10
Proxy Objects and Lazy Evaluation

In this chapter, you will learn how to use proxy objects and lazy evaluation in order to postpone the execution of certain code until required. Using proxy objects enables optimizations to occur under the hood, thereby leaving the exposed interfaces intact.

This chapter covers:

- Lazy and eager evaluation
- Using proxy objects to avoid superfluous computations
- Overloading operators when working with proxy objects

Introducing lazy evaluation and proxy objects

First and foremost, the techniques used in this chapter are used to hide optimizations in a library from the user of that library. This is useful because exposing every single optimization technique as a separate function requires a lot of attention and education from the user of the library. It also bloats the code base with a multitude of specific functions, making it hard to read and understand. By using proxy objects, we can achieve optimizations under the hood; the resultant code is both optimized and readable.

Lazy versus eager evaluation

Lazy evaluation is a technique used to postpone an operation until its result is really needed. The opposite, where operations are performed right away, is called **eager evaluation**. In some situations, eager evaluation is undesirable as we might end up constructing a value that is never used.

To demonstrate the difference between eager and lazy evaluation, let's assume we are writing some sort of game with multiple levels. Whenever a level has been completed, we need to display the current score. Here we will focus on a few components of our game:

- A `ScoreView` class responsible for displaying the user's score with an optional bonus image if a bonus was achieved
- An `Image` class that represents an image loaded into memory
- A `load()` function that loads images from disk

The implementation of the classes and functions is not important in this example, but the declarations look like this:

```
class Image { /* ... */ };              // Buffer with JPG data
auto load(std::string_view path) -> Image;   // Load image at path
class ScoreView {
public:
 // Eager, requires loaded bonus image
 void display(const Image& bonus);
 // Lazy, only load bonus image if necessary
 void display(std::function<Image()> bonus);
```

```
  // ...
};
```

Two versions of `display()` are provided: the first one requires a fully loaded bonus image, whereas the second one accepts a function that will be called only if a bonus image is needed. Using the first *eager* version would look like this:

```
// Always load bonus image eagerly
const auto eager = load("/images/stars.jpg");
score.display(eager);
```

Using the second *lazy* version would look like this:

```
// Load default image lazily if needed
auto lazy = [] { return load("/images/stars.jpg"); };
score.display(lazy);
```

The eager version will always load the default image into memory even if it's never displayed. However, the lazy loading of the bonus image will guarantee that the image is only loaded if the `ScoreView` really needs to show the bonus image.

This is a very simple example, but the idea is that your code gets expressed almost in the same way as if it were declared eagerly. A technique for hiding the fact that the code evaluates lazily is to use proxy objects.

Proxy objects

Proxy objects are internal library objects that aren't intended to be visible to the user of the library. Their task is to postpone operations until required and to collect the data of an expression until it can be evaluated and optimized. However, proxy objects act in the dark; the user of the library should be able to handle the expressions as if the proxy objects were not there. In other words, using proxy objects, you can encapsulate optimizations in your libraries while leaving the interfaces intact. You will now learn how to use proxy objects in order to evaluate more advanced expressions lazily.

Avoiding constructing objects using proxy objects

Eager evaluation can have the undesirable effect that objects are unnecessarily constructed. Often this is not a problem, but if the objects are expensive to construct (because of heap allocations, for example), there might be legitimate reasons to optimize away the unnecessary construction of short-lived objects that serve no purpose.

Comparing concatenated strings using a proxy

We will now walk through a minimal example of using proxy objects to give you an idea of what they are and can be used for. It's not meant to provide you with a general production-ready solution to optimizing string comparisons.

With that said, take a look at this code snippet that concatenates two strings and compares the result:

```
auto a = std::string{"Cole"};
auto b = std::string{"Porter"};
```

```
auto c = std::string{"ColePorter"};
auto is_equal = (a + b) == c;       // true
```

Here is a visual representation of the preceding code snippet:

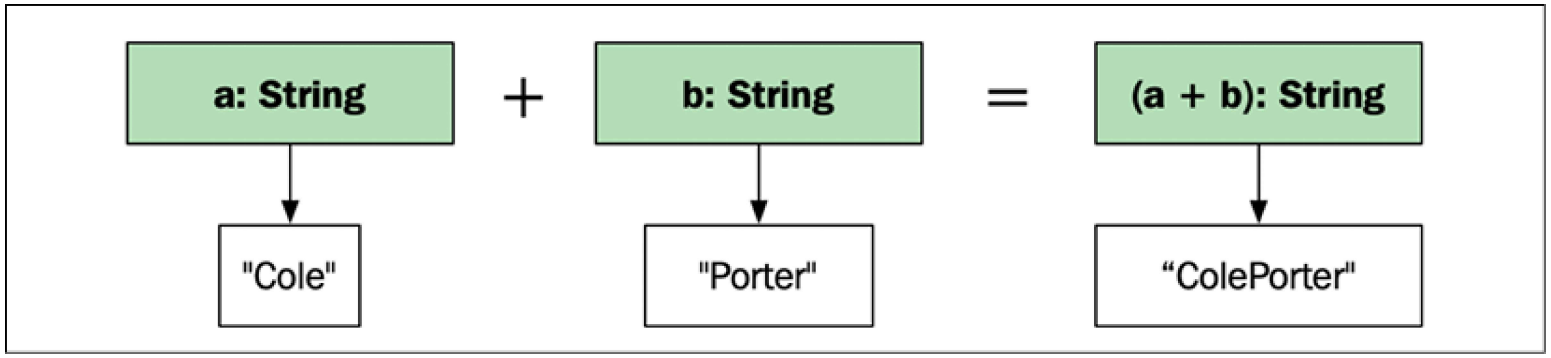

Figure 10.1: Concatenating two strings into a new string

The problem here is that (`a + b`) constructs a new temporary string in order to compare it with `c` . Instead of constructing a new string, we can just compare the concatenation right away, like this:

```
auto is_concat_equal(const std::string& a, const std::string& b,
                     const std::string& c) {
  return
    a.size() + b.size() == c.size() &&
    std::equal(a.begin(), a.end(), c.begin()) &&
    std::equal(b.begin(), b.end(), c.begin() + a.size());
}
```

We can then use it like this:

```
auto is_equal = is_concat_equal(a, b, c);
```

Performance-wise, we've achieved a win, but syntactically, a code base littered with special-case convenience functions like this is hard to maintain. So, let's see how this optimization can be achieved with the original syntax still intact.

Implementing the proxy

First, we'll create a proxy class representing the concatenation of two strings:

```
struct ConcatProxy {
  const std::string& a;
  const std::string& b;
};
```

Then, we'll construct our own `String` class that contains a `std::string` and an overloaded `operator+()` function. Note that this is an example of how to make and use proxy objects; creating your own `String` class is not something I recommend:

```
class String {
public:
  String() = default;
  String(std::string str) : str_{std::move(str)} {}
  std::string str_{};
```

```
};

auto operator+(const String& a, const String& b) {
  return ConcatProxy{a.str_, b.str_};
}
```

Here's a visual representation of the preceding code snippet:

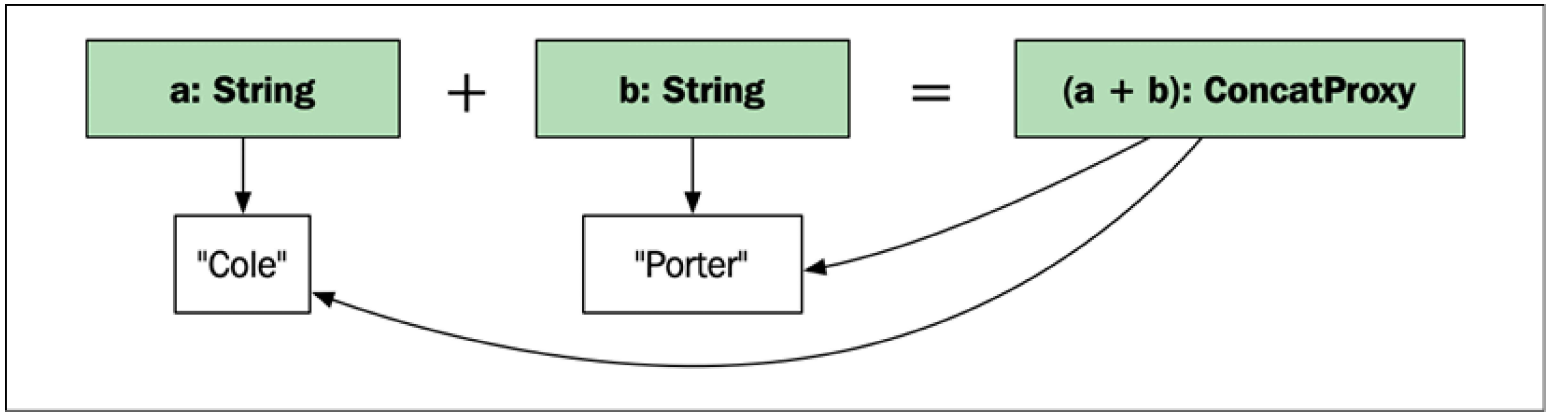

Figure 10.2: A proxy object representing the concatenation of two strings

Lastly, we'll create a global `operator==()` function, which in turn will use the optimized `is_concat_equal()` function, as follows:

```
auto operator==(ConcatProxy&& concat, const String& str) {
  return is_concat_equal(concat.a, concat.b, str.str_);
}
```

Now that we have everything in place, we can get the best of both worlds:

```
auto a = String{"Cole"};
auto b = String{"Porter"};
auto c = String{"ColePorter"};
auto is_equal = (a + b) == c;     // true
```

In other words, we gained the performance of `is_concat_equal()` while preserving the expressive syntax of using `operator==()`.

The rvalue modifier

In the preceding code, the global `operator==()` function only accepts `ConcatProxy` rvalues:

```
auto operator==(ConcatProxy&& concat, const String& str) { // ...
```

If we were to accept a `ConcatProxy` lvalue, we could end up accidentally misusing the proxy, like this:

```
auto concat = String{"Cole"} + String{"Porter"};
auto is_cole_porter = concat == String{"ColePorter"};
```

The problem here is that both the temporary `String` objects holding `"Cole"` and `"Porter"` have been destructed by the time the comparison is executed, leading to a failure. (Remember that the `ConcatProxy` class only holds references to the strings.) But since we forced the `concat` object to be an rvalue, the

preceding code will not compile and thereby saves us from a likely runtime crash. Of course, you could force it to compile by casting it to an rvalue using `std::move(concat) == String("ColePorter")`, but that wouldn't be a realistic case.

Assigning a concatenated proxy

Now, you might be thinking, what if we actually want to store the concatenated string as a new string rather than just compare it? What we do is simply overload an `operator String()` function, as follows:

```
struct ConcatProxy {
  const std::string& a;
  const std::string& b;
  operator String() const && { return String{a + b}; }
};
```

The concatenation of two strings can now implicitly convert itself to a string:

```
String c = String{"Marc"} + String{"Chagall"};
```

There is one little snag, though: we cannot initialize the new `String` object with the `auto` keyword, as this would result in `ConcatProxy`:

```
auto c = String{"Marc"} + String{"Chagall"};
// c is a ConcatProxy due to the auto keyword here
```

Unfortunately, we have no way to get around this; the result must be explicitly cast to `String`.

It's time to see how much faster our optimized version is compared to the normal case.

Performance evaluation

To evaluate the performance benefits, we'll use the following benchmark, which concatenates and compares `10'000` strings of size `50`:

```
template <typename T>
auto create_strings(int n, size_t length) -> std::vector<T> {
 // Create n random strings of the specified length
 // ...
}
template <typename T>
void bm_string_compare(benchmark::State& state) {
 const auto n = 10'000, length = 50;
 const auto a = create_strings<T>(n, length);
 const auto b = create_strings<T>(n, length);
 const auto c = create_strings<T>(n, length * 2);
 for (auto _ : state) {
  for (auto i = 0; i < n; ++i) {
   auto is_equal = a[i] + b[i] == c[i];
   benchmark::DoNotOptimize(is_equal);
  }
 }
}
BENCHMARK_TEMPLATE(bm_string_compare, std::string);
```

```
BENCHMARK_TEMPLATE(bm_string_compare, String);
BENCHMARK_MAIN();
```

I achieved a 40x speedup using gcc when executing on an Intel Core i7 CPU. The version using `std::string` directly completed in 1.6 ms, whereas the proxy version using `String` completed in only 0.04 ms. When running the same test using short strings of length 10, the speedup was around 20x. One reason for the big variation is that small strings will avoid heap allocations by utilizing the small string optimization discussed in *Chapter 7*, *Memory Management*. The benchmark shows us that the speedup with a proxy object is considerable when we get rid of the temporary string and the possible heap allocation that comes with it.

The `ConcatProxy` class helped us to hide an optimization when comparing strings. Hopefully this simple example has inspired you to start thinking about ways to keep your API design clean while implementing performance optimizations.

Next, you will see another useful optimization that can be hidden behind a proxy class.

Postponing sqrt computations

This section will show you how to use a proxy object in order to postpone, or even avoid, using the computationally heavy `std::sqrt()` function when comparing the length of two-dimensional vectors.

A simple two-dimensional vector class

Let's start with a simple two-dimensional vector class. It has *x* and *y* coordinates and a member function called `length()` that calculates the distance from the origin to the location *(x, y)*. We will call the class `Vec2D`. Here follows the definition:

```
class Vec2D {
public:
  Vec2D(float x, float y) : x_{x}, y_{y} {}
  auto length() const {
    auto squared = x_*x_ + y_*y_;
    return std::sqrt(squared);
  }
private:
  float x_{};
  float y_{};
};
```

Here is an example of how clients can use `Vec2D`:

```
auto a = Vec2D{3, 4};
auto b = Vec2D{4, 4};
auto shortest = a.length() < b.length() ? a : b;
auto length = shortest.length();
std::cout << length; // Prints 5
```

The example creates two vectors and compares their lengths. The length of the shortest vector is then printed to standard out. *Figure 10.3* illustrates the vector and the calculated length to the origin:

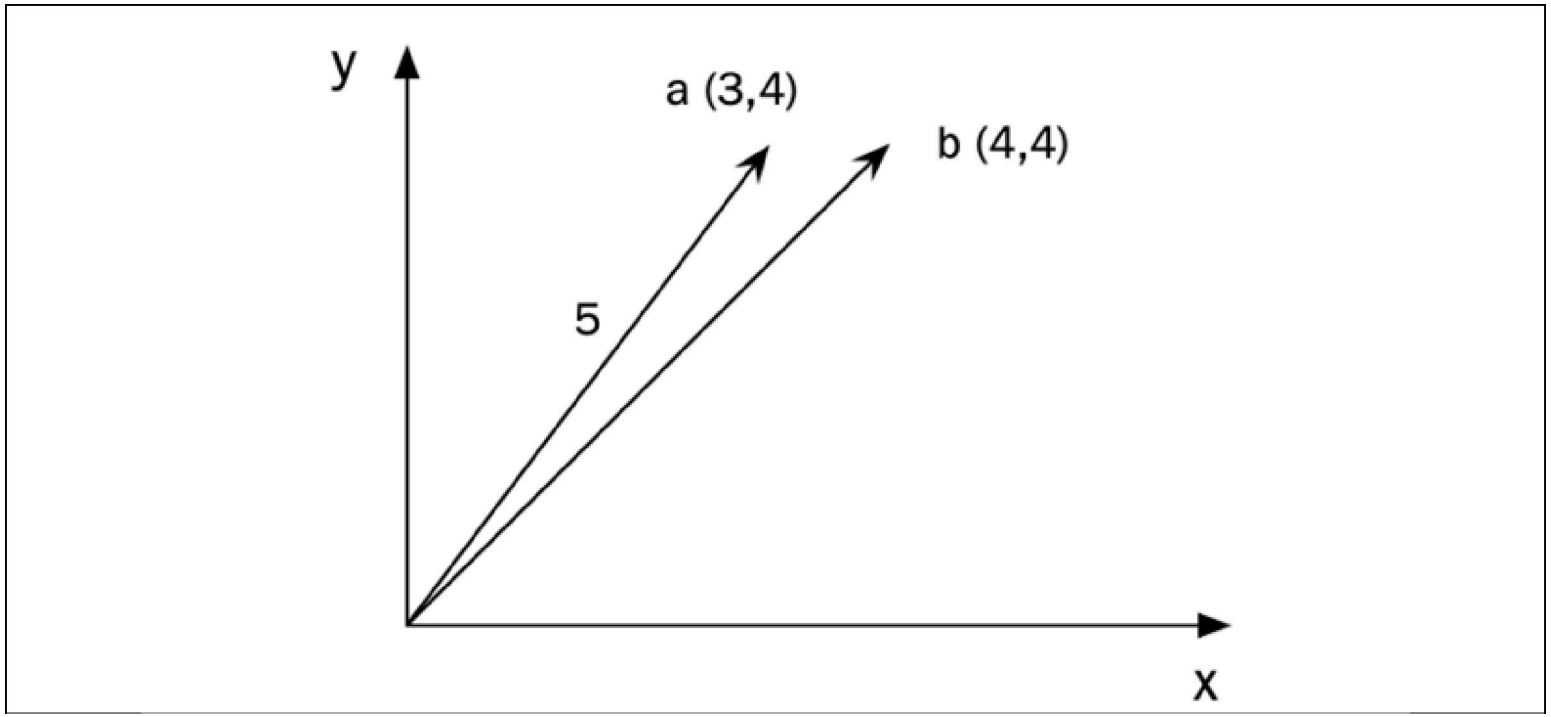

Figure 10.3: Two 2D vectors of different lengths. The length of vector a is 5.

The underlying mathematics

Looking into the mathematics of the calculation, you may notice something interesting. The formula used for length is as follows:

$$length = \sqrt{x^2 + y^2}$$

However, if we only need to compare the distance between two vectors, the squared length is all we need, as the following formula shows:

$$length^2 = x^2 + y^2$$

The square root can be computed using the function `std::sqrt()`. But, as mentioned, as the square root operation is not required if we just want to compare lengths between two vectors, we can omit it. The nice thing is that `std::sqrt()` is a relatively slow operation, meaning that if we compare a lot of vectors by their length, we can gain some performance. The question is, how can we do this while preserving a clean syntax? Let's see how we can use a proxy object to make a simple library perform this optimization under the hood when comparing lengths.

For clarity, we start with the original `Vec2D` class but we split the `length()` function into two parts – `length_squared()` and `length()`, as follows:

```
class Vec2D {
public:
  Vec2D(float x, float y) : x_{x}, y_{y} {}
  auto length_squared() const {
    return x_*x_ + y_*y_;
  }
  auto length() const {
    return std::sqrt(length_squared());
  }
private:
  float x_{};
  float y_{};
};
```

Now clients to our `Vec2D` class can use `length_squared()` if they want to gain some performance when only comparing the lengths of different vectors.

Let's say that we want to implement a convenient utility function that returns the minimum length of a range of `Vec2D` objects. We now have two options: either use the `length()` function or the `length_squared()` function when doing the comparison. Their corresponding implementations are shown in the following examples:

```
// Simple version using length()
auto min_length(const auto& r) -> float {
  assert(!r.empty());
  auto cmp = [](auto&& a, auto&& b) {
    return a.length () < b.length();
  };
  auto it = std::ranges::min_element(r, cmp);
  return it->length();
}
```

The second optimized version using `length_squared()` for comparison would look like this:

```
// Fast version using length_squared()
auto min_length(const auto& r) -> float {
  assert(!r.empty());
  auto cmp = [](auto&& a, auto&& b) {
    return a.length_squared() < b.length_squared(); // Faster
  };
  auto it = std::ranges::min_element(r, cmp);
```

```
    return it->length(); // But remember to use length() here!
}
```

The first version using `length()` inside `cmp` has the advantage of being more readable and easier to get right, whereas the second version has the advantage of being faster. To remind you, the speedup of the second version is because we can avoid the call to `std::sqrt()` inside the `cmp` lambda.

The optimal solution would be to have the syntax of the first version using `length()` and the performance of the second version using `length_squared()`.

Depending on the context this class will be used in, there might be good reasons to expose a function such as `length_squared()`. But let's assume that other developers on our team don't understand the reason for having the `length_squared()` function and find the class confusing. So, we decide to come up with something better to avoid having two versions of a function that exposes a length property of the vector. As you might have guessed, it's time for a proxy class that hides this complexity.

In order to achieve this, instead of returning a `float` value from the `length()` member function, we return an intermediate object hidden from the user. Depending on how the user uses the hidden proxy object, it should avoid the `std::sqrt()` operation until it is really required. In sections to come, we will implement a class called `LengthProxy`, which will be the type of proxy object we will return from `Vec2D::length()`.

Implementing the LengthProxy object

It's time to implement the `LengthProxy` class containing a `float` data member that represents the squared length. The actual squared length is never exposed in order to prevent users of the class from

mixing the squared length with the regular length. Instead, `LengthProxy` has a hidden `friend` function that compares its squared length with a regular length, as follows:

```
class LengthProxy {
public:
  LengthProxy(float x, float y) : squared_{x * x + y * y} {}
  bool operator==(const LengthProxy& other) const = default;
  auto operator<=>(const LengthProxy& other) const = default;
  friend auto operator<=>(const LengthProxy& proxy, float len) {
    return proxy.squared_ <=> len*len;   // C++20
  }
  operator float() const {     // Allow implicit cast to float
    return std::sqrt(squared_);
  }
private:
  float squared_{};
};
```

We have defined `operator float()` to allow implicit casts from `LengthProxy` to `float`. `LengthProxy` objects can also be compared with each other. By using the new C++20 comparisons, we simply `default` the equality operator and three-way comparison operator to have the compiler generate all the necessary comparison operators for us.

Next, we rewrite the `Vec2D` class to return objects of the class `LengthProxy` instead of the actual `float` length:

```
class Vec2D {
public:
  Vec2D(float x, float y) : x_{x}, y_{y} {}
  auto length() const {
    return LengthProxy{x_, y_};    // Return proxy object
  }
  float x_{};
  float y_{};
};
```

With these additions in place, it's time to use our new proxy class.

Comparing lengths with LengthProxy

In this example, we'll compare two vectors, `a` and `b`, and determine whether `a` is shorter than `b`. Note how the code syntactically looks exactly the same as if we had not utilized a proxy class:

```
auto a = Vec2D{23, 42};
auto b = Vec2D{33, 40};
bool a_is_shortest = a.length() < b.length();
```

Under the hood, the final statement is expanded to something similar to this:

```
// These LengthProxy objects are never visible from the outside
LengthProxy a_length = a.length();
LengthProxy b_length = b.length();
```

```
// Member operator< on LengthProxy is invoked,
// which compares member squared_
auto a_is_shortest = a_length < b_length;
```

Nice! The `std::sqrt()` operation is omitted while the interface of the `Vec2D` class is still intact. The simple version of `min_length()` we implemented earlier now performs its comparison more efficiently, as the `std::sqrt()` operation is omitted. What follows is the simple implementation, which now has become efficient as well:

```
// Simple and efficient
auto min_length(const auto& r) -> float {
  assert(!r.empty());
  auto cmp = [](auto&& a, auto&& b) {
    return a.length () < b.length();
  };
  auto it = std::ranges::min_element(r, cmp);
  return it->length();
}
```

The optimized length comparisons between `Vec2D` objects now happen under the hood. The programmer implementing the `min_length()` function doesn't need to know about this optimization in order to benefit from it. Let's see what it looks like if we need the actual length.

Calculating length with LengthProxy

When requesting the actual length, the calling code changes a little bit. To trig the implicit cast to `float`, we have to commit to a `float` when declaring the `len` variable below; that is, we can't just use `auto` as we usually do:

```
auto a = Vec2D{23, 42};
float len = a.length(); // Note, we cannot use auto here
```

If we were to just write `auto`, the `len` object would be of type `LengthProxy` rather than `float`. We do not want the users of our code base to explicitly handle `LengthProxy` objects; proxy objects should operate in the dark and only their results should be utilized (in this case, the comparison result or the actual distance value is `float`). Even though we cannot hide proxy objects completely, let's see how we can tighten them to prevent misuse.

Preventing the misuse of LengthProxy

You may have noted that there can be a case where using the `LengthProxy` class might lead to worse performance. In the example that follows, the `std::sqrt()` function is invoked multiple times according to the programmer's request for the length value:

```
auto a = Vec2D{23, 42};
auto len = a.length();
float f0 = len;      // Assignment invoked std::sqrt()
float f1 = len;      // std::sqrt() of len is invoked again
```

Although this is an artificial example, there can be real-world cases where this might happen, and we want to force users of `Vec2d` to only invoke `operator float()` once per `LengthProxy` object. In order to prevent misuse we make the `operator float()` member function invocable only on rvalues; that is, the `LengthProxy` object can only be converted to a floating point if it is not tied to a variable.

We force this behavior by using `&&` as a modifier on the `operator float()` member function. The `&&` modifier works just like a `const` modifier, but where a `const` modifier forces the member function to not modify the object the `&&` modifier forces the function to operate on temporary objects.

The modification looks like this:

```
operator float() const && { return std::sqrt(squared_); }
```

If we were to invoke `operator float()` on a `LengthProxy` object tied to a variable, such as the `dist` object in the following example, the compiler would refuse to compile:

```
auto a = Vec2D{23, 42};
auto len = a.length(); // len is of type LenghtProxy
float f = len;       // Doesn't compile: len is not an rvalue
```

However, we can still invoke `operator float()` directly on the rvalue returned from `length()`, like this:

```
auto a = Vec2D{23, 42};
float f = a.length();   // OK: call operator float() on rvalue
```

A temporary `LengthProxy` instance will still be created in the background, but since it is not tied to a variable, we are allowed to implicitly convert it to `float`. This will prevent misuse such as invoking `operator float()` several times on a `LengthProxy` object.

Performance evaluation

For the sake of it, let's see how much performance we've actually gained. We will benchmark the following version of `min_element()`:

```
auto min_length(const auto& r) -> float {
  assert(!r.empty());
  auto it = std::ranges::min_element(r, [](auto&& a, auto&& b) {
    return a.length () < b.length(); });
  return it->length();
}
```

In order to compare the proxy object optimization with something, we will define an alternative version, `Vec2DSlow`, which always computes the actual length using `std::sqrt()`:

```
struct Vec2DSlow {
  float length() const {              // Always compute
    auto squared = x_ * x_ + y_ * y_;    // actual length
    return std::sqrt(squared);          // using sqrt()
  }
  float x_, y_;
};
```

Using Google Benchmark with a function template, we can see how much performance we gain when finding the minimum length of 1,000 vectors:

```
template <typename T>
void bm_min_length(benchmark::State& state) {
 auto v = std::vector<T>{};
 std::generate_n(std::back_inserter(v), 1000, [] {
  auto x = static_cast<float>(std::rand());
  auto y = static_cast<float>(std::rand());
  return T{x, y};
 });
 for (auto _ : state) {
  auto res = min_length(v);
  benchmark::DoNotOptimize(res);
 }
}
BENCHMARK_TEMPLATE(bm_min_length, Vec2DSlow);
BENCHMARK_TEMPLATE(bm_min_length, Vec2D);
BENCHMARK_MAIN();
```

Running this benchmark on an Intel i7 CPU generated the following results:

- Using unoptimized `Vec2DSlow` with `std::sqrt()` took 7,900 ns
- Using `Vec2D` with `LengthProxy` took 1,800 ns

This performance win corresponds to a speedup of more than 4x.

This was one example of how we can avoid computations that are not necessary in some situations. But instead of making the interface of `Vec2D` more complicated, we managed to encapsulate the optimization inside the proxy object so that all clients could benefit from the optimization, without sacrificing clarity.

A related technique for optimizing expressions in C++ is **expression templates**. This uses template metaprogramming to generate expression trees at compile time. The technique can be used for avoiding temporaries and to enable lazy evaluation. Expression templates is one of the techniques that makes linear algebra algorithms and matrix operations fast in Boost **Basic Linear Algebra Library** (**uBLAS**) and **Eigen**, **http://eigen.tuxfamily.org**. You can read more about how expression templates and fused operations can be used when designing a matrix class in *The C++ Programming Language*, *4th Edition*, by Bjarne Stroustrup.

We will end this chapter by looking at other ways to benefit from proxy objects when they are combined with overloaded operators.

Creative operator overloading and proxy objects

As you might already know, C++ has the ability to overload several operators, including the standard math operators such as plus and minus. Overloaded math operators can be utilized to create custom math classes that behave as numeric built-in types to make the code more readable. Another example is the stream operator, which in the standard library is overloaded in order to convert the objects to streams, as shown here:

```
std::cout << "iostream " << "uses " << "overloaded " << "operators.";
```

Some libraries, however, use overloading in other contexts. The Ranges library, as discussed earlier, uses overloading to compose views like this:

```
const auto r = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
auto odd_positive_numbers = r
  | std::views::filter([](auto v) { return v > 0; })
  | std::views::filter([](auto v) { return (v % 2) == 1; });
```

Next, we will explore how to use the pipe operator with proxy classes.

The pipe operator as an extension method

Compared to other languages, for example, C#, Swift, and JavaScript, C++ does not support extension methods; that is, you cannot extend a class locally with a new member function.

For example, you cannot extend `std::vector` with a `contains(T val)` function to be used like this:

```
auto numbers = std::vector{1, 2, 3, 4};
auto has_two = numbers.contains(2);
```

However, you can overload the pipe operator to achieve this, almost equivalent, syntax:

```
auto has_two = numbers | contains(2);
```

By using a proxy class, it's possible to accomplish this without much trouble.

The pipe operator

Our goal here is to implement a simple pipe operator so that we can write the following:

```
auto numbers = std::vector{1, 3, 5, 7, 9};
auto seven = 7;
bool has_seven = numbers | contains(seven);
```

The `contains()` function used with a pipeable syntax has two arguments: `numbers` and `seven`. Since the left argument, `numbers`, could be anything, we need the overload to contain something unique on the right-hand side. So, we create a `struct` template named `ContainsProxy`, which holds onto the argument on the right-hand side; this way, the overloaded pipe operator can recognize the overload:

```
template <typename T>
struct ContainsProxy { const T& value_; };
template <typename Range, typename T>
auto operator|(const Range& r, const ContainsProxy<T>& proxy) {
 const auto& v = proxy.value_;
 return std::find(r.begin(), r.end(), v) != r.end();
}
```

Now we can use `ContainsProxy` like this:

```
auto numbers = std::vector{1, 3, 5, 7, 9};
auto seven = 7;
```

```
auto proxy = ContainsProxy<decltype(seven)>{seven};
bool has_seven = numbers | proxy;
```

The pipe operator works, although the syntax is still ugly as we need to specify the type. In order to make the syntax neater, we can simply make a convenience function that takes the value and creates a proxy containing the type:

```
template <typename T>
auto contains(const T& v) { return ContainsProxy<T>{v}; }
```

That's all we need; we can now use it for any type or container:

```
auto penguins = std::vector<std::string>{"Ping","Roy","Silo"};
bool has_silo = penguins | contains("Silo");
```

The example covered in this section show a rudimentary approach to implementing the pipe operator. Libraries such as the Ranges library and the Fit library by Paul Fultz, available at **https://github.com/pfultz2/Fit**, implement adaptors that take a regular function and give it the ability to be invoked using the pipe syntax.

Summary

In this chapter, you learned the difference between lazy evaluation and eager evaluation. You also learned how to use hidden proxy objects to implement lazy evaluation behind the scenes, meaning that

you now understand how to implement lazy evaluation optimizations while preserving the easy-to-use interface of your classes. Hiding complex optimizations inside library classes instead of having them exposed in the application code makes the application code more readable and less error-prone.

In the next chapter, we will shift focus and move on to concurrent and parallel programming using C++.