A brief introduction to C++'s model for type- and resource-safety

- fundamental recurring errors:
  - resource leaks
  - access through an invalid pointer
  - memory corruption (dynamic memory is often reused and if accessed through an invalidated pointer it will be corrupted)
  - confusion between static and dynamic objects (free on static object, no free on dynamic object)
  - type-incorrect deallocation
- these problems have persisted for more than forty years starting in C
- garbage collection and finalizers are not an adequate solution

Design constraints

- ideals:
  i. Perfect static type safety
  ii. Automatic release of general resources (perfect resource safety)
  iii. No run-time overhead compared to good hand-crafted code (the zero-overhead principle)
  iv. No restriction of the application domain compared to C and previous versions of C++
  v. Compatibility with previous versions of C++ (long-term stability is a feature)

Memory safety

- dangling pointers are a major problem

- ctor/dtor handles the resource management portion of the problem

- pointers can escape scopes through:

  - global variables
  - smart pointers
  - containers of pointers
  - objects holding pointers
  - more

- memory safety means all objects are allocated and deallocated once and only once and no object (pointer/reference etc) is accessed outside of this lifetime

- type safety is related to memory safety -> objects cannot be accessed through dangling pointers, no range errors, no null pointer access, etc
- use `owner` objects to manage pointers
    - includes pointer containers, smart pointers, etc
- dynamic ownership model:
    - attach an ownership bit to every pointer
    - set the ownership bit when a value returned from new is assigned
    - use delete whenever a pointer with the ownership bit is set goes out of scope
    - use delete before overwriting a pointer with an ownership bit set
    - clear the ownership bit when a pointer not returned from new is assigned
    - set the ownership bit when a pointer with the ownership bit set is assigned (and clear the ownership bit on the source; an object cannot have two owners)
- does not work in C++ (violates zero-overhead principal)
- static ownership model:
    - Mark every T* that is an owner as owner<T*> in the source code.
    - Let new return an owner<T*>
    - Make sure that every owner<T*> is deleted or transferred to another owner<T*>
    - Never assign a plain T* to an owner<T*>
- simply:

```
template<typename X> using owner = X;
```

- mainly used to assist programmers and static analysis tools
- see the GSL for more
- limitations include mixture with containers, returning from functions
- pointer safety:
    - Don't transfer a pointer to a local to where it could be accessed by a caller
    - A pointer passed as an argument can be passed back as a result
    - A pointer obtained from new can be passed back from a function as a result
- also apply to references and objects containing pointers or references
- pointer invalidation:
    - happens after delete/free
    - may happen in containers like vector after push_back, clear, etc

- a dynamic invalidation model is not possible due to compatibility

- requires a static model and static checking

- be conservative when considering whether a function invalidates a pointer:

    - assume const functions do not invalidate
    - assume non-const functions might invalidate
    - can use `[[lifetime(const)]]` annotation

- limit smart pointer usage in interfaces to locations where pointer ownership is actually manipulated (for generality and performance)

Resource Safety

- always delete new'ed objects only once, always consider where an exception may require
- use smart pointers to ensure safety
- polymorphism requires some form of pointer
- eliminate the use of pointers in containers because run-time polymorphism should not be used with them
- use move semantics to speed access in and out of these containers (enables transferring potentially large objects from one scope to another by "stealing" a handle, etc)

###!!!!IMPORTANT

- move semantics enables complete encapsulation of the management of non-scoped memory

- always place news and deletes inside abstractions - no naked news & no naked deletes

- RAII requires all failable resource acquisition to happen within a resource manager that can handle error reporting/management

- do not allow an owning pointer to be the only handle to an object in a context/scope where an exception can be thrown

Leak-Freedom in C++ By Default

- lifetime guarantees:
    - no dangling/invalid dereferences (use after delete)
    - no null pointer dereferences
    - no leaks (always delete objects once and only once when they are no longer used)
- strategy:
    i. prefer scoped lifetime by default (locals/members directly owned) - 80% of objects

  ii. prefer make_unique & unique_ptr or a container if a an object must have its own lifetime and ownership can be unique (no ownership cycles) -> trees, lists - roughly 10% of objects
  iii. use make_shared and shared_ptrs for an object with its own lifetime and shared ownership with owning cycles -> node-based DAGs (including trees that share references)

- don't use owning raw *'s means don't use explicit delete
- use weak_ptr to break cycles
- always prefer a non-static data member (non pointer) to model a "has-a" relationship
- always prefer a unique_ptr for a decoupled "has-a" relationship
  - separate/delayed/lazy/on-demand initialization
  - changeable component (polymorphism)
  - (manage move operations for non-null decoupled "has-a" members by writing your own)
- use a const unique_ptr for the pimpl idiom (compilation firewall)

```
template<class T>
using Pimpl = const unique_ptr<T>;

class MyClass {
    class Impl; // defined in .cpp
    Pimpl<Impl> pimpl;
    /*... Note: declare destructor and write it elsewhere ...*/
};
```

- use `const unique_ptr<Data[]>` for a dynamic array member (with make_unique)
- use unique_ptr for trees (both node children and root)
  - use a raw Node* for parents and enforce invariant `left->parent == this && right->parent == this`
  - releasing subtrees:
    - recursive:
      - automatic and correct
      - unbounded stack depth
    - iterative:
      - manual optimization
      - bounded stack depth
- use unique_ptr for doubly linked list forward links, root nodes and use raw Node* for reverse links
- used shared_ptr for children and root in Trees that hand out strong references to nodes use shared_ptr for each node's data
- use shared_ptr and vector<shared_ptr> for DAGs of heap objects (use vector<Node *> for parent references) use shared_ptr for each node's data

- for circular lists still use unique_ptrs but use a dummy node for the head and keep a reference in each node, when calling next either return next.get() or head.get()
- for cyclic graphs with nodes use vector<Node *> for children and parents in nodes and for roots, use vector<unique_ptr> nodes for unreachable node
- for factories
    - prefer unique_ptr + make_unique (if the object might not be shared by subsequent code)
    - use shared_ptr + make_shared if the object will definitely be shared
- for factories + caching:
    - use shared_ptrs with weak_ptrs
- avoiding issues (cycles):
    - don't pass an owner to unknown code
    - don't store an owner in a callback
- not all cycles can be broken with weak_ptrs
    - experimenting with deferred_ptrs and deferred containers to resolve naturally

Lifetime Safety: Preventing Leaks and Dangling

- goal: eliminate leaks and dangling for */&/iterators/views/ranges
    - no leaks or dangling
    - need efficiency of just using an address
    - cannot add run-time overhead
    - cannot add significant source code impact
    - cannot add requirements (for analysis) that adds excessive false positives
- uses owner<> type alias

I. Approach and principles

- basic rules:
    1. Prefer to allocate heap objects using (owning) make_unique/make_shared or containers
    2. Otherwise, use owner<T*> for source/layout compatibility with old code. Each non-null owner<> must be deleted exactly once, or moved
    3. Never dereference a null or invalid Pointer
    4. Never allow an invalid Pointer to escape a function

II. Informal overview and rationale

- shared owner (shared_ptr, raw_shared_owner) -> can't dangle
- unique owner (containers, unique_ptr, owner) -> can't dangle
- pointer non-owning (raw * and &, iterators, ranges, views) -> can dangle

1. Aliasing: taking addresses and dereferencing

- non-owning raw pointer
- local or member pointer is never an owner (only onwer<> via new)
- T& is not owner<T&> and no conversion from T* to owner<T*>
- address of local variable is invalidated at end of scope
- address of member variable is invalidated at end of containing object's lifetime
- a pointer or object obtained by dereferencing has a lifetime dictated by the pointer/object it refers to

2. Invalidation by modifying Owners

- modifying ownership invalidates any pointer
- containers of containers

3. Branches

4. Loops

5. null

6. throw and catch

7. Calling functions (arguments and in/inout parameters)

8. Calling functions (return values and out/inout parameters)

9. Transferring ownership

10. Lifetime const

- NOTE: review each for all details -> contains applied examples at end

A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management

- TODO: more here
- for now ->
- object construction:

  i. Every object and each of its subobjects (if any) is constructed exactly once
  ii. No object construction should rely on parts yet to be constructed, and no object destruction should rely on parts already destroyed
  iii. Every object is destroyed in the exact reverse order it was constructed

- (means multi-part initialzation is possible/fine)
- construction stages:

  i. Unconstructed: Construction has not started yet.
  ii. StartedConstructing: The construction of base-class subobjects has started, but not the fields.
  iii. BasesConstructed: The base-class subobjects are completely constructed. Now constructing the fields or executing the constructor body.
  iv. Constructed: The constructor body has returned, and destruction has not started yet.
  v. StartedDestructing: The body of the destructor is executing or the fields are undergoing destruction.
  vi. DestructingBases: The fields have been completely destructed. Bases are undergoing destruction.
  vii. Destructed: All bases and fields have been destructed.

- properties:
  - run time invariants
  - progress
  - safety of field accesses and virtual function calls
  - evolution of construction states
  - object lifetimes
  - RAII

    - generalized dynamic types

Writing Good C++14 By Default

- target guarantee: C++ code compiled in the safe subset is never the root cause of type/memory safety errors, except where explicitly annotated as unsafe

Type safety overview

- GSL
    - byte (not char)
    - variant (tagged union)
- rules:
    1. Don't use reinterpret_cast.
    2. Don't use static_cast downcasts. Use dynamic_cast instead.
    3. Don't use const_cast to cast away const (i.e., at all).
    4. Don't use C-style (T)expression casts that would perform a reinterpret_cast, static_cast downcast, or const_cast.
    5. Don't use a local variable before it has been initialized.
    6. Always initialize a member variable.
    7. Avoid accessing members of raw unions. Prefer variant instead.
    8. Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.

Bounds safety overview

- GSL
    - array_view -> replaces passing an array + length
    - string_view -> convenience for char array_view
- rules:
    1. Don't use pointer arithmetic. Use array_view instead.
    2. Only index into arrays using constant expressions.
    3. Don't use array-to-pointer decay.
    4. Don't use std:: functions and types that are not bounds-checked.

Lifetime safety overview

- use smart pointers (but don't overuse them)

- use raw pointers and references for efficiency, especially on the stack (locals, params, return values)

- GSL
  - Owner (can't dangle): owner<>, containers, smart pointers
  - Pointer (can dangle): *, &, iterators, array_view/string_view, ranges
  - not_null: wraps indirection and enforces non-null
- rules:
  1. Prefer to allocate heap objects using make_unique/make_shared or containers.
  2. Otherwise, use owner<> for source/layout compatibility with old code. Each non-null owner<> must be deleted exactly once, or moved.
  3. Never dereference a null or invalid Pointer.
  4. Never allow an invalid Pointer to escape a function.

Function bodies

- pointer to local -> do not access after scope (increase or decrease lifetime to match)
- address-of and pointer to pointer -> all fine but observe ownership rules
- dereferencing -> fine but observe ownership rules
- pointer from owner -> invalidated after owner modification
- be aware of temporaries for smart pointers and pointers from owners and dereferencing
- branches -> add the possibility of "or", be aware of invalidation in each branch
- loops -> like branches
- nullptr -> do not access if there is the possibility of null or not (consider not_null above)
- try/catch -> treat catch as if it can be entered from any point in the try block
  - check invalidations in try block
  - any and all revalidations in the try block may not have been executed

Function parameters

- for function definitions -> assume any pointer is valid
- in callers -> enforce that no invalid arguments are passed
- do not overuse smart pointers -> use for ownership modification

Function return and in/out values

- minimize need to annotate ownership:
  - output pointers derived from input owner and pointer -> no annotation
  - no inputs = static pointer output -> no annotation (consider && refs)

  - anything else -> annotate
- a returned pointer is assumed to come from owner/pointer inputs

Writing Good C++14

Core Rules

- no leaks
  - scope every object in container/abstraction
  - RAII -> no naked new/delete
- no dangling pointers
  - distinguish owners
  - assume raw pointers are non-owners
  - catch all attempts for a pointer to escape its owner's scope
    - return, throw, out parameters, long-lived containers
  - something that holds an owner is an owner (i.e. containers of owners)
  - applies equally to references, containers of pointers, smart pointers
  - use owner<> for low-level code and static analysis
  - rules:
    - Don’t transfer to pointer to a local to where it could be accessed by a caller
    - A pointer passed as an argument can be passed back as a result
    - A pointer obtained from new can be passed back as a result as an owner
  - do not resort to just testing everywhere
- no type violations through pointers
  - eliminate range errors (array_view, string_view)
  - eliminate nullptr dereference
  - (casts)
  - (unions)

Misuses of smart pointers

- used to:
  - represent ownership
  - avoid dangling pointers (unnecessary if ownership rules are followed)

- issues:
  - adds overhead (shared_ptr)
  - complicates function interfaces
  - may not need pointers
    - unnecessary for scoped objects
    - necessary for:
      - OO interfaces -> polymorphism
      - need to change the object referred to -> ownership and polymorphism
      - (limit cost of passing large objects i.e. use of references)
- uses:
  - void f(T*) -> use; no ownership transfer or sharing
  - void f(unique_ptr) -> transfer unique ownership and use
  - void f(shared_ptr) -> share ownership and use
- do not use a smart pointer in an interface if the function just uses the smart pointer