb 16, 2014 at 3:32 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

I haven't read the whole thread, but see
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2297.html#cycles
and
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2286.pdf
[Quoted text hidden]

Thiago Macieira <thiago@macieira.org> Sun, Feb 16, 2014 at 3:51 AM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

Em sáb 15 fev 2014, às 16:38:39, Philipp Maximilian Stephani escreveu:
> Another bad thing about reference counting is that it forces atomic
> operations, which can kill performance in multi-threaded applications. The
> problem gets worse the more cores per machine we get.
> The problem I see is that garbage collection in managed languages can only
> get better, and simplistic attempts like reference counting can only get
> worse. I don't have data, but I'd expect typical Java programs to
> outperform equivalent reference-counted C++ programs even today or in the
> near future.

I think you're generalising based on sketchy information. You've confessed to having no data to prove your theory, so I can make the opposite claim with equally little data and we'd be no better off.

Modern CPUs share data in block units of cache lines. In order to execute an atomic operation, CPUs need to somehow ensure that other execution units in the system don't modify the same cacheline at the same time. And there are multiple techniques to do that, some used by very high performance servers and designed for this very kind of contentious sharing. You're also discounting advances in hardware techniques that could improve performance, as you can see from the intel TSX extensions.
[Quoted text hidden]

Evgeny Panasyuk <evgeny.panasyuk@gmail.com> Tue, Feb 18, 2014 at 6:57 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

> 11 Feb 2014 г., 18:03:27 UTC+4 Andrew Tomazos:
> We propose a core language feature that allows objects of user-selected class types to be cyclically garbage collected. Constraints on the usage of class types so selected, and pointers to such class types, are imposed to enable the implementation of fast safe precise collection.

I think it, or maybe most part of it, can be implemented as library-only solution.
Refer following examples:
http://www.codeproject.com/Articles/912/A-garbage-collection-framework-for-C
http://sourceforge.net/projects/smieciuch/
[Quoted text hidden]

David Krauss <potswa@gmail.com> Tue, Feb 18, 2014 at 8:02 PM

Reply-To: std-proposals@isocpp.org
To: std-proposals@isocpp.org

On Feb 18, 2014, at 4:57 PM, Evgeny Panasyuk <evgeny.panasyuk@gmail.com> wrote:

11 Feb 2014 г., 18:03:27 UTC+4 Andrew Tomazos:

> We propose a core language feature that allows objects of user-selected class types to be cyclically garbage collected. Constraints on the usage of class types so selected, and pointers to such class types, are imposed to enable the implementation of fast safe precise collection.

I think it, or maybe most part of it, can be implemented as library-only solution.
Refer following examples:
http://www.codeproject.com/Articles/912/A-garbage-collection-framework-for-C
http://sourceforge.net/projects/smieciuch/

He acknowledges that already, but the core language feature is supposed to improve the interface by hooking the constructors to the collector.

It would be nice to see the specific library Andrew has in mind, though. Unconditional registration by the constructor isn't usually called GC.

Many models are possible. I've made one intrusive GC where the root pointers were registered, and used to seed a mark-and-sweep, and one where all the managed pointers were registered, and instead of mark-and-sweep it just checked whether each allocation arena was occupied at all. Both provided significant gains, and it seems like this

proposal is based on something completely different. I wonder how it works, how generally applicable it is, and what are the gains.

A general intrusive GC facility would ideally accommodate several models. But which rough edges need to be smoothed has to be spelled out specifically, since reasonable libraries do already essentially work.

I wish I understood how the C++11 GC support features (reachable and safely-derived pointers) are supposed to enable non-intrusive implementations… or had an available implementation to play with.