2
Essential C++ Techniques

In this chapter, we will take an in-depth look at some fundamental C++ techniques, such as move semantics, error handling, and lambda expressions, that will be used throughout this book. Some of these concepts still confuse even experienced C++ programmers and therefore we will look into both their use cases and how they work under the hood.

This chapter will cover the following topics:

- Automatic type deduction and how to use the `auto` keyword when declaring functions and variables.
- Move semantics and the *rule of five* and *rule of zero*.
- Error handling and contracts. Although these topics don't present anything that can be considered modern C++, both exceptions and contracts are highly debated areas within C++ today.
- Creating function objects using lambda expressions, one of the most important features from C++11.

Let's begin by taking a look at automatic type deduction.

Automatic type deduction with the auto keyword

Since the introduction of the `auto` keyword in C++11, there has been a lot of confusion in the C++ community about how to use the different flavors of `auto`, such as `const auto&`, `auto&`, `auto&&`, and `decltype(auto)`.

Using auto in function signatures

Although discouraged by some C++ programmers, in my experience the use of `auto` in function signatures can increase readability when browsing and viewing header files.

Here is how the `auto` syntax looks compared to the traditional syntax with explicit types:

Traditional syntax with explicit type:

New syntax with auto:

```
struct Foo {
 int val() const {   return m_;  } const int& cref() const {   return m_;  } int& mref() {   return m_;  } int m_{};};
```

```
struct Foo {
 auto val() const {   return m_;  }
```

The `auto` syntax can be used both with and without a trailing return type. The trailing return is necessary in some contexts. For example, if we are writing a virtual function, or the function declaration is put in a header file and the function definition is in a `.cpp` file.

Note that the `auto` syntax can also be used with free functions:

Return type	Syntactic variants (a, b, and c correspond to the same result):

Value	`auto val() const // a) auto, deduced type`
	`auto val() const -> int // b) auto, trailing type`
	`int val() const // c) explicit type`

Const reference	`auto& cref() const // a) auto, deduced type`
	`auto cref() const -> const int& // b) auto, trailing type`
	`const int& cref() const // c) explicit type`

Mutable reference	`auto& mref() // a) auto, deduced type`
	`auto mref() -> int& // b) auto, trailing type`
	`int& mref() // c) explicit type`

Forwarding the return type using decltype(auto)

There is a somewhat rare version of automatic type deduction called `decltype(auto)`. Its most common use is for forwarding the exact type from a function. Imagine that we are writing wrapper functions for `val()` and `mref()` declared in the previous table, like this:

```
int val_wrapper() { return val(); }    // Returns int
int& mref_wrapper() { return mref(); } // Returns int&
```

Now, if we wanted to use return type deduction for the wrapper functions, the `auto` keyword would deduce the return type to an `int` in both cases:

```
auto val_wrapper() { return val(); }   // Returns int
auto mref_wrapper() { return mref(); } // Also returns int
```

If we wanted our `mref_wrapper()` to return an `int&`, we would need to write `auto&`. In this example, this would be fine, since we know the return type of `mref()`. However, that's not always the case. So if we want the compiler to instead choose the exact same type without explicitly saying `int&` or `auto&` for `mref_wrapper()`, we can use `decltype(auto)`:

```
decltype(auto) val_wrapper() { return val(); }   // Returns int
decltype(auto) mref_wrapper() { return mref(); } // Returns int&
```

In this way, we can avoid explicitly choosing between writing `auto` or `auto&` when we don't know what the function `val()` or `mref()` return. This is a scenario that usually happens in generic code where the type of the function that is being wrapped is a template parameter.

Using auto for variables

The introduction of the `auto` keyword in C++11 has initiated quite a debate among C++ programmers. Many people think it reduces readability, or even that it makes C++ similar to a dynamically typed language. I tend to not participate in those debates, but my personal opinion is that you should (almost) always use `auto` as, in my experience, it makes the code safer and less littered with clutter.

Overusing `auto` can make the code harder to understand. When reading code, we usually want to know which operations are supported by some object. A good IDE can provide us with this information, but it's not explicitly there in the source code. C++20 concepts address this issue by focusing on the behavior of an object. See *Chapter 8*, *Compile-Time Programming*, for more information about C++ concepts.

I prefer to use `auto` for local variables using the left-to-right initialization style. This means keeping the variable on the left, followed by an equals sign, and then the type on the right side, like this:

```
auto i = 0;
auto x = Foo{};
auto y = create_object();
auto z = std::mutex{};    // OK since C++17
```

With *guaranteed copy elision* introduced in C++17, the statement `auto x = Foo{}` is identical to `Foo x{}`; that is, the language guarantees that there is no temporary object that needs to be moved or copied in this case. This means that we can now use the left-to-right initialization style without worrying about performance and we can also use it for non-movable/non-copyable types, such as `std::atomic` or `std::mutex`.

One big advantage of using `auto` for variables is that you will never leave a variable uninitialized since `auto x;` doesn't compile. Uninitialized variables are a particularly common source of undefined behavior that you can completely eliminate by following the style suggested here.

Using `auto` will help you with using the correct type for your variables. What you still need to do, though, is to express how you intend to use a variable by specifying whether you need a reference or a copy, and whether you want to modify the variable or just read from it.

A const reference

A `const` reference, denoted by `const auto&`, has the ability to bind to anything. The original object can never be mutated through such a reference. I believe that the `const` reference should be the default choice for objects that are potentially expensive to copy.

If the `const` reference is bound to a temporary object, the lifetime of the temporary will be extended to the lifetime of the reference. This is demonstrated in the following example:

```
void some_func(const std::string& a, const std::string& b) {
 const auto& str = a + b;  // a + b returns a temporary
 // ...
} // str goes out of scope, temporary will be destroyed
```

It's also possible to end up with a `const` reference by using `auto&`. This can be seen in the following example:

```
auto foo = Foo{};
auto& cref = foo.cref(); // cref is a const reference
auto& mref = foo.mref(); // mref is a mutable reference
```

Even though this is perfectly valid, it is preferable to always explicitly express that we are dealing with `const` references by using `const auto&`, and, more importantly, we should use `auto&` to *only* denote mutable references.

A mutable reference

In contrast to a `const` reference, a mutable reference cannot bind to a temporary. As mentioned, we use `auto&` to denote mutable references. Use a mutable reference only when you intend to change the object it references.

A forwarding reference

`auto&&` is called a forwarding reference (also referred to as a *universal reference*). It can bind to anything, which makes it useful for certain cases. Forwarding references will, just like `const` references, extend the lifetime of a temporary. But in contrast to the `const` reference, `auto&&` allows us to mutate objects it references, temporaries included.

Use `auto&&` for variables that you only forward to some other code. In those forwarding cases, you rarely care about whether the variable is a `const` or a mutable; you just want to pass it to some code that is actually going to use the variable.

It's important to note that `auto&&` and `T&&` are only forwarding references if used in a function template where `T` is a template parameter of that function template. Using the `&&` syntax with an explicit type, for example `std::string&&`, denotes an **rvalue** reference and does not have the properties of a forwarding reference (rvalues and move semantics will be discussed later in this chapter).

Practices for ease of use

Although this is my personal opinion, I recommend using `const auto` for fundamental types (`int`, `float`, and so on) and small non-fundamental types like `std::pair` and `std::complex`. For bigger types that are potentially expensive to copy, use `const auto&`. This should cover the majority of the variable declarations in a C++ code base.

`auto&` and `auto` should only be used when you require the behavior of a mutable reference or an explicit copy; this communicates to the reader of the code that those variables are important as they either copy an object or mutate a referenced object. Finally, use `auto&&` for forwarding code only.

Following these rules makes your code base easier to read, debug, and reason about.

It might seem odd that while I recommend using `const auto` and `const auto&` for most variable declarations, I tend to use a simple `auto` in some places in this book. The reason for using plain `auto` is the limited space that the format of a book provides.

Before moving on, we will spend a little time talking about `const` and how to propagate `const` when using pointers.

Const propagation for pointers

By using the keyword `const`, we can inform the compiler about which objects are immutable. The compiler can then check that we don't try to mutate objects that aren't intended to be changed. In other words, the compiler checks our code for `const`-correctness. A common mistake when writing `const`-correct code in C++ is that a `const`-initialized object can still manipulate the values that member pointers point at. The following example illustrates the problem:

```
class Foo {
public:
  Foo(int* ptr) : ptr_{ptr} {}
  auto set_ptr_val(int v) const {
    *ptr_ = v; // Compiles despite function being declared const!
  }
private:
  int* ptr_{};
};
int main() {
  auto i = 0;
  const auto foo = Foo{&i};
  foo.set_ptr_val(42);
}
```

Although the function `set_ptr_val()` is mutating the `int` value, it's valid to declare it `const` since the pointer `ptr_` itself is not mutated, only the `int` object that the pointer is pointing at.

In order to prevent this in a readable way, a wrapper called `std::experimental::propagate_const` has been added to the standard library extensions (included in, at the time of writing, the latest versions of Clang and GCC). Using `propagate_const`, the function `set_ptr_val()` will not compile. Note that `propagate_const` only applies to pointers, and pointer-like classes such as `std::shared_ptr` and `std::unique_ptr`, but not `std::function`.

The following example demonstrates how `propagate_const` can be used to generate compilation errors when trying to mutate an object inside a `const` function:

```
#include <experimental/propagate_const>
class Foo {
public:
  Foo(int* ptr) : ptr_{ptr} {}
  auto set_ptr(int* p) const {
    ptr_ = p;  // Will not compile, as expected
  }
  auto set_val(int v) const {
    val_ = v;  // Will not compile, as expected
  }
  auto set_ptr_val(int v) const {
    *ptr_ = v; // Will not compile, const is propagated
  }
private:
  std::experimental::propagate_const<int*> ptr_ = nullptr;
  int val_{};
};
```

The importance of proper use of `const` in large code bases cannot be overstated, and the introduction of `propagate_const` makes `const`-correctness even more effective.

Next, we will have a look at move semantics and some important rules for handling resources inside a class.

Move semantics explained

Move semantics is a concept introduced in C++11 that, in my experience, is quite hard to grasp, even by experienced programmers. Therefore, I will try to give you an in-depth explanation of how it works, when the compiler utilizes it, and, most importantly, why it is needed.

Essentially, the reason C++ even has the concept of move semantics, whereas most other languages don't, is a result of it being a value-based language, as discussed in *Chapter 1*, *A Brief Introduction to C++*. If C++ did not have move semantics built in, the advantages of value-based semantics would get lost in many cases and programmers would have to perform one of the following trade-offs:

- Performing redundant deep-cloning operations with high performance costs
- Using pointers for objects like Java does, losing the robustness of value semantics

- Performing error-prone swapping operations at the cost of readability

We do not want any of these, so let's have a look at how move semantics help us.

Copy-construction, swap, and move

Before we go into the details of move, I will first explain and illustrate the differences between copy-constructing an object, swapping two objects, and move-constructing an object.

Copy-constructing an object

When copying an object handling a resource, a new resource needs to be allocated, and the resource from the source object needs to be copied so that the two objects are completely separated. Imagine that we have a class, `Widget`, that references some sort of resource that needs to be allocated on construction. The following code default-constructs a `Widget` object and then copy-constructs a new instance:

```
auto a = Widget{};
auto b = a;      // Copy-construction
```

The resource allocations that are carried out are illustrated in the following figure:

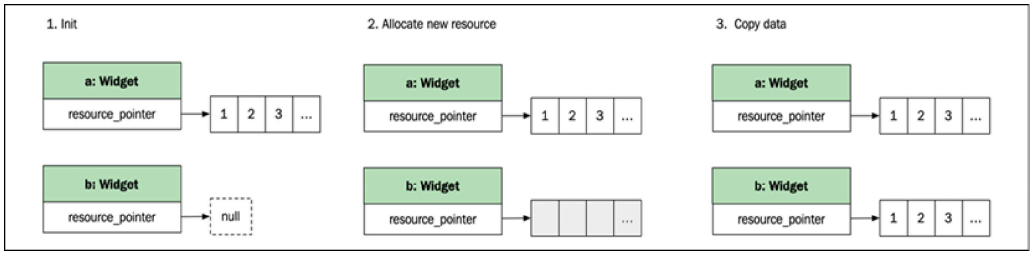

Figure 2.1: Copying an object with resources

The allocation and copying are slow processes, and, in many cases, the source object isn't needed anymore. With move semantics, the compiler detects cases like these where the old object is not tied to a variable, and instead performs a move operation.

Swapping two objects

Before move semantics were added in C++11, swapping the content of two objects was a common way to transfer data without allocating and copying. As shown next, objects simply swap their content with each

other:

```
auto a = Widget{};
auto b = Widget{};
std::swap(a, b);
```

The following figure illustrates the process:

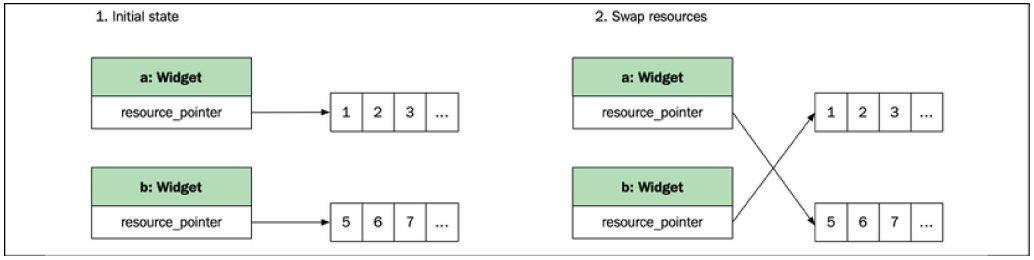

Figure 2.2: Swapping resources between two objects

The `std::swap()` function is a simple but useful utility used in the copy-and-swap idiom covered later in this chapter.

Move-constructing an object

When moving an object, the destination object steals the resource straight from the source object, and the source object is reset.

As you can see, it is very similar to swapping, except that the *moved-from* object does not have to receive the resources from the *moved-to* object:

```
auto a = Widget{};
auto b = std::move(a); // Tell the compiler to move the resource into b
```

The following figure illustrates the process:

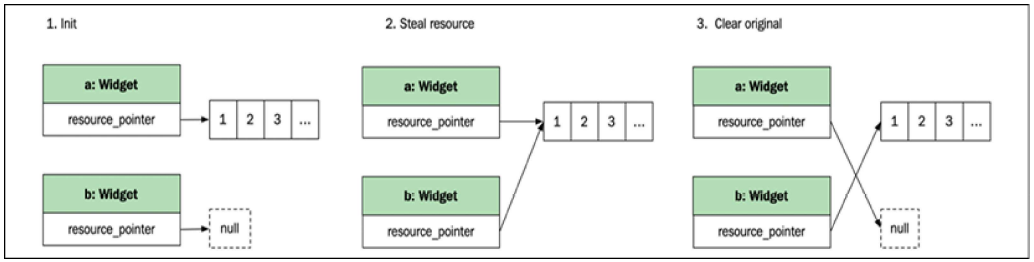

Figure 2.3: Moving resources from one object to another

Although the source object is reset, it's still in a valid state. This resetting of the source object is not something that the compiler does automatically for us. Instead, we need to implement the resetting in the move constructor to ensure that the object is in a valid state that can be destroyed or assigned to. We will talk more about valid states later on in this chapter.

Moving objects only makes sense if the object type owns a resource of some sort (the most common case being heap-allocated memory). If all data is contained within the object, the most efficient way to move an object is to just copy it.

Now that you have a basic grasp of move semantics, let's go into the details.

Resource acquisition and the rule of five

To fully understand move semantics, we need to go back to the basics of classes and resource acquisition in C++. One of the basic concepts in C++ is that a class should completely handle its resources. This means that when a class is copied, moved, copy-assigned, move-assigned, or destructed, the class should make sure its resources are handled accordingly. The necessity of implementing these five functions is commonly referred to as **the rule of five**.

Let's have a look at how the rule of five can be implemented in a class handling an allocated resource. In the `Buffer` class defined in the following code snippet, the allocated resource is an array of `float`s pointed at by the raw pointer `ptr_`:

```
class Buffer {
public:
 // Constructor
 Buffer(const std::initializer_list<float>& values)       : size_{values.size()} {
   ptr_ = new float[values.size()];
   std::copy(values.begin(), values.end(), ptr_);
 }
```

```
  auto begin() const { return ptr_; }
  auto end() const { return ptr_ + size_; }
  /* The 5 special functions are defined below */
private:
  size_t size_{0};
  float* ptr_{nullptr};
};
```

In this case, the handled resource is a block of memory allocated in the constructor of the `Buffer` class. Memory is probably the most common resource for classes to handle, but a resource can be so much more: a mutex, a handle for a texture on the graphics card, a thread handle, and so on.

The five functions that are referred to in the rule of five have been left out and will follow next. We will begin with the copy-constructor, copy-assignment, and destructor, which all need to be involved in the resource handling:

```
// 1. Copy constructor
Buffer::Buffer(const Buffer& other) : size_{other.size_} {
  ptr_ = new float[size_];
  std::copy(other.ptr_, other.ptr_ + size_, ptr_);
}
// 2. Copy assignment
auto& Buffer::operator=(const Buffer& other) {
  delete [] ptr_;
  ptr_ = new float[other.size_];
  size_ = other.size_;
  std::copy(other.ptr_, other.ptr_ + size_, ptr_);
  return *this;
}
// 3. Destructor
Buffer::~Buffer() {
  delete [] ptr_; // OK, it is valid to delete a nullptr
  ptr_ = nullptr;
}
```

Before the introduction of move semantics in C++11, these three functions were usually referred to as the **rule of three**. The copy-constructor, copy-assignment, and destructor are invoked in the following cases:

```
auto func() {
  // Construct
  auto b0 = Buffer({0.0f, 0.5f, 1.0f, 1.5f});
  // 1. Copy-construct
```

```
  auto b1 = b0;
  // 2. Copy-assignment as b0 is already initialized
  b0 = b1;
} // 3. End of scope, the destructors are automatically invoked
```

Although a correct implementation of these three functions is all that is required for a class to handle its internal resources, two problems arise:

- **Resources that cannot be copied**: In the `Buffer` class example, our resource can be copied, but there are other types of resources where a copy wouldn't make sense. For example, the resource contained in a class might be a `std::thread`, a network connection, or something else that it's not possible to copy. In these cases, we cannot pass around the object.
- **Unnecessary copies**: If we return our `Buffer` class from a function, the entire array needs to be copied. (The compiler optimizes away the copy in some cases, though, but let's ignore that for now.)

The solution to these problems is move semantics. In addition to the copy-constructor and copy-assignment, we can add a move-constructor and a move-assignment operator to our class. Instead of taking a `const` reference (`const Buffer&`) as a parameter, the move versions accept a `Buffer&&` object.

The `&&` modifier indicates that the parameter is an object that we intend to move from instead of copying it. Speaking in C++ terms, this is called an rvalue, and we will talk a little bit more about those later.

Whereas the `copy()` functions copy an object, the move equivalents are intended to move resources from one object to another, freeing the moved-from object from the resource.

This is how we would extend our `Buffer` class with the move-constructor and move-assignment. As you can see, these functions will not throw any exceptions and can therefore be marked as `noexcept`. This is because, as opposed to the copy-constructor/copy-assignment, they do not allocate memory or do something that might throw exceptions:

```
// 4. Move constructor
Buffer::Buffer(Buffer&& other) noexcept    : size_{other.size_}, ptr_{other.ptr_} {
  other.ptr_ = nullptr;
  other.size_ = 0;
}
// 5. Move assignment
auto& Buffer::operator=(Buffer&& other) noexcept {
  ptr_ = other.ptr_;
  size_ = other.size_;
  other.ptr_ = nullptr;
  other.size_ = 0;
```

```
  return *this;
}
```

Now, when the compiler detects that we perform what seems to be a copy, such as returning a `Buffer` from a function, but the copied-from value isn't used anymore, it will utilize the no-throw move-constructor/move-assignment instead of copying.

This is pretty sweet; the interface remains as clear as when copying but, under the hood, the compiler has performed a simple move. Thus, the programmer does not need to use any esoteric pointers or out-parameters in order to avoid a copy; as the class has move semantics implemented, the compiler handles this automatically.

Do not forget to mark your move-constructors and move-assignment operators as `noexcept` (unless they might throw an exception, of course). Not marking them `noexcept` prevents standard library containers and algorithms from utilizing them, instead resorting to using a regular copy/assignment under certain conditions.

To be able to know when the compiler is allowed to move an object instead of copying, an understanding of rvalues is necessary.

Named variables and rvalues

So, when is the compiler allowed to move objects instead of copying? As a short answer, the compiler moves an object when the object can be categorized as an rvalue. The term **rvalue** might sound complicated, but in essence it is just an object that is not tied to a named variable, for either of the following reasons:

- It's coming straight out of a function
- We make a variable an rvalue by using `std::move()`

The following example demonstrates both of these scenarios:

```
// The object returned by make_buffer is not tied to a variable
x = make_buffer();  // move-assigned
// The variable "x" is passed into std::move()
y = std::move(x);   // move-assigned
```

I will also use the terms **lvalue** and **named variable** interchangeably in this book. An lvalue corresponds to objects that we can refer to by name in our code.

Now we will make this a little more advanced by using a member variable of type `std::string` in a class. The following `Button` class will serve as an example:

```
class Button {
public:
 Button() {}
 auto set_title(const std::string& s) {
  title_ = s;
 }
 auto set_title(std::string&& s) {
  title_ = std::move(s);
 }
 std::string title_;
};
```

We also need a free function returning a title and a `Button` variable:

```
auto get_ok() {
 return std::string("OK");
}
auto button = Button{};
```

Given these prerequisites, let's look at a few cases of copying and moving in detail:

- **Case 1**: `Button::title_` is copy-assigned because the `string` object is tied to the variable `str`:

  ```
  auto str = std::string{"OK"};
  button.set_title(str);          // copy-assigned
  ```

- **Case 2**: `Button::title_` is move-assigned because `str` is passed through `std::move()`:

  ```
  auto str = std::string{"OK"};
  button.set_title(std::move(str));  // move-assigned
  ```

- **Case 3**: `Button::title_` is move-assigned because the new `std::string` object is coming straight out of a function:

  ```
  button.set_title(get_ok());      // move-assigned
  ```

- **Case 4**: `Button::title_` is copy-assigned because the `string` object is tied to `s` (this is the same as *Case 1*):

```
auto str = get_ok();
button.set_title(str);          // copy-assigned
```

- **Case 5**: `Button::title_` is copy-assigned because `str` is declared `const` and therefore is not allowed to mutate:

```
const auto str = get_ok();
button.set_title(std::move(str));  // copy-assigned
```

As you can see, determining whether an object is moved or copied is quite simple. If it has a variable name, it is copied; otherwise, it is moved. If you are using `std::move()` to move a named object, the object cannot be declared `const`.

Default move semantics and the rule of zero

This section discusses automatically generated copy-assignment operators. It's important to know that the generated function does not have strong exception guarantees. Therefore, if an exception is thrown during the copy-assignment, the object might end up in a state where it is only partially copied.

As with the copy-constructor and copy-assignment, the move-constructor and move-assignment can be generated by the compiler. Although some compilers allow themselves to automatically generate these functions under certain conditions (more about this later), we can simply force the compiler to generate them by using the `default` keyword.

In the case of the `Button` class, which doesn't manually handle any resources, we can simply extend it like this:

```
class Button {
public:
  Button() {} // Same as before

  // Copy-constructor/copy-assignment
  Button(const Button&) = default;
  auto operator=(const Button&) -> Button& = default;
  // Move-constructor/move-assignment
```

```
  Button(Button&&) noexcept = default;
  auto operator=(Button&&) noexcept -> Button& = default;
  // Destructor
  ~Button() = default;
  // ...
};
```

To make it even simpler, if we do not declare *any* custom copy-constructor/copy-assignment or destructor, the move-constructors/move-assignments are implicitly declared, meaning that the first `Button` class actually handles everything:

```
class Button {
public:
  Button() {} // Same as before

  // Nothing here, the compiler generates everything automatically!
  // ...
};
```

It's easy to forget that adding just one of the five functions prevents the compiler from generating the other ones. The following version of the `Button` class has a custom destructor. As a result, the move operators are not generated, and the class will always be copied:

```
class Button {
public:
  Button() {}
  ~Button()
    std::cout << "destructed\n"
  }
  // ...
};
```

Let's see how we can use this insight into generated functions when implementing application classes.

Rule of zero in a real code base

In practice, the cases where you have to write your own copy/move-constructors, copy/move-assignments, and constructors should be very few. Writing your classes so that they don't require any of these special member functions to be explicitly written (or `default` - declared) is often referred to as **the rule**

of zero. This means that if a class in the application code base is required to have any of these functions written explicitly, that piece of code would probably be better off in the library part of your code base.

Later on in this book, we will discuss `std::optional`, which is a handy utility class for dealing with optional members when applying the rule of zero.

A note on empty destructors

Writing an empty destructor can prevent the compiler from implementing certain optimizations. As you can see in the following snippets, copying an array of a trivial class with an empty destructor yields the same (non-optimized) assembler code as copying with a handcrafted `for`-loop. The first version uses an empty destructor with `std::copy()`:

```
struct Point {
 int x_, y_;
 ~Point() {}    // Empty destructor, don't use!
};
auto copy(Point* src, Point* dst) {
 std::copy(src, src+64, dst);
}
```

The second version uses a `Point` class with no destructor but with a handcrafted `for`-loop:

```
struct Point {
 int x_, y_;
};
auto copy(Point* src, Point* dst) {
 const auto end = src + 64;
 for (; src != end; ++src, ++dst) {
  *dst = *src;
 }
}
```

Both versions generate the following x86 assembler, which corresponds to a simple loop:

```
 xor eax, eax
.L2:
 mov rdx, QWORD PTR [rdi+rax]
 mov QWORD PTR [rsi+rax], rdx
 add rax, 8
 cmp rax, 512
```

```
 jne .L2
 rep ret
```

However, if we remove the destructor or declare the destructor `default`, the compiler optimizes `std::copy()` to utilize `memmove()` instead of a loop:

```
struct Point {
 int x_, y_;
 ~Point() = default; // OK: Use default or no constructor at all
};
auto copy(Point* src, Point* dst) {
 std::copy(src, src+64, dst);
}
```

The preceding code generates the following x86 assembler, with the `memmove()` optimization:

```
 mov rax, rdi
 mov edx, 512
 mov rdi, rsi
 mov rsi, rax
 jmp memmove
```

The assembler was generated using GCC 7.1 in *Compiler Explorer*, which is available at **https://godbolt.org/**.

To summarize, use `default` destructors or no destructors at all in favor of empty destructors to squeeze a little bit more performance out of your application.

A common pitfall - moving non-resources

There is one common pitfall when using default-created move-assignments: classes that mix fundamental types with more advanced compound types. As opposed to compound types, fundamental types (such as `int`, `float`, and `bool`) are simply copied when moved, as they don't handle any resources.

When a simple type is mixed with a resource-owning type, the move-assignment becomes a mixture of move and copy.

Here is an example of a class that will fail:

```
class Menu {
public:
  Menu(const std::initializer_list<std::string>& items)      : items_{items} {}
  auto select(int i) {
    index_ = i;
  }
  auto selected_item() const {
    return index_ != -1 ? items_[index_] : "";
  }
  // ...
private:
  int index_{-1}; // Currently selected item
  std::vector<std::string> items_;
};
```

The `Menu` class will have undefined behavior if it's used like this:

```
auto a = Menu{"New", "Open", "Close", "Save"};
a.select(2);
auto b = std::move(a);
auto selected = a.selected_item(); // crash
```

The undefined behavior happens as the `items_` vector is moved and is therefore empty. The `index_`, on the other hand, is copied, and therefore still has the value `2` in the moved-from object `a`. When `selected_item()` is called, the function will try to access `items_` at index `2` and the program will crash.

In these cases, the move-constructor/assignment is better implemented by simply swapping the members, like this:

```
Menu(Menu&& other) noexcept {
  std::swap(items_, other.items_);
  std::swap(index_, other.index_);
}
auto& operator=(Menu&& other) noexcept {
  std::swap(items_, other.items_);
  std::swap(index_, other.index_);
  return *this;
}
```

This way, the `Menu` class can be safely moved while still preserving the no-throw guarantee. In *Chapter 8*, *Compile-Time Programming*, you will learn how to take advantage of reflection techniques in C++ in order to automate the process of creating move-constructor/assignment functions that swap the elements.

Applying the && modifier to class member functions

In addition to being applied to objects, you can also add the `&&` modifier to a member function of a class, just as you can apply a `const` modifier to a member function. As with the `const` modifier, a member function that has the `&&` modifier will only be considered by overload resolution if the object is an rvalue:

```
struct Foo {
  auto func() && {}
};
auto a = Foo{};
a.func();          // Doesn't compile, 'a' is not an rvalue
std::move(a).func(); // Compiles
Foo{}.func();      // Compiles
```

It might seem odd that anyone would ever want this behavior, but there are use cases. We will investigate one of them in *Chapter 10*, *Proxy Objects and Lazy Evaluation*.

Don't move when copies are elided anyway

It might be tempting to use `std::move()` when returning a value from a function, like this:

```
auto func() {
  auto x = X{};
  // ...
  return std::move(x);  // Don't, RVO is prevented
}
```

However, unless `x` is a move-only type, you shouldn't be doing this. This usage of `std::move()` prevents the compiler from using **return value optimization** (**RVO**) and thereby completely elides the copying of `x`, which is more efficient than moving it. So, when returning a newly created object by value, don't use `std::move()`; instead, just return the object:

```
auto func() {
  auto x = X{};
```

```
  // ...
  return x;  // OK
}
```

This particular example where a *named* object is elided is usually called **NRVO**, or **Named-RVO**. RVO and NRVO are implemented by all major C++ compilers today. If you want to read more about RVO and copy elision, you can find a detailed summary at **https://en.cppreference.com/w/cpp/language/copy_elision**.

Pass by value when applicable

Consider a function that converts a `std::string` to lowercase. In order to use the move-constructor where applicable, and the copy-constructor otherwise, it may seem like two functions are required:

```
// Argument s is a const reference
auto str_to_lower(const std::string& s) -> std::string {
  auto clone = s;
  for (auto& c: clone) c = std::tolower(c);
  return clone;
}
// Argument s is an rvalue reference
auto str_to_lower(std::string&& s) -> std::string {
  for (auto& c: s) c = std::tolower(c);
  return s;
}
```

However, by taking the `std::string` by value instead, we can write one function that covers both cases:

```
auto str_to_lower(std::string s) -> std::string {
  for (auto& c: s) c = std::tolower(c);
  return s;
}
```

Let's see why this implementation of `str_to_lower()` avoids unnecessary copying where possible. When passed a regular variable, shown as follows, the content of `str` is copy-constructed into `s` prior to the function call, and then move-assigned back to `str` when the functions returns:

```
auto str = std::string{"ABC"};
str = str_to_lower(str);
```

When passed an rvalue, as shown below, the content of `str` is move-constructed into `s` prior to the function call, and then move-assigned back to `str` when the function returns. Therefore, no copy is made through the function call:

```
auto str = std::string{"ABC"};
str = str_to_lower(std::move(str));
```

At first sight, it seems like this technique could be applicable to all parameters. However, this pattern is not always optimal, as you will see next.

Cases where pass-by-value is not applicable

Sometimes this pattern of accept-by-value-then-move is actually a pessimization. For example, consider the following class where the function `set_data()` will keep a copy of the argument passed to it:

```
class Widget {
  std::vector<int> data_{};
  // ...
public:
  void set_data(std::vector<int> x) {
    data_ = std::move(x);
  }
};
```

Assume we call `set_data()` and pass it an lvalue, like this:

```
auto v = std::vector<int>{1, 2, 3, 4};
widget.set_data(v);               // Pass an lvalue
```

Since we are passing a named object, `v`, the code will copy-construct a new `std::vector` object, `x`, and then move-assign that object into the `data_` member. Unless we pass an empty vector object to `set_data()`, the `std::vector` copy-constructor will perform a heap allocation for its internal buffer.

Now compare this with the following version of `set_data()` optimized for lvalues:

```
void set_data(const std::vector<int>& x) {
  data_ = x;  // Reuse internal buffer in data_ if possible
}
```

Here, there will only be a heap allocation inside the assignment operator if the capacity of the current vector, `data_`, is smaller than the size of the source object, `x`. In other words, the internal pre-allocated buffer of `data_` can be reused in the assignment operator in many cases and save us from an extra heap allocation.

If we find it necessary to optimize `set_data()` for lvalues and rvalues, it's better, in this case, to provide two overloads:

```
void set_data(const std::vector<int>& x) {
  data_ = x;
}
void set_data(std::vector<int>&& x) noexcept {
  data_ = std::move(x);
}
```

The first version is optimal for lvalues and the second version for rvalues.

Finally, we will now look at a scenario where we can safely pass by value without worrying about the pessimization just demonstrated.

Moving constructor parameters

When initializing class members in a constructor, we can safely use the pass-by-value-then-move pattern. During the construction of a new object, there is no chance that there are pre-allocated buffers that could have been utilized to avoid heap allocations. What follows is an example of a class with one `std::vector` member and a constructor to demonstrate this pattern:

```
class Widget {
  std::vector<int> data_;
public:
  Widget(std::vector<int> x)      // By value
    : data_{std::move(x)} {}    // Move-construct
  // ...
};
```

We will now shift our focus to a topic that cannot be considered *modern C++* but is frequently discussed even today.

Designing interfaces with error handling

Error handling is an important and often overlooked part of the interface of functions and classes. Error handling is a heavily debated topic in C++, but often the discussions tend to focus on exceptions versus some other error mechanism. Although this is an interesting area, there are other aspects of error handling that are even more important to understand before focusing on the actual implementation of error handling. Obviously, both exceptions and error codes have been used in numerous successful software projects, and it is not uncommon to stumble upon projects that combine the two.

A fundamental aspect of error handling, regardless of programming language, is to distinguish between **programming errors** (also known as bugs) and **runtime errors**. Runtime errors can be further divided into **recoverable runtime errors** and **unrecoverable runtime errors**. An example of an unrecoverable runtime error is *stack overflow* (see *Chapter 7*, *Memory Management*). When an unrecoverable error occurs, the program typically terminates immediately, so there is no point in signaling these types of errors. However, some errors might be considered recoverable in one type of application but unrecoverable in others.

An edge case that often comes up when discussing recoverable and unrecoverable errors is the somewhat unfortunate behavior of the C++ standard library when running out of memory. When your program runs out of memory, this is typically unrecoverable, yet the standard library (tries) to throw a `std::bad_alloc` exception when this happens. We will not spend time on unrecoverable errors here, but the talk *De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable* by Herb Sutter (**https://sched.co/SiVW**) is highly recommended if you want to dig deeper into this topic.

When designing and implementing an API, you should always reflect on what type of error you are dealing with, because errors from different categories should be handled in completely different ways. Deciding whether errors are programming errors or runtime errors can be done by using a methodology called **Design by Contract**; this is a topic that deserves a book on its own. However, I will here introduce the fundamentals, which are enough for our purposes.

There are proposals for adding language support for contracts in C++, but currently contracts haven't made it to the standard yet. However, many C++ APIs and guidelines assume that you know the basics about contracts because the terminology contracts use makes it easier to discuss and document interfaces of classes and functions.

Contracts

A **contract** is a set of rules between the caller of some function and the function itself (the callee). C++ allows us to explicitly specify some rules using the C++ type system. For example, consider the following function signature:

```
int func(float x, float y)
```

It specifies that `func()` is returning an integer (unless it throws an exception), and that the caller has to pass two floating-point values. However, it doesn't say anything about what floating-point values that are allowed. For instance, can we pass the value 0.0 or a negative value? In addition, there might be some required relationship between `x` and `y` that cannot easily be expressed using the C++ type system. When we talk about contracts in C++, we usually refer to the rules that exist between a caller and a callee that cannot easily be expressed using the type system.

Without being too formal, a few concepts related to Design by Contract will be introduced here in order to give you some terms that you can use to reason about interfaces and error handling:

- **A precondition** specifies the *responsibilities of the caller* of a function. There may be constraints on the parameters passed to the function. Or, if it's a member function, the object might have to be in a specific state before calling the function. For example, the precondition when calling `pop_back()` on a `std::vector` is that the vector is not empty. It's the responsibility of the *caller* of `pop_back()` to ensure that the vector is not empty.
- **A postcondition** specifies the *responsibilities of the function* upon returning. If it's a member function, in what state does the function leave the object? For example, the postcondition of `std::list::sort()` is that the elements in the list are sorted in ascending order.
- **An invariant** is a condition that should always hold true. Invariants can be used in many contexts. A *loop invariant* is a condition that must be true at the beginning of every loop iteration. Further, a *class invariant* defines the valid states of an object. For example, an invariant of `std::vector` is that `size() <= capacity()`. Explicitly stating the invariants around some code gives us a better understanding of the code. Invariants are also a tool that can be used when proving that some algorithm does what it's supposed to do.

Class invariants are very important; we will therefore spend some more time discussing what they are and how they affect the design of classes.

Class invariants

As mentioned, a **class invariant** defines the valid states of an object. It specifies the relationship between the data members inside a class. An object can temporarily be in an invalid state during the time a member function is being executed. The important thing is that the invariant is upheld whenever the function passes the control to some other code that can observe the state of the object. This can happen when the function:

- Returns
- Throws an exception
- Invokes a callback function

- Calls some other function that might observe the state of the currently calling object; a common scenario is when passing a reference to `this` to some other function

It's important to realize that the class invariant is an implicit part of the precondition and postcondition for every member function of a class. If a member function leaves an object in an invalid state, the postcondition has not been fulfilled. Similarly, a member function can always assume that the object is in a valid state when the function is called. The exception to this rule is the constructors and the destructor of a class. If we wanted to insert code to check that the class invariant holds true, we could do that at the following points:

```
struct Widget {
  Widget() {
    // Initialize object...
    // Check class invariant
  }
  ~Widget() {
    // Check class invariant
    // Destroy object...
  }
  auto some_func() {
    // Check precondition (including class invariant)
    // Do the actual work...
    // Check postcondition (including class invariant)
  }
};
```

The copy/move constructors and copy/move assignment operators were left out here, but they follow the same pattern as the constructor and `some_func()`, respectively.

When an object has been moved from, the object might be in some empty or reset state. This is also a valid state of the object and is therefore part of the class invariant. However, usually there are only a few member functions that can be called when the object is in this state. For example, you cannot call `push_back()`, `empty()`, or `size()` on a `std::vector` that has been moved from, but you can call `clear()`, which will put the vector in a state where it is ready to be used again.

You should be aware, though, that this extra reset state makes the class invariant weaker and less useful. To avoid this state completely, you should implement your classes in such a way so that moved-from objects are reset to the state the object would have after default construction. My recommendation is to always do this, except in the very rare cases where resetting the moved-from state to the default state carries an unacceptable performance penalty. In that way, you can reason much better about moved-from states, and the class is safer to use because calling member functions on that object is fine.

If you can ensure that an object is always in a valid state (the class invariant holds true), you are likely to have a class that is hard to misuse, and if you have bugs in the implementation, they will usually be easy to spot. The last thing you want is to find a class in your code base and wonder whether some behavior of that class is a bug or a feature. Violation of a contract is always a serious bug.

In order to be able to write meaningful class invariants, we are required to write classes with high cohesion and with few possible states. If you have ever written a unit test for a class that you have authored yourself, you have probably noticed that while writing the unit test, it became clear that the API could be improved from the initial version. A unit test forces you to use and reflect on the interface of the class rather than the implementation details. In the same way, a class invariant makes you think about all the valid states an object could be in. If you find it hard to define a class invariant, it's usually because your class has too many responsibilities and handles too many states. Therefore, defining class invariants usually means that you end up with well-designed classes.

Maintaining contracts

Contracts are parts of the API that you design and implement. But how do you maintain and communicate a contract to the clients using your API? C++ has no built-in support for contracts yet, but there is ongoing work to add it to future versions of C++. There are some options, though:

- Use a library such as Boost.Contract.
- Document the contracts. This has the disadvantage that the contracts are not checked when running the program. Also, documentation tends to be outdated when the code changes.
- Use `static_assert()` and the `assert()` macro defined in `<cassert>`. Asserts are portable, standard C++.
- Build a custom library with custom macros similar to asserts but with better control of the behavior of failed contracts.

In this book, we will use asserts, one of the most primitive ways of checking for contract violations. Still, asserts can be very effective and have an enormous impact on code quality.

Enabling and disabling asserts

Technically, we have two standard ways to assert things in C++: using `static_assert()` or the `assert()` macro from the `<cassert>` header. `static_assert()` is validated during the compilation of the code, and therefore requires an expression that can be checked during compile time rather than runtime. A failed `static_assert()` results in a compilation error.

For asserts that can only be evaluated during runtime, you need to use the `assert()` macro instead. The `assert()` macro is a runtime check that is typically active during debugging and testing, and completely

disabled when the program is built in release mode. The `assert()` macro is typically defined something like this:

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /* implementation defined */
#endif
```

This means that you can completely remove all the asserts and the code for checking the conditions by defining `NDEBUG`.

Now, with some terminology from Design by Contract under your belt, let's focus on contract violations (errors) and how to handle them in your code.

Error handling

The first thing to do when designing APIs with proper error handling is to distinguish between programming errors and runtime errors. So, before we dive into error handling strategies, we will use Design by Contract to define what type of error we are dealing with.

Programming error or runtime error?

If we find a violation of a contract, we have also found an error in our program. For example, if we can detect that someone is calling `pop_back()` on an empty vector, we know that there is at least one bug in our source code that needs to be fixed. Whenever a precondition is not met, we know we are dealing with a *programming error*.

On the other hand, if we have a function that loads some record from disk and cannot return the record because of a read error on the disk, then we have detected a *runtime error*:

```
auto load_record(std::uint32_t id) {
  assert(id != 0);         // Precondition
  auto record = read(id);    // Read from disk, may throw
  assert(record.is_valid()); // Postcondition
  return record;
}
```

The precondition is fulfilled, but the postcondition cannot be met because of something outside of our program. There is no bug in the source code, but the function cannot return the record found on disk be-

cause of some disk-related error. Since the postcondition cannot be fulfilled, a runtime error has to be reported back to the caller, unless the caller can recover from it itself by retrying and so on.

Programming errors (bugs)

In general, there is no point in writing code that signals and handles bugs in your code. Instead, use asserts (or some of the other alternatives mentioned previously) to make the developer aware of issues in the code. You should only use exceptions or error codes for recoverable runtime errors.

Narrowing the problem space by assumptions

An assert specifies what assumptions you, as the author of some code, have made. You can only guarantee that the code works as intended if all the asserts in your code hold true. This makes coding much easier because you can effectively limit the amount of cases that you need to handle. Asserts are also a tremendous help for your team when using, reading, and modifying code written by you. All the assumptions are clearly documented in the form of assert statements.

Finding bugs with asserts

A failed assert is always a serious bug. There are basically three options when you find an assert that fails during testing:

- The assert is correct, but the code is wrong (either because of a bug in the implementation of the function, or a bug on the call-site). In my experience, this is the most common case. Getting the asserts correct is usually easier than getting the code around them correct. Fix the code and test again.
- The code is correct, but the assert is wrong. Sometimes this happens and it is usually pretty uncomfortable if you are looking at old code. Changing or removing an assert that fails can be time consuming because you need to be 100% sure that the code actually works and understand why an old assert has suddenly started to fail. Usually, this is because of a new use case that the original authors did not think about.
- Both the assert and the code are wrong. This usually requires a redesign of the class or function. Maybe the requirements have changed, and the assumptions made by the programmer are no longer true. But don't despair; instead, you should be glad that those assumptions were explicitly written using asserts; now you know why the code is not working anymore.

Runtime asserts require testing, otherwise the asserts will not be exercised. Newly written code with many asserts usually breaks when testing. This doesn't mean that you are a bad programmer; it means that you have added meaningful asserts that catch some of the errors that otherwise could have made it to production. Also, bugs that make a test version of your program terminate are also likely to be fixed.

Performance impact

Having many runtime asserts in your code will most likely degrade the performance of your test builds. However, asserts are never meant to be used in the final version of your optimized program. If your asserts make your test build too slow to be usable, finding the set of asserts that slows down your code is usually easy to track in a profiler (see *Chapter 3*, *Analyzing and Measuring Performance*, for more info about profiling).

By having the release build of your program completely ignore all sorts of programming errors, your program will not spend time checking error states caused by bugs. Instead, your code will run faster and only spend time solving the actual problem it was meant to solve. It will only check for runtime errors that need to be recovered.

To summarize, programming errors should be detected when testing the program. There is no need to use exceptions or some other error handling mechanism for dealing with programming errors. Instead, a programming error should preferably log something meaningful and terminate the program to inform the programmer that the bug needs to be fixed. Following this guideline dramatically reduces the number of places we need to handle exceptions in our code. We will have better performance in our optimized build and hopefully fewer bugs since they have been detected by failed asserts. However, there are situations where errors can occur at runtime, and those errors need to be handled and recovered by the code we implement.

Recoverable runtime errors

If a function cannot uphold its part of the contract (the postcondition, that is), a runtime error has occurred and needs to be signaled to some place in the code that can handle it and recover the valid state. The purpose of handling recoverable errors is to pass an error from the place where the error occurred to the place where the valid state can be recovered. There are many ways to achieve this. There are two sides of this coin:

- For the signaling part we can choose between C++ exceptions, error codes, returning a `std::optional` or `std::pair`, or using `boost::outcome` or `std::experimental::expected`.
- Preserving the valid state of the program without leaking any resources. Deterministic destructors and automatic storage duration are the tools that make this possible in C++.

The utility classes `std::optional` and `std::pair` will be covered in *Chapter 9*, *Essential Utilities*. We will now focus on C++ exceptions and how to avoid leaking resources when recovering from an error.

Exceptions

Exceptions are the standard error handling mechanism provided by C++. The language was designed to be used with exceptions. One example of this is constructors that fail; the only way to signal errors from

constructors is by using exceptions.

In my experience, exceptions are used in many different ways. One reason for this is that distinct applications can have vastly different requirements when dealing with runtime errors. With some applications, such as a pacemaker or a power plant control system, which may have a severe impact if they crash, we may have to deal with every possible exceptional circumstance, such as running out of memory, and keep the application in a running state. Some applications even completely stay away from using the heap memory, either because the platform doesn't have any heap available at all, or because the heap introduces an uncontrollable level of uncertainty as the mechanics of allocating new memory are out of the application's control.

I assume that you already know the syntax of throwing and catching exceptions and will not cover it here. A function that is guaranteed to not throw an exception can be marked as `noexcept`. It's important to understand that the compiler does *not* verify this; instead, it is up to the author of the code to figure out whether their function could throw an exception.

A function marked with `noexcept` makes it possible for the compiler to generate faster code in some cases. If an exception would be thrown from a function marked with `noexcept`, the program will call `std::terminate()` instead of unwinding the stack. The following code demonstrates how to mark a function as not throwing:

```
auto add(int a, int b) noexcept {
  return a + b;
}
```

You may notice that many code examples in this book don't use `noexcept` (or `const`) even if it would have been appropriate in production code. This is only because of the format of a book; it would make the code hard to read to add `noexcept` and `const` at all the places that I normally would.

Preserving the valid state

Exception handling requires us programmers to think about exception safety guarantees; that is, what is the program state before and after an exception has occurred? Strong exception safety can be seen as a transaction. A function either commits all state changes, or performs a complete rollback in the case of an exception.

To make this a bit more concrete, let's take a look at the following simple function:

```
void func(std::string& str) {
  str += f1();  // Could throw
```

```
  str += f2();  // Could throw
}
```

The function appends the result of `f1()` and `f2()` to the string, `str`. Now consider what would happen if an exception was thrown when calling the function `f2()`; only the result from `f1()` would be appended to `str`. What we want instead is to have `str` untouched if an exception occurs. This can be fixed by using an idiom called **copy-and-swap**. It means that we perform the operations that might throw exceptions on temporary copies before we let the application's state be modified by non-throwing `swap()` functions:

```
void func(std::string& str) {
  auto tmp = std::string{str};  // Copy
  tmp += f1();                  // Mutate copy, may throw
  tmp += f2();                  // Mutate copy, may throw
  std::swap(tmp, str);          // Swap, never throws
}
```

The same pattern can be used in member functions to preserve the valid state of an object. Let's say we have a class with two data members and a class invariant that says that the data members cannot compare equal, as follows:

```
class Number { /* ... */ };
class Widget {
public:
  Widget(const Number& x, const Number& y) : x_{x}, y_{y} {
    assert(is_valid());         // Check class invariant
  }
private:
  Number x_{};
  Number y_{};
  bool is_valid() const {       // Class invariant
    return x_ != y_;            // x_ and y_ must not be equal
  }
};
```

Next, assume we are adding a member function that updates both data members, like this:

```
void Widget::update(const Number& x, const Number& y) {
  assert(x != y && is_valid());  // Precondition
  x_ = x;
  y_ = y;
```

```
  assert(is_valid());          // Postcondition
}
```

The precondition states that `x` and `y` must not compare equal. If the assignment of `x_` and `y_` can throw, `x_` might be updated but not `y_`. This may result in a broken class invariant; that is, an object in an invalid state. We want the function to preserve the valid state of the object it had before the assignment operations if an error occurs. Again, one possible solution is to use the copy-and-swap idiom:

```
void Widget::update(const Number& x, const Number& y) {
  assert(x != y && is_valid());    // Precondition
  auto x_tmp = x;
  auto y_tmp = y;
  std::swap(x_tmp, x_);
  std::swap(y_tmp, y_);
  assert(is_valid());          // Postcondition
}
```

First, local copies are created without modifying the state of the object. Then, if no exception has been thrown, the state of the object can be changed using a non-throwing `swap()`. The copy-and-swap idiom can also be used when implementing assignment operators to achieve strong exception safety guarantees.

Another important aspect of error handling is to avoid leaking resources when an error occurs.

Resource acquisition

The destruction of C++ objects is predictable, meaning that we have full control over when, and in what order, resources that we have acquired are released. This is further illustrated in the following example, where the mutex variable `m` is always unlocked when exiting the function, as the scoped lock releases it when we exit the scope, regardless of how and where we exit:

```
auto func(std::mutex& m, bool x, bool y) {
  auto guard = std::scoped_lock{m}; // Lock mutex
  if (x) {
    // The guard automatically releases the mutex at early exit
    return;
  }
  if (y) {
    // The guard automatically releases if an exception is thrown
    throw std::exception{};
  }
```

```
    // The guard automatically releases the mutex at function exit
}
```

Ownership, lifetime of objects, and resource acquisition are fundamental concepts in C++, and we will cover them in *Chapter 7*, *Memory Management*.

Performance

Unfortunately, exceptions have a bad reputation when it comes to performance. Some concerns are legitimate, whereas some are based on historical observations when exceptions were not implemented efficiently by the compilers. However, today there are two main reasons why people abandon exceptions:

- The size of the binary program is increased even if exceptions are not being thrown. Even though this is usually not an issue, it doesn't follow the zero-overhead principle since we are paying for something that we don't use.
- Throwing and catching exceptions is relatively expensive. The runtime cost of throwing and catching exceptions is not deterministic. This makes exceptions unsuitable in contexts with hard real-time requirements. In this case, other alternatives such as returning a `std::pair` with a return value and an error code might better.

On the other hand, exceptions perform outstandingly when no exceptions are being thrown; that is, when the program follows the success path. Other error reporting mechanisms such as error codes require checking return codes in `if-else` statements even when the program runs without any errors.

Exceptions should happen rarely, and typically when an exception occurs, the extra performance penalty that exception handling adds is usually not an issue in those situations. It's usually possible to perform computations that could potentially throw before or after some performance-critical code runs. In that way, we can avoid having exceptions thrown and caught at the places in our program where we cannot afford to have exceptions.

To make a fair comparison between exceptions and some other error reporting mechanism, it's important to specify what to compare. Sometimes exceptions are compared with no error handling at all, which is unfair; exceptions need to be compared with a mechanism that offers the same functionality, of course. Don't abandon exceptions for performance reasons before you have measured the impact they might have. You can read more about analyzing and measuring performance in the next chapter.

Now we will move away from error handling and explore how we can use lambda expressions to create function objects.

Function objects and lambda expressions

Lambda expressions, introduced in C++11, and further enhanced with every C++ version since, are one of the most useful features in modern C++. Their versatility comes not only from easily passing functions to algorithms, but also their use in a lot of circumstances where you need to pass the code around, especially as you can store a lambda in a `std::function`.

Although lambdas made these programming techniques vastly simpler to work with, everything mentioned in this section is possible to perform without them. A lambda—or, more formally, a lambda expression—is a convenient way of constructing a function object. But instead of using lambda expressions, we could instead implement classes with `operator()` overloaded, and then instantiate these to create function objects.

We will explore the lambda's similarities to these kinds of classes later, but first I will introduce lambda expressions in a simple use case.

The basic syntax of a C++ lambda

In a nutshell, lambdas enable programmers to pass functions to other functions, just as easily as a variable is passed.

Let's compare passing a lambda to an algorithm with passing a variable:

```
// Prerequisite
auto v = std::vector{1, 3, 2, 5, 4};

// Look for number three
auto three = 3;
auto num_threes = std::count(v.begin(), v.end(), three);
// num_threes is 1

// Look for numbers which is larger than three
auto is_above_3 = [](int v) { return v > 3; };
auto num_above_3 = std::count_if(v.begin(), v.end(), is_above_3);
// num_above_3 is 2
```

In the first case, we pass a variable to `std::count()`, and in the latter case we pass a function object to `std::count_if()`. This is a typical use case for lambdas; we pass a function to be evaluated many times by another function (in this case, `std::count_if()`).

Also, the lambda does not need to be tied to a variable; just as we can put a variable right into an expression, we can do the same with a lambda:

```
auto num_3 = std::count(v.begin(), v.end(), 3);
auto num_above_3 = std::count_if(v.begin(), v.end(), [](int i) {
  return i > 3;
});
```

The lambdas you have seen so far are called **stateless lambdas**; they don't copy or reference any variables from outside the lambda and therefore don't need any internal state. Let's make this a little more advanced by introducing **stateful lambdas** by using capture blocks.

The capture clause

In the previous example, we hard-coded the value `3` inside the lambda so that we always counted the numbers greater than three. What if we want to use external variables inside the lambda? What we do is capture the external variables by putting them in the **capture clause**; that is, the `[]` part of the lambda:

```
auto count_value_above(const std::vector<int>& v, int x) {
  auto is_above = [x](int i) { return i > x; };
  return std::count_if(v.begin(), v.end(), is_above);
}
```

In this example, we captured the variable `x` by copying it into the lambda. If we want to declare `x` as a reference, we put an `&` at the beginning, like this:

```
auto is_above = [&x](int i) { return i > x; };
```

The variable is now merely a reference to the outer `x` variable, just like a regular reference variable in C++. Of course, we need to be very cautious about the lifetime of objects we pass by reference into a lambda since the lambda might execute in a context where the referenced objects have ceased to exist. It's therefore safer to capture by value.

Capture by reference versus capture by value

Using the capture clause for referencing and copying variables works just like regular variables. Take a look at these two examples and see if you can spot the difference:

Capture by value	Capture by reference

```
auto func() {
 auto vals = {1,2,3,4,5,6};
 auto x = 3;
 auto is_above = [x](int v) {
  return v > x;
 };
 x = 4;
 auto count_b = std::count_if(
  vals.begin(),
  vals.end(),
  is_above
 );  // count_b equals 3 }
```

```
auto func() {
 auto vals = {1,2,3,4,5,6};
 auto x = 3;
 auto is_above = [&x](int v) {
  return v > x;
 };
 x = 4;
 auto count_b = std::count_if(
  vals.begin(),
  vals.end(),
  is_above
 );  // count_b equals 2 }
```

In the first example, `x` was *copied* into the lambda and was therefore not affected when `x` was mutated; consequently `std::count_if()` counts the number of values above 3.

In the second example, `x` was *captured by reference*, and therefore `std::count_if()` instead counts the number of values above 4.

Similarities between a lambda and a class

I mentioned earlier that lambda expressions generate function objects. A function object is an instance of a class that has the call operator, `operator()()`, defined.

To understand what a lambda expression consists of, you can view it as a regular class with restrictions:

- The class only consists of one member function
- The capture clause is a combination of the class' member variables and its constructor

The following table shows lambda expressions and the corresponding classes. The left column uses *capture by value* and the right column *capture by reference*:

A lambda with capture by value...	A lambda with capture

```
auto x = 3;auto is_above = [x](int y) { return y > x;};auto test = is_above(5);
```

...corresponds to this class:

```
auto x = 3;class IsAbove {
public: IsAbove(int x) : x{x} {} auto operator()(int y) const {   return y > x; }private: int x{}; // Value };auto is_above = IsAbove{x};
auto test = is_above(5);
```

Thanks to lambda expressions, we don't have to manually implement these function object types as classes.

Initializing variables in capture

As seen in the previous example, the capture clause initializes member variables in the corresponding class. This means that we can also initialize member variables inside a lambda. These variables will only be visible from inside the lambda. Here is an example of a lambda that initializes a capture variable called `numbers`:

```
auto some_func = [numbers = std::list<int>{4,2}]() {
 for (auto i : numbers)
   std::cout << i;
};
some_func();  // Output: 42
```

The corresponding class would look something like this:

```
class SomeFunc {
public:
 SomeFunc() : numbers{4, 2} {}
 void operator()() const {
  for (auto i : numbers)
    std::cout << i;
 }
```

```
private:
 std::list<int> numbers;
};
auto some_func = SomeFunc{};
some_func(); // Output: 42
```

When initializing a variable inside a capture, you can imagine that there is a hidden `auto` keyword in front of the variable name. In this case, you can think about `numbers` as being defined like `auto numbers = std::list<int>{4, 2}`. If you want to initialize a reference, you can use an ampersand in front of the name, which would correspond to `auto&`. Here is an example:

```
auto x = 1;
auto some_func = [&y = x]() {
 // y is a reference to x
};
```

Again, you have to be very cautious about lifetimes when referencing (and not copying) objects outside the lambda.

It's also possible to move an object inside a lambda, which is necessary when using move-only types such as `std::unique_ptr`. Here is how it can be done:

```
auto x = std::make_unique<int>();
auto some_func = [x = std::move(x)]() {
 // Use x here..
};
```

This also demonstrates that it is possible to use the same name (`x`) for the variable. This is not necessary. Instead, we could have used some other name inside the lambda, for example `[y = std::move(x)]`.

Mutating lambda member variables

As the lambda works just like a class with member variables, it can also mutate them. However, the function call operator of a lambda is `const` by default, so we explicitly need to specify that the lambda can mutate its members by using the `mutable` keyword. In the following example, the lambda mutates the `counter` variable every time it's invoked:

```
auto counter_func = [counter = 1]() mutable {
 std::cout << counter++;
```

```
};
counter_func(); // Output: 1
counter_func(); // Output: 2
counter_func(); // Output: 3
```

If a lambda only captures variables by reference, we do not have to add the `mutable` modifier to the declaration, as the lambda itself doesn't mutate. The difference between mutable and non-mutable lambdas is demonstrated in the following code snippets:

Capture by value

```
auto some_func() {
 auto v = 7;
 auto lambda = [v]() mutable {
  std::cout << v << " ";
  ++v;
 };
 assert(v == 7);
 lambda();  lambda();
 assert(v == 7);
 std::cout << v;
}
```

Output: `7 8 7`

Capture by reference

```
auto some_func() {
 auto v = 7;
 auto lambda = [&v]() {
  std::cout << v << " ";
  ++v;
 };
 assert(v == 7);
 lambda();
 lambda();
 assert(v == 9);
 std::cout << v;
}
```

Output: `7 8 9`

In the example to the right where `v` is captured by reference, the lambda will mutate the variable `v`, which is owned by the scope of `some_func()`. The mutating lambda in the left column will only mutate a copy of `v`, owned by the lambda itself. This is the reason why we will end up with different outputs in the two versions.

Mutating member variables from the compiler's perspective

To understand what's going on in the preceding example, take a look at how the compiler sees the previous lambda objects:

Capture by value

Capture by reference

```
class Lambda {
 public:
 Lambda(int m) : v{m} {}
 auto operator()() {
  std::cout<< v << " ";
  ++v;
 }
private:
 int v{};
};
```

```
class Lambda {
 public:
 Lambda(int& m) : v{m} {}
 auto operator()() const {
  std::cout<< v << " ";
  ++v;
 }
private:
 int& v;
};
```

As you can see, the first case corresponds to a class with a regular member, whereas the capture by reference case simply corresponds to a class where the member variable is a reference.

You might have noticed that we add the modifier `const` on the `operator()` member function of the capture by reference class, and we also do not specify `mutable` on the corresponding lambda. The reason this class is still considered `const` is that we do not mutate anything inside the actual class/lambda; the actual mutation applies to the referenced value, and therefore the function is still considered `const`.

Capture all

In addition to capturing variables one by one, all variables in the scope can be captured by simply writing `[=]` or `[&]`.

Using `[=]` means that every variable will be captured by value, whereas `[&]` captures all variables by reference.

If we use lambdas inside a member function, it is also possible to capture the entire object by reference using `[this]` or by copy by writing `[*this]`:

```
class Foo {
public:
 auto member_function() {
  auto a = 0;
  auto b = 1.0f;
  // Capture all variables by copy
  auto lambda_0 = [=]() { std::cout << a << b; };
```

```
  // Capture all variables by reference
  auto lambda_1 = [&]() { std::cout << a << b; };
  // Capture object by reference
  auto lambda_2 = [this]() { std::cout << m_; };
  // Capture object by copy
  auto lambda_3 = [*this]() { std::cout << m_; };
 }
private:
 int m_{};
};
```

Note that using `[=]` does not mean that all variables in the scope are copied into the lambda; only the variables actually used inside the lambda are copied.

When capturing all variables by value, you can specify variables to be captured by reference (and vice versa). The following table shows the result of different combinations in the capture block:

Capture block	Resulting capture types
`int a, b, c;auto func = [=] { /*...*/ };`	Capture `a`, `b`, `c` by value.
`int a, b, c;auto func = [&] { /*...*/ };`	Capture `a`, `b`, `c` by reference.
`int a, b, c;auto func = [=, &c] { /*...*/ };`	Capture `a`, `b` by value. Capture `c` by reference.
`int a, b, c;auto func = [&, c] { /*...*/ };`	Capture `a`, `b` by reference. Capture `c` by value.

Although it is convenient to capture all variables with `[&]` or `[=]`, I recommend capturing variables one by one, as it improves the readability of the code by clarifying exactly which variables are used inside the lambda scope.

Assigning C function pointers to lambdas

Lambdas without captures can be implicitly converted to function pointers. Let's say you are using a C library, or an older C++ library, that uses a callback function as a parameter, like this:

```
extern void download_webpage(const char* url,
                             void (*callback)(int, const char*));
```

The callback is called with a return code and some downloaded content. It is possible to pass a lambda as a parameter when calling `download_webpage()`. Since the callback is a regular function pointer, the lambda must not have any captures and you have to use a plus (`+`) in front of the lambda:

```
auto lambda = +[](int result, const char* str) {
  // Process result and str
};
download_webpage("http://www.packt.com", lambda);
```

This way, the lambda is converted into a regular function pointer. Note that the lambda cannot have any captures at all in order to use this functionality.

Lambda types

Since C++20, lambdas without captures are default-constructible and assignable. By using `decltype`, it's now easy to construct different lambda objects that have the same type:

```
auto x = [] {};   // A lambda without captures
auto y = x;       // Assignable
decltype(y) z;    // Default-constructible
static_assert(std::is_same_v<decltype(x), decltype(y)>); // passes
static_assert(std::is_same_v<decltype(x), decltype(z)>); // passes
```

However, this only applies to lambdas without captures. Lambdas *with* captures have their own unique type. Even if two lambda functions with captures are plain clones of each other, they still have their own unique type. Therefore, it's not possible to assign one lambda with captures to another lambda.

Lambdas and std::function

As mentioned in the previous section, lambdas with captures (stateful lambdas) cannot be assigned to each other since they have unique types, even if they look exactly the same. To be able to store and pass around lambdas with captures, we can use `std::function` to hold a function object constructed by a lambda expression.

The signature of a `std::function` is defined as follows:

```
std::function< return_type ( parameter0, parameter1...) >
```

So, a `std::function` returning nothing and having no parameters is defined like this:

```
auto func = std::function<void(void)>{};
```

A `std::function` returning a `bool` with an `int` and a `std::string` as parameters is defined like this:

```
auto func = std::function<bool(int, std::string)>{};
```

Lambda functions sharing the same signature (same parameters and same return type) can be held by the same type of `std::function` objects. A `std::function` can also be reassigned at runtime.

What is important here is that what is captured by the lambda does not affect its signature, and therefore both lambdas with and without captures can be assigned to the same `std::function` variable. The following code shows how different lambdas are assigned to the same `std::function` object called `func`:

```
// Create an unassigned std::function object
auto func = std::function<void(int)>{};
// Assign a lambda without capture to the std::function object
func = [](int v) { std::cout << v; };
func(12); // Prints 12
// Assign a lambda with capture to the same std::function object
auto forty_two = 42;
func = [forty_two](int v) { std::cout << (v + forty_two); };
func(12); // Prints 54
```

Let's put the `std::function` to use in something that resembles a real-world example next.

Implementing a simple Button class with std::function

Assume that we set out to implement a `Button` class. We can then use the `std::function` to store the action corresponding to clicking the button, so that when we call the `on_click()` member function, the corresponding code is executed.

We can declare the `Button` class like this:

```
class Button {
public:
  Button(std::function<void(void)> click) : handler_{click} {}
  auto on_click() const { handler_(); }
private:
  std::function<void(void)> handler_{};
};
```

We can then use it to create a multitude of buttons with different actions. The buttons can conveniently be stored in a container because they all have the same type:

```
auto create_buttons () {
  auto beep = Button([counter = 0]() mutable {
    std::cout << "Beep:" << counter << "! ";
    ++counter;
  });
  auto bop = Button([] { std::cout << "Bop. "; });
  auto silent = Button([] {});
  return std::vector<Button>{beep, bop, silent};
}
```

Iterating the list and calling `on_click()` on each button will execute the corresponding function:

```
const auto& buttons = create_buttons();
for (const auto& b: buttons) {
  b.on_click();
}
buttons.front().on_click(); // counter has been incremented
// Output: "Beep:0! Bop. Beep:1!"
```

The preceding example with buttons and click handlers demonstrates some of the benefits of using `std::function` in combination with lambdas; even though each stateful lambda will have its own unique

type, a single `std::function` type can wrap lambdas that share the same signature (return type and arguments).

As a side note, you might have noticed that the `on_click()` member function is declared `const`. However, it's mutating the member variable `handler_` by increasing the `counter` variable in one of the click handlers. This might seem like it breaks const-correctness rules, as a const member function of `Button` is allowed to call a mutating function on one of its class members. The reason it is allowed is the same reason that member pointers are allowed to mutate their pointed-to value in a const context. Earlier in this chapter, we discussed how to propagate constness for pointer data members.

Performance consideration of std::function

A `std::function` has a few performance losses compared to a function object constructed by a lambda expression directly. This section will discuss some of the things related to performance to consider when using `std::function`.

Prevented inline optimizations

When it comes to lambdas, the compiler has the ability to inline the function call; that is, the overhead of the function call is eliminated. The flexible design of `std::function` make it nearly impossible for the compiler to inline a function wrapped in a `std::function`. The prevention of inline optimizations can have a negative impact on the performance if small functions wrapped in `std::function` are called very frequently.

Dynamically allocated memory for captured variables

If a `std::function` is assigned to a lambda with captured variables/references, the `std::function` will, in most cases, use heap-allocated memory to store the captured variables. Some implementations of `std::function` do not allocate additional memory if the size of the captured variable is below some threshold.

This means that not only is there a performance penalty due to the extra dynamic memory allocation, but also that it is slower, as heap-allocated memory can increase the number of cache misses (read more about cache misses in *Chapter 4*, *Data Structures*).

Additional run-time computation

Calling a `std::function` is generally a bit slower than executing a lambda, as a little more code is involved. For small and frequently called `std::function`s, this overhead may become significant. Imagine that we have a really small lambda defined like this:

```
auto lambda = [](int v) { return v * 3; };
```

The benchmark that follows demonstrates the difference between executing 10 million function calls for a `std::vector` of the explicit lambda type versus a `std::vector` of a corresponding `std::function`. We will begin with the version using the explicit lambda:

```
auto use_lambda() {
  using T = decltype(lambda);
  auto fs = std::vector<T>(10'000'000, lambda);
  auto res = 1;
  // Start clock
  for (const auto& f: fs)
    res = f(res);
  // Stop clock here
  return res;
}
```

We only measure the time it takes to execute the loop inside the function. The next version wraps our lambda in a `std::function`, and looks like this:

```
auto use_std_function() {
  using T = std::function<int(int)>;
  auto fs = std::vector<T>(10'000'000, T{lambda});
  auto res = 1;
  // Start clock
  for (const auto& f: fs)
    res = f(res);
  // Stop clock here
  return res;
}
```

I'm compiling this code on my MacBook Pro from 2018 using Clang with optimizations turned on (`-O3`). The first version, `use_lambda()`, executes the loop at roughly 2 ms, whereas the second version, `use_std_function()`, takes almost 36 ms to execute the loop.

Generic lambdas

A generic lambda is a lambda accepting `auto` parameters, making it possible to invoke it with any type. It works just like a regular lambda, but the `operator()` has been defined as a member function template.

Only the parameters are template variables, not the captured values. In other words, the captured value, `v`, in the following example will be of type `int` regardless of the types of `v0` and `v1`:

```
auto v = 3; // int
auto lambda = [v](auto v0, auto v1) {
 return v + v0*v1;
};
```

If we translate the above lambda to a class, it would correspond to something like this:

```
class Lambda {
public:
 Lambda(int v) : v_{v} {}
 template <typename T0, typename T1>
 auto operator()(T0 v0, T1 v1) const {
  return v_ + v0*v1;
 }
private:
 int v_{};
};
auto v = 3;
auto lambda = Lambda{v};
```

Just like the templated version, the compiler won't generate the actual function until the lambda is invoked. So, if we invoke the previous lambda like this:

```
auto res_int = lambda(1, 2);
auto res_float = lambda(1.0f, 2.0f);
```

the compiler will generate something similar to the following lambdas:

```
auto lambda_int = [v](int v0, const int v1) { return v + v0*v1; };
auto lambda_float = [v](float v0, float v1) { return v + v0*v1; };
auto res_int = lambda_int(1, 2);
auto res_float = lambda_float(1.0f, 2.0f);
```

As you might have figured out, these versions are further handled just like regular lambdas.

A new feature of C++20 is that we can use `typename` instead of just `auto` for the parameter types of a generic lambda. The following generic lambdas are identical:

```
// Using auto
auto x = [](auto v) { return v + 1; };
// Using typename
auto y = []<typename Val>(Val v) { return v + 1; };
```

This makes it possible to name the type or refer to the type inside the body of the lambda.

Summary

In this chapter, you have learned how to use modern C++ features that will be used throughout this book. Automatic type deduction, move semantics, and lambda expressions are fundamental techniques that every C++ programmer needs to feel comfortable with today.

We also spent some time looking at error handling and how to think about bugs, along with valid states and how to recover from runtime errors. Error handling is an extremely important part of programming that is easily overlooked. Thinking about contracts between callers and callees is a way to make your code correct and avoid unnecessary defensive checks in the released version of your program.

In the next chapter, we will look into strategies for analyzing and measuring performance in C++.