Smart Pointer Guidelines

What are smart pointers?

Smart pointers are a specific kind of "scoping object". They are like regular pointers but can automatically deallocate the object they point to when they go out of scope. Since C++ is not a garbage collected language, such functionality is important. The pattern where scoping objects are used to automatically manage the lifetime of heap-allocated objects is called RAII - **R**esource **A**cquisition **I**s **I**nitialization.

Here's some sample use of `std::unique_ptr<>`, the most common type of smart pointer:

```
// We can put a pointer into a std::unique_ptr<> at construction time...
std::unique_ptr value(base::JSONReader::Read(data));
std::unique_ptr foo_ptr(new Foo(...));

// ...or by using reset().
std::unique_ptr bar_ptr;      // Like "Bar* bar_ptr = nullptr;".
bar_ptr.reset(new Bar(...));  // Now |bar_ptr| is non-nullptr and owns the object.

// We can test the std::unique_ptr<> directly.
if (!value)
  return false;

// get() accesses the raw pointer underneath.
Foo* raw_ptr = foo_ptr.get();

// We can call through the std::unique_ptr<> as if it were a raw pointer.
DictionaryValue* dict;
if (!value->GetAsDictionary(&dict))
  return false;
```

Why do we use them?

Smart pointers ensure we properly destroy an object even if its creation and destruction are widely separated. They make functions simpler and safer by ensuring that no matter how many different exit paths exist, local objects are always cleaned up correctly. They help enforce that exactly one object owns another object at any given time, preventing both leaks and double-frees. Finally, their use clarifies ownership transference expectations at function calls.

What types of smart pointers exist?

The two common smart pointers in Chromium are `std::unique_ptr<>` and `scoped_refptr<>`. The former is used for singly-owned objects, while the latter is used for reference-counted objects (though normally you should avoid these -- see below). If you're familiar with C++11, `scoped_refptr<>` is similar in intent to `std::shared_ptr<>` (Note: the latter is banned).

base/memory/ has a few other objects of interest:

- `WeakPtr<>` is not actually a smart pointer; it functions like a pointer type, but rather than being used to automatically free objects, it's used to track whether an object owned elsewhere is still alive. When the object is destroyed, the `WeakPtr<>` will be automatically set to null, so you can see that it's no longer alive. (You still need to test for null before dereferencing -- a blind dereference of a null `WeakPtr<>` is the equivalent of dereferencing null, rather than a no-op.) This is somewhat like C++11's `std::weak_ptr<>`, but with a different API and fewer restrictions.

When do we use each smart pointer?

- **Singly-owned objects** - use `std::unique_ptr<>`. Specifically, these are for non-reference-counted, heap-allocated objects that you own.
- **Non-owned objects** - use raw pointers or `WeakPtr<>`. Note that `WeakPtr<>`s must only be dereferenced on the same thread where they were created (usually by a `WeakPtrFactory<>`), and if you need to take some action immediately before or after an object is destroyed, you'd likely be better-served with some sort of callback or notification instead of a `WeakPtr<>`.
- **Ref-counted objects** - use `scoped_refptr<>`, but better yet, rethink your design. Reference-counted objects make it difficult to understand ownership and destruction order, especially when multiple threads are involved. There is almost always another way to design your object hierarchy to avoid refcounting. Avoiding refcounting in multithreaded situations is usually easier if you restrict each class to operating on just one thread, and use `PostTask()` and the like to proxy calls to the correct thread. `base::Bind()`, `WeakPtr<>`, and other tools make it possible to automatically cancel calls to such an object when it dies. Note that too much of our existing code uses refcounting, so just because you see existing code doing it does not mean it's the right solution. (Bonus points if you're able to clean up such cases.)
- **Platform-specific types** - use one of the many platform-specific scoping objects, such as `base::win::ScopedHandle`, `base::win::ScopedComPtr`, or `base::mac::ScopedCFTypeRef`. Note that these may have slightly different usage patterns than `std::unique_ptr<>`; for example, they might be assigned as outparams via a `.receive()` type of method.

What are the calling conventions involving different kinds of pointers?

See the calling conventions section of the Chromium style guide for the rules; some common cases are illustrated below.

- If a function **takes** a `std::unique_ptr<>`, that means it takes ownership of the argument. Callers need to use `std::move()` to indicate that they're passing ownership if the object being passed is not a temporary:

```
// Foo() takes ownership of |bar|.
void Foo(std::unique_ptr<Bar> bar);

...
std::unique_ptr<Bar> bar_ptr(new Bar());
Foo(std::move(bar_ptr));          // After this statement, |bar_ptr| is null.
Foo(std::unique_ptr<Bar>(new Bar()));  // No need to use std::move() on temporaries.
```

- If a function **returns** a `std::unique_ptr<>`, that means the caller takes ownership of the returned object.

```
class Base { ... };
class Derived : public Base { ... };
```

```
// Foo takes ownership of |base|, and the caller takes ownership of the returned
// object.
std::unique_ptr<Base> Foo(std::unique_ptr<Base> base) {
  if (cond) {
    return base;                            // Transfers ownership of |base| back to
                                            // the caller.
  }

  // Note that on these next codepaths, |base| is deleted on exit.
  if (cond2) {
    return std::make_unique<Base>();  // No std::move() necessary on temporaries.
  }
  std::unique_ptr<Derived> derived(new Derived());
  return derived;               // No need to use std::move() when
                                // returning local variables.
}
```

- If a function takes or returns a raw pointer, it may mean no ownership is transferred, or it may not. Much of Chromium was written before `std::unique_ptr<>` existed, or by people unfamiliar with using it to indicate ownership transfers, and thus takes or returns raw pointers but transfers ownership in the process. Because the compiler can't enforce correct behavior here, this is less safe. Consider cleaning up such code, so that functions which take or return raw pointers never transfer ownership.

What about passing or returning a smart pointer by reference?

Don't do this.

In principle, passing a `const std::unique_ptr<T>&` to a function which does not take ownership has some advantages over passing a `T*`: the caller can't accidentally pass in something utterly bogus (e.g. an `int` converted to a `T*`), and the caller is forced to guarantee the lifetime of the object persists across the function call. However, this declaration also forces callers to heap-allocate the objects in question, even if they could otherwise have declared them on the stack. Passing such arguments as raw pointers decouples the ownership issue from the allocation issue, so that the function is merely expressing a preference about the former. For the sake of simplicity and consistency, we avoid asking authors to balance these tradeoffs, and simply say to always use raw pointers.

One exception is lambda functions used with STL algorithms operating on containers of smart pointers; these may have to take e.g. `const std::unique_ptr<T>&` in order to compile. And speaking of that...

I want to use an STL container to hold pointers. Can I use smart pointers?

Yes! As of C++11, you can store smart pointers in STL containers.

General references on smart pointers