On the surface there does seem to be a difference between how objects `untracked` and `tracked` are constructed. The first variable is having its constructor directly invoked, while the second is being constructed from an object being returned by-value from `trackConstruction`. This construction, however, has long been something that has been optimized away to avoid any additional objects and construct the object in question just once. In this case, because the object being returned is initialized by the **`return`** statement of `trackConstruction`, the optimization is called **return value optimization** (RVO). C++ has always allowed this optimization by enabling **copy elision**. Ensuring that this elision actually happens (on all current compilers of which the authors are aware) is possible by publicly **declaring** but not **defining** the copy constructor for `BigObject`.[3] We find that this code will still compile and link with such an object, providing observable proof that the copy constructor is never actually invoked with this pattern.

Emplacement

Prior to C++11, inserting an object into a Standard Library container always required the programmer to first create such an object and then copy it inside the container's storage. As an example, consider inserting a temporary `std::string` object in an `std::vector<std::string>`:

```
void f(std::vector<std::string>& v)
{
    v.push_back(std::string("hello world"));
        // invokes std::string::string(const char*) and the copy constructor
}
```

In the function above, a temporary `std::string` object is created on the stack frame of `f` and is then copied to the dynamically allocated buffer managed by `v`. Additionally, the buffer might have insufficient capacity and hence might require reallocation, which would in turn require every element of `v` to be copied from the old buffer to the new, larger one.

In C++11, the situation is significantly better thanks to *rvalue* references. The temporary will be moved into `v`, and any subsequent buffer reallocation will *move* the elements between buffers rather than copy them, assuming that the element's move constructor has a **`noexcept`** specifier (see Section 3.1."**`noexcept`** Specifier" on page 1085). The amount of work can, however, be further reduced: What if, instead of first creating an object externally, we constructed the new `std::string` object directly in `v`'s buffer?

This is where **emplacement** comes into play. All standard library containers, including `std::vector`, now provide an emplacement API powered by variadic templates (see Section 2.1."Variadic Templates" on page 873) and perfect forwarding (see *Perfect forwarding for generic factory functions* on page 388). Rather than accepting a fully-constructed element, emplacement operations accept an arbitrary number of arguments, which will in turn

[3] ~~In C++17, this~~ copy elision can be guaranteed and is allowed to be done for objects that have no copy or move constructors.