The Best Designed Library You Shouldn’t Use

Ahmed Charles

A9.COM

Overview

- Motivation
- Features of `shared_ptr`
- Issues using `shared_ptr`
  - Potential uses of `shared_ptr`
  - Guidelines

Motivation

- Learned C++ at University/Microsoft
  - Mostly wrote in other languages
  - The C++ codebases didn't use the STL very much
  - Even then, there was one instance where someone used `shared_ptr` unnecessarily.
- Moved to A9
  - Heavier use of STL/Boost
  - `shared_ptr` seems like a go to mechanism
  - This overuse of `shared_ptr` is probably common
- Influenced by:
  - Herb Sutter – GOTW
  - Sean Parent – GoingNative 2013 talk

Features of `shared_ptr`

- Lifetime management
- Type erasure
  - Allows having `shared_ptr<T>` refer to a T which uses an arbitrary allocation or deletion strategy
- Polymorphism with non-virtual destructors
  - It remembers the original type of the object
- Pointer-like:
  - Syntax
  - Casting
  - Declared using incomplete types
- `weak_ptr` allows breaking circular references/caching

Lifetime management

- Basic use case

```
shared_ptr<T> p1(new T);
```

- Does this do the right thing?

```
shared_ptr<void> p2(new int(5));
```

- Can be stored in containers

```
vector<shared_ptr<T>> vec;
```

- Ultimately, it's really easy to use and really hard to write code that doesn't work.

Type Erasure – Allocation/Deletion

- Custom Allocator (arena)

```
arena a;
shared_ptr<T> p1 =
  allocate_shared<T>(a, args…);
shared_ptr<T> p2 =
  make_shared<T>(args…)
```

- Custom Deleter (FILE)

```
auto fclose_deleter = [](FILE *f)
  { fclose(f); }
shared_ptr<FILE> file(fopen("name", "r"),
                      fclose_deleter);
```

Polymorphism with non-virtual destructors

```
class B { virtual void f() {}
  protected: ~B() {} };
class D : public B {};

shared_ptr<B> p = make_shared<D>();
std::shared_ptr<D> p2 =
  std::dynamic_pointer_cast<D>(p1);
```

Pointer-like - Syntax

- `shared_ptr` has *, ->, conversion to bool, !

```
shared_ptr<T> p = make_shared<T>();
if (p) T t = *p;
if (!p) throw exception("impossible");
else p->some_function();
assert(p != nullptr);
```

Pointer-like - Casting

- `static_pointer_cast`, `dynamic_pointer_cast` and `const_pointer_cast`:

```
shared_ptr<const B> p1 = make_shared<D>();
shared_ptr<const D> p2 =
  static_pointer_cast<const D>(p1);
shared_ptr<const D> p3 =
  dynamic_pointer_cast<const D>(p1);
shared_ptr<B> p4 = const_pointer_cast<B>(p1);
shared_ptr<D> p5 =
  const_pointer_cast<D>(
    dynamic_pointer_cast<const D>(p1));
```

Pointer-like – Incomplete types

```
class C;

shared_ptr<C> p;

class C {};

p = make_shared<C>();
```

weak_ptr – Circular references

- Imagine a node type which tracks it's parent:

```
struct Node {
  shared_ptr<Node> left, right;
  weak_ptr<Node> parent;
};

root.reset();
```

- weak_ptr will break the circular references.

weak_ptr – Cache

- Herb Sutter's Favorite C++ 10-Liner:

```
shared_ptr<widget> get_widget(int id) {
  static map<int, weak_ptr<widget>> cache;
  static mutex m;
  lock_guard<mutex> hold(m);
  auto sp = cache[id].lock();
  if (!sp) cache[id] = sp = load_widget(id);
  return sp;
}

widget& instance() {
  static widget w;
  return w;
}
```

Issues using `shared_ptr`

- Only use `shared_ptr` when there is ambiguous ownership
- `shared_ptr`'s prevent local reasoning about code
- When used in interfaces, `shared_ptr`'s restrict users to a specific lifetime management strategy

Ambiguous ownership sharing data across threads

```
auto ic = make_shared<indexed_corpus>(data);
vector<future<void>> threads;
for (int i = 0; i != 4; ++i) {
  threads.push_back(async([ic] {
    engine(*ic).search();
  }));
}

for (auto t : threads) {
  t.get();
}
```

Local reasoning

- Consider the following:

```
shared_ptr<const element> sp =
  document.get_element(0);
cout << *sp << endl;
document.load(file_path);
cout << *sp << endl;
```

- Does this print the same thing twice or not?

Restricting users of interfaces

- Consider the following definition:

```
string name(shared_ptr<employee> e)
  { return e->name; }
```

- What restrictions does this imply?
- Alternative:

```
string name(const employee& e)
  { return e.name; }
```

- `shared_ptr` parameters imply sink functions

Potential places to use `shared_ptr`

- Local scope
- Global/Static scope
- Member variable
- Function parameter
- Function return value

Guidelines – Local scope

- `shared_ptr`'s are only really useful at local scope as an intermediary step before being given to a sink function or assigned to a non-local variable

```
shared_ptr<data> p = get_data();
data_processor proc(p);
```

- Prefer using values as locals instead of doing allocations

Guidelines – Global/Static scope

- Globals should generally be avoided
- `shared_ptr`'s are effectively globals
  - They do not allow local reasoning

Guidelines – Member variable

- Shared data
  - Use `shared_ptr`
- Normal members
  - Prefer storing by value

```
class Bad { shared_ptr<string> name; };
class Good { string name; };
```

Guidelines – Member variable

- Polymorphic members
  - Use `unique_ptr` with an explicit copy in the copy constructor or `shared_ptr` to `const`

```
class Bad { shared_ptr<B> base_; }
class Good {
  Good(const Good& o)
    : base_(o.base_->copy()) {}
  unique_ptr<B> base_;
}
class Best { shared_ptr<const B> base_; }
```

  - Good and Best are regular types.

Guidelines – Function parameters

- Read only parameters
  - `const` reference
- Modified parameters
  - Non-`const` reference
- Optional parameters
  - Raw pointers to `const`
- Optional Modified parameters
  - Raw pointers to non-`const`

Guidelines – Sink Function Parameters

- Sink functions make a copy of their parameter and store it
- Sink functions
  - Pass by value
- Sink functions with optional data
  - `boost/std::optional` by value
- Sink functions with shared data
  - Use `shared_ptr`

Guidelines – Function return value

- Factory functions
  - Prefer to return by value
  - If it is polymorphic, use `unique_ptr`
    - If it needs to be shared later, it can be transferred to a `shared_ptr`
- Singletons
  - Return by reference instead
- Ambiguous lifetimes (e.g. multithreading)
  - Use `shared_ptr`

Thanks & Questions

- Sean Parent's talks at GoingNative
  - http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning
  - http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil
- Herb Sutter's GOTW on shared_ptr parameters
  - http://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/