Serenity C++ patterns: References instead of Pointers

This post describes the use of C++ references to enhance autodocumentation in the Serenity Operating System.

Six years ago I wrote on the WebKit blog about Using References Instead of Pointers. Re-reading that post today, I remember how excited I was about the technique back then.

In the years since that post, references have become part of my everyday C++ vernacular, and consequently references are widely used in Serenity. I'm writing this post to refresh myself on what's great about references, and to share my views with you, dear reader. So let's get to the reasons I love these things…

References enforce object existence at the type level.

If you write a function that takes a `Foo*`, it may receive a `Foo`, but you may also receive `nullptr`. Since you don't know this at compile-time, you have to somehow deal with the scenario in which someone passes you a `nullptr`.

It's not uncommon to see code like this:

```
void do_something(Foo* foo) {
    if (!foo)
        return;
    ...
}
```

If it's an error to call the function with `nullptr`, a common technique to catch bad calls is to `assert` at runtime that the pointer is non-null:

```
void do_something(Foo* foo) {
    assert(foo);
    ...
}
```

While I'm definitely in favor of assertions, I do consider it a bug whenever I see a function that takes a pointer and immediately asserts that it's non-null.

Now let's use a reference instead:

```
void do_something(Foo& foo) {
    // No need to null-check or assert anything.
}
```

By taking a reference, the function communicates to the outside world that it requires a non-null object, and it's the caller's job to validate and dereference any pointers before passing them.

Note: If we don't intend to modify anything about the `Foo` we're being passed, we should take it as a `const Foo&` to prevent `do_something` from calling any of `Foo`'s non-const member functions.

Reference chasing is way less suspicious than pointer chasing.

Consider the following two examples:

```
// Using pointers:
my_object->subsystem()->component()->do_thing();

// Using references:
my_object.subsystem().component().do_thing();
```

When using pointers, it's up to *you* to know that the pointers you're dereferencing are non-null. The compiler has very limited ability to detect mistakes and let you know if you're messing up.

However, if you've been consistently using references, your getters now return T& (or `const T&`) because they know that the object will exist and so they alleviate the caller of having to worry about that.

Getters for objects that may not exist should of course still return T*, which then communicates to the caller that they may be `nullptr` and must be checked.

References are an important part of self-documenting C++.

I’m a big fan of autodocumentation. By sticking to a consistent coding style and combining good names with strong types, we can encode a large amount of information in the code itself.

Consistent use of references contributes greatly to that, since it means you can comfortably make assumptions about object lifetimes and ownerships when you see an & in the Serenity codebase. For example:

- If something returns a `const` T&, we can assume that the T is kept alive for at least as long as the object that vended the reference.
- If something takes a T&, we can assume that it might modify the T by calling any of its non-const member functions.
- And of course, when we spot a `*`, it means we have to watch out and check for `nullptr`.

Until next time!

Written on July 24, 2019