```
struct SimpleStringOwner {
  SimpleStringOwner(const SimpleString& my_string) : string{ my_string }❶ { }
  --snip--
private:
  SimpleString string; ❷
};
```

Listing 4-32: A naive approach to member initialization containing a wasteful copy

There is hidden waste in this approach. You have a copy construction ❶, but the caller never uses the pointed-to object again after constructing `string` ❷. Figure 4-6 illustrates the issue.

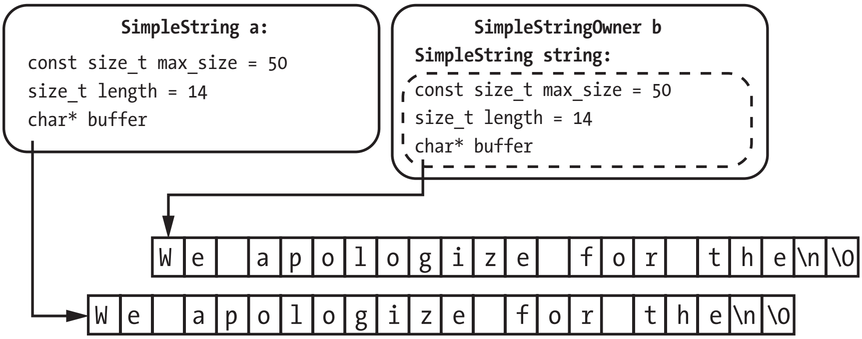


Figure 4-6: Using the copy constructor for `string` is wasteful.

You should move the guts of `SimpleString a` into the `string` field of `SimpleStringOwner`. Figure 4-7 shows what you want to achieve: `SimpleString Owner b` steals `buffer` and sets `SimpleString a` into a destructible state.

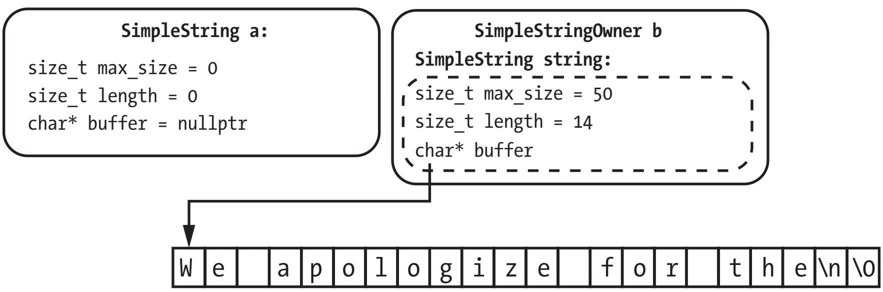


Figure 4-7: Swapping the buffer of `a` into `b`

After moving `a`, the `SimpleString` of `b` is equivalent to the former state of `a`, and `a` is destructible.

Moving can be dangerous. If you accidentally use moved-from `a`, you'd invite disaster. The class invariants of `SimpleString` aren't satisfied when `a` is moved from.

Fortunately, the compiler has built-in safeguards: lvalues and rvalues.

Value Categories

Every expression has two important characteristics: its *type* and its *value category*. A value category describes what kinds of operations are valid for the expression. Thanks to the evolutionary nature of C++, value categories are complicated: an expression can be a "generalized lvalue" (*glvalue*), a "pure rvalue" (*prvalue*), an "expiring value" (*xvalue*), an *lvalue* (a glvalue that isn't an xvalue), or an *rvalue* (a prvalue or an xvalue). Fortunately for the newcomer, you don't need to know much about most of these value categories.

We'll consider a very simplified view of value categories. For now, you'll just need a general understanding of lvalues and rvalues. An lvalue is any value that has a name, and an rvalue is anything that isn't an lvalue.

Consider the following initializations:

```
SimpleString a{ 50 };
SimpleStringOwner b{ a };                   // a is an lvalue
SimpleStringOwner c{ SimpleString{ 50 } };  // SimpleString{ 50 } is an rvalue
```

The etymology of these terms is *right value* and *left value*, referring to where each appears with respect to the equal sign in construction. In the statement `int x = 50;`, `x` is left of the equal sign (lvalue) and `50` is right of the equal sign (rvalue). These terms aren't totally accurate because you can have an lvalue on the right side of an equal sign (as in copy assignment, for example).

NOTE *The ISO C++ Standard details Value Categories in [basic] and [expr].*

lvalue and rvalue References

You can communicate to the compiler that a function accepts lvalues or rvalues using *lvalue references* and *rvalue references*. Up to this point in this book, every reference parameter has been an lvalue reference, and these are denoted with a single `&`. You can also take a parameter by rvalue reference using `&&`.

Fortunately, the compiler does an excellent job of determining whether an object is an lvalue or an rvalue. In fact, you can define multiple functions with the same name but with different parameters, and the compiler will automatically call the correct version depending on what arguments you provide when you invoke the function.

Listing 4-33 contains two functions with the name `ref_type` function to discern whether the invoker passed an lvalue or an rvalue reference.

```
#include <cstdio>

void ref_type(int &x) { ❶
  printf("lvalue reference %d\n", x);
}
```

```
void ref_type(int &&x) { ❷
  printf("rvalue reference %d\n", x);
}

int main() {
  auto x = 1;
  ref_type(x); ❸
  ref_type(2); ❹
  ref_type(x + 2); ❺
}
--------------------------------------------------------------------------
lvalue reference 1 ❸
rvalue reference 2 ❹
rvalue reference 3 ❺
```

Listing 4-33: A program containing an overloaded function with lvalue and rvalue references

The `int &x` version ❶ takes an lvalue reference, and the `int &&x` version ❷ takes an rvalue reference. You invoke `ref_type` three times. First, you invoke the lvalue reference version, because `x` is an lvalue (it has a name) ❸. Second, you invoke the rvalue reference version because `2` is an integer literal without a name ❹. Third, the result of adding 2 to `x` is not bound to a name, so it's an rvalue ❺.

NOTE *Defining multiple functions with the same name but different parameters is called* function overloading*, a topic you'll explore in detail in Chapter 9.*

The std::move Function

You can cast an lvalue reference to an rvalue reference using the `std::move` function from the `<utility>` header. Listing 4-34 updates Listing 4-33 to illustrate the use of the `std::move` function.

```
#include <utility>
--snip--
int main() {
  auto x = 1;
  ref_type(std::move(x)); ❶
  ref_type(2);
  ref_type(x + 2);
}
--------------------------------------------------------------------------
rvalue reference 1 ❶
rvalue reference 2
rvalue reference 3
```

Listing 4-34: An update to Listing 4-33 using `std::move` to cast `x` to an rvalue

As expected, `std::move` changes the lvalue `x` into an rvalue ❶. You never call the lvalue `ref_type` overload.

NOTE *The C++ committee probably should have named* `std::move` *as* `std::rvalue`*, but it's the name we're stuck with. The* `std:move` *function doesn't actually move anything—it casts.*

Be very careful when you're using `std::move`, because you remove the safeguards keeping you from interacting with a moved-from object. You can perform two actions on a moved-from object: destroy it or reassign it.

How lvalue and rvalue semantics enable move semantics should now be clear. If an lvalue is at hand, moving is suppressed. If an rvalue is at hand, moving is enabled.

Move Construction

Move constructors look like copy constructors except they take rvalue references instead of lvalue references.

Consider the `SimpleString` move constructor in Listing 4-35.

```
SimpleString(SimpleString&& other) noexcept
  : max_size{ other.max_size }, ❶
  buffer(other.buffer),
  length(other.length) {
  other.length = 0; ❷
  other.buffer = nullptr;
  other.max_size = 0;
}
```

Listing 4-35: A move constructor for `SimpleString`

Because `other` is an rvalue reference, you're allowed to cannibalize it. In the case of `SimpleString`, this is easy: just copy all fields of `other` into `this` ❶ and then zero out the fields of `other` ❷. The latter step is important: it puts `other` in a moved-from state. (Consider what would happen upon the destruction of `other` had you not cleared its members.)

Executing this move constructor is a *lot* less expensive than executing the copy constructor.

The move constructor is designed to *not* throw an exception, so you mark it `noexcept`. Your preference should be to use `noexcept` move constructors; often, the compiler cannot use exception-throwing move constructors and will use copy constructors instead. Compilers prefer slow, correct code instead of fast, incorrect code.

Move Assignment

You can also create a move analogue to copy assignment via `operator=`. The move assignment operator takes an rvalue reference rather than a `const` lvalue reference, and you usually mark it `noexcept`. Listing 4-36 implements such a move assignment operator for `SimpleString`.

```
SimpleString& operator=(SimpleString&& other) noexcept { ❶
  if (this == &other) return *this; ❷
  delete[] buffer; ❸
```

```
  buffer = other.buffer; ❹
  length = other.length;
  max_size = other.max_size;
  other.buffer = nullptr; ❺
  other.length = 0;
  other.max_size = 0;
  return *this;
}
```

Listing 4-36: A move assignment operator for `SimpleString`

You declare the move assignment operator using the rvalue reference syntax and the `noexcept` qualifier, as with the move constructor ❶. The self-reference check ❷ handles the move assignment of a `SimpleString` to itself. You clean up `buffer` ❸ before assigning the fields of `this` to the fields of `other` ❹ and zero out the fields of `other` ❺. Aside from the self-reference check ❷ and the cleanup ❸, the move assignment operator and the move constructor are functionally identical.

Now that `SimpleString` is movable, you can complete the `SimpleString` constructor of `SimpleStringOwner`:

```
SimpleStringOwner(SimpleString&& x) : string{ std::move(x)❶ } { }
```

The `x` is an lvalue, so you must `std::move x` into the move constructor of `string` ❶. You might find `std::move` odd, because `x` is an rvalue reference. Recall that lvalue/rvalue and lvalue reference/rvalue reference are distinct descriptors.

Consider if `std::move` weren't required here: what if you moved from `x` and then used it inside the constructor? This could lead to bugs that are hard to diagnose. Remember that you cannot use moved-from objects except to reassign or destruct them. Doing anything else is undefined behavior.

Listing 4-37 illustrates the `SimpleString` move assignment.

```
--snip--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the"); ❶
  SimpleString b{ 50 };
  b.append_line("Last message"); ❷
  a.print("a"); ❸
  b.print("b"); ❹
  b = std::move(a); ❺
  // a is "moved-from"
  b.print("b"); ❻
}
---------------------------------------------------------------------------
a: We apologize for the ❸
b: Last message ❹
b: We apologize for the ❻
```

Listing 4-37: A program illustrating move assignment with the `SimpleString` class

As in Listing 4-31, you begin by declaring two `SimpleString` classes with different messages: the string `a` contains `We apologize for the` ❶, and `b` contains `Last message` ❷. You print these strings to verify that they contain the strings you've specified ❸❹. Next, you move assign `b` equal to `a` ❺. Note that you had to cast `a` to an `rvalue` using `std::move`. After the move assignment, `a` is in a moved-from state, and you can't use it unless you reassign it to a new value. Now, `b` owns the message that `a` used to own, `We apologize for the` ❻.

The Final Product

You now have a fully implemented `SimpleString` that supports move and copy semantics. Listing 4-38 brings these all together for your reference.

```
#include <cstdio>
#include <cstring>
#include <stdexcept>
#include <utility>

struct SimpleString {
  SimpleString(size_t max_size)
    : max_size{ max_size },
    length{} {
    if (max_size == 0) {
      throw std::runtime_error{ "Max size must be at least 1." };
    }
    buffer = new char[max_size];
    buffer[0] = 0;
  }
  ~SimpleString() {
    delete[] buffer;
  }
  SimpleString(const SimpleString& other)
    : max_size{ other.max_size },
    buffer{ new char[other.max_size] },
    length{ other.length } {
    std::strncpy(buffer, other.buffer, max_size);
  }
  SimpleString(SimpleString&& other) noexcept
    : max_size(other.max_size),
    buffer(other.buffer),
    length(other.length) {
    other.length = 0;
    other.buffer = nullptr;
    other.max_size = 0;
  }
  SimpleString& operator=(const SimpleString& other) {
    if (this == &other) return *this;
    const auto new_buffer = new char[other.max_size];
    delete[] buffer;
    buffer = new_buffer;
    length = other.length;
    max_size = other.max_size;
    std::strncpy(buffer, other.buffer, max_size);
```

```
    return *this;
  }
  SimpleString& operator=(SimpleString&& other) noexcept {
    if (this == &other) return *this;
    delete[] buffer;
    buffer = other.buffer;
    length = other.length;
    max_size = other.max_size;
    other.buffer = nullptr;
    other.length = 0;
    other.max_size = 0;
    return *this;
  }
  void print(const char* tag) const {
    printf("%s: %s", tag, buffer);
  }
  bool append_line(const char* x) {
    const auto x_len = strlen(x);
    if (x_len + length + 2 > max_size) return false;
    std::strncpy(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n';
    buffer[length] = 0;
    return true;
  }
private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

Listing 4-38: A fully specified `SimpleString` class supporting copy and move semantics

Compiler-Generated Methods

Five methods govern move and copy behavior:

- The destructor
- The copy constructor
- The move constructor
- The copy assignment operator
- The move assignment operator

The compiler can generate default implementations for each under certain circumstances. Unfortunately, the rules for which methods get generated are complicated and potentially uneven across compilers.

You can eliminate this complexity by setting these methods to `default`/`delete` or by implementing them as appropriate. This general rule is the *rule of five*, because there are five methods to specify. Being explicit costs a little time, but it saves a lot of future headaches.

The alternative is memorizing Figure 4-8, which summarizes the interactions between each of the five functions you implement and each that the compiler generates on your behalf.

If you explicitly define:

You'll end up with:	Nothing	Destructor	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment
Destructor `~Foo()`	✓	✓	✓	✓	✓	✓
Copy Constructor `Foo(const Foo&)`	✓	✓	✓	✓		
Copy Assignment `Foo& operator=(const Foo&)`	✓	✓	✓	✓		
Move Constructor `Foo(Foo&&)`	✓		Copies are used in place of moves		✓	✓
Move Assignment `Foo& operator=(Foo&&)`	✓					

Figure 4-8: A chart illustrating which methods the compiler generates when given various inputs

If you provide nothing, the compiler will generate all five destruct/copy/move functions. This is the *rule of zero*.

If you explicitly define any of destructor/copy constructor/copy assignment operator, you get all three. This is dangerous, as demonstrated earlier with `SimpleString`: it's too easy to get into an unintended situation in which the compiler will essentially convert all your moves into copies.

Finally, if you provide only move semantics for your class, the compiler will not automatically generate anything except a destructor.

Summary

You've completed the exploration of the object life cycle. Your journey began in storage durations, where you saw an object lifetime from construction to destruction. Subsequent study of exception handling illustrated deft, lifetime-aware error handling and enriched your understanding of RAII. Finally, you saw how copy and move semantics grant you granular control over object lifetimes.

EXERCISES

4-1. Create a `struct TimerClass`. In its constructor, record the current time in a field called `timestamp` (compare with the POSIX function `gettimeofday`).

4-2. In the destructor of `TimerClass`, record the current time and subtract the time at construction. This time is roughly the *age* of the timer. Print this value.

4-3. Implement a copy constructor and a copy assignment operator for `TimerClass`. The copies should share `timestamp` values.

4-4. Implement a move constructor and a move assignment operator for `TimerClass`. A moved-from `TimerClass` shouldn't print any output to the console when it gets destructed.

4-5. Elaborate the `TimerClass` constructor to accept an additional `const char* name` parameter. When `TimerClass` is destructed and prints to stdout, include the name of the timer in the output.

4-6. Experiment with your `TimerClass`. Create a timer and move it into a function that performs some computationally intensive operation (for example, lots of math in a loop). Verify that your timer behaves as you expect.

4-7. Identify each method in the `SimpleString` class (Listing 4-38). Try reimplementing it from scratch without referring to the book.

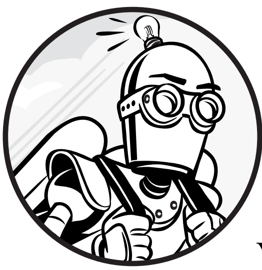
FURTHER READING

- *Optimized C++: Proven Techniques for Heightened Performance* by Kurt Guntheroth (O'Reilly Media, 2016)
- *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* by Scott Meyers (O'Reilly Media, 2015)

5

RUNTIME POLYMORPHISM

One day Trurl the constructor put together a machine that could create anything starting with n.
—*Stanislaw Lem,* The Cyberiad



In this chapter, you'll learn what polymorphism is and the problems it solves. You'll then learn how to achieve runtime polymorphism, which allows you to change the behavior of your programs by swapping out components during program execution. The chapter starts with a discussion of several crucial concepts in runtime polymorphic code, including interfaces, object composition, and inheritance. Next, you'll develop an ongoing example of logging bank transactions with multiple kinds of loggers. You'll finish the chapter by refactoring this initial, naive solution with a more elegant, interface-based solution.

Polymorphism

Polymorphic code is code you write once and can reuse with different types. Ultimately, this flexibility yields loosely coupled and highly reusable code. It eliminates tedious copying and pasting, making your code more maintainable and readable.

C++ offers two polymorphic approaches. *Compile-time polymorphic code* incorporates polymorphic types you can determine at compile time. The other approach is *runtime polymorphism*, which instead incorporates types determined at runtime. Which approach you choose depends on whether you know the types you want to use with your polymorphic code at compile time or at runtime. Because these closely related topics are fairly involved, they're separated into two chapters. Chapter 6 will focus on compile-time polymorphism.

A Motivating Example

Suppose you're in charge of implementing a `Bank` class that transfers money between accounts. Auditing is very important for the `Bank` class's transactions, so you provide support for logging with a `ConsoleLogger` class, as shown in Listing 5-1.

```
#include <cstdio>

struct ConsoleLogger {
  void log_transfer(long from, long to, double amount) { ❶
    printf("%ld -> %ld: %f\n", from, to, amount); ❷
  }
};

struct Bank {
  void make_transfer(long from, long to, double amount) { ❸
    --snip-- ❹
    logger.log_transfer(from, to, amount); ❺
  }
  ConsoleLogger logger;
};

int main() {
  Bank bank;
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
}
```

```
1000 -> 2000: 49.950000
2000 -> 4000: 20.000000
```

Listing 5-1: A `ConsoleLogger` and a `Bank` class that uses it

First, you implement `ConsoleLogger` with a `log_transfer` method ❶, which accepts the details of a transaction (sender, recipient, amount) and prints them ❷. The `Bank` class has the `make_transfer` method ❸, which (notionally) processes the transaction ❹ and then uses the `logger` member ❺ to log the transaction. The `Bank` and the `ConsoleLogger` have separate concerns—the `Bank` deals with bank logic, and the `ConsoleLogger` deals with logging.

Suppose you have a requirement to implement different kinds of loggers. For example, you might require a remote server logger, a local file logger, or even a logger that sends jobs to a printer. In addition, you must be able to change how the program logs at runtime (for example, an administrator might need to switch from logging over the network to logging to the local filesystem because of some server maintenance).

How can you accomplish such a task?

A simple approach is to use an `enum class` to switch between the various loggers. Listing 5-2 adds a `FileLogger` to Listing 5-1.

```
#include <cstdio>
#include <stdexcept>

struct FileLogger {
  void log_transfer(long from, long to, double amount) { ❶
    --snip--
    printf("[file] %ld,%ld,%f\n", from, to, amount);
  }
};

struct ConsoleLogger {
  void log_transfer(long from, long to, double amount) {
    printf("[cons] %ld -> %ld: %f\n", from, to, amount);
  }
};

enum class LoggerType { ❷
  Console,
  File
};

struct Bank {
  Bank() : type { LoggerType::Console } { } ❸

  void set_logger(LoggerType new_type) { ❹
    type = new_type;
  }

  void make_transfer(long from, long to, double amount) {
    --snip--
    switch(type) { ❺
    case LoggerType::Console: {
      consoleLogger.log_transfer(from, to, amount);
      break;
    } case LoggerType::File: {
      fileLogger.log_transfer(from, to, amount);
      break;
    } default: {
```

```
      throw std::logic_error("Unknown Logger type encountered.");
    } }
  }
private:
  LoggerType type;
  ConsoleLogger consoleLogger;
  FileLogger fileLogger;
};

int main() {
  Bank bank;
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
  bank.set_logger(LoggerType::File); ❻
  bank.make_transfer(3000, 2000, 75.00);
}
```

```
[cons] 1000 -> 2000: 49.950000
[cons] 2000 -> 4000: 20.000000
[file] 3000,2000,75.000000
```

Listing 5-2: An updated Listing 5-1 with a runtime polymorphic logger

You (notionally) add the ability to log to a file ❶ by implementing a `FileLogger`. You also create an `enum class LoggerType` ❷ so you can switch logging behavior at runtime. You initialize the type field to `Console` within the `Bank` constructor ❸. Within the updated `Bank` class, you add a `set_logger` function ❹ to perform the desired logging behavior. You use the `type` within `make_transfer` to `switch` on the correct logger ❺. To alter a `Bank` class's logging behavior, you use the `set_logger` method ❻, and the object handles dispatching internally.

Adding New Loggers

Listing 5-2 works, but this approach suffers from several design problems. Adding a new kind of logging requires you to make several updates throughout the code:

1. You need to write a new logger type.
2. You need to add a new `enum` value to the `enum class LoggerType`.
3. You must add a new case in the `switch` statement ❺.
4. You must add the new logging class as a member to `Bank`.

That's a lot of work for a simple change!

Consider an alternative approach where `Bank` holds a pointer to a logger. This way, you can set the pointer directly and get rid of `LoggerType` entirely. You exploit the fact that your loggers have the same function prototype. This is the idea behind an interface: the `Bank` class doesn't need to know the implementation details of the `Logger` reference it holds, just how to invoke its methods.

Wouldn't it be nice if we could swap out the `ConsoleLogger` for another type that supports the same operations? Say, a `FileLogger`?

Allow me to introduce you to the interface.

Interfaces

In software engineering, an *interface* is a shared boundary that contains no data or code. It defines function signatures that all implementations of the interface agree to support. An *implementation* is code or data that declares support for an interface. You can think of an interface as a contract between classes that implement the interface and users (also called *consumers*) of that class.

Consumers know how to use implementations because they know the contract. In fact, the consumer never needs to know the underlying implementation type. For example, in Listing 5-1 `Bank` is a consumer of `ConsoleLogger`.

Interfaces impose stringent requirements. A consumer of an interface can use only the methods explicitly defined in the interface. The `Bank` class doesn't need to know anything about how `ConsoleLogger` performs its function. All it needs to know is how to call the `log_transfer` method.

Interfaces promote highly reusable and loosely coupled code. You can understand the notation for specifying an interface, but you'll need to know a bit about object composition and implementation inheritance.

Object Composition and Implementation Inheritance

Object composition is a design pattern where a class contains members of other class types. An alternate, antiquated design pattern called *implementation inheritance* achieves runtime polymorphism. Implementation inheritance allows you to build hierarchies of classes; each child inherits functionality from its parents. Over the years, accumulated experience with implementation inheritance has convinced many that it's an antipattern. For example, Go and Rust—two new and increasingly popular system-programming languages—have zero support for implementation inheritance. A brief discussion of implementation inheritance is warranted for two reasons:

- You might encounter it infecting legacy code.
- The quirky way you define C++ interfaces has a shared lineage with implementation inheritance, so you'll be familiar with the mechanics anyway.

NOTE *If you're dealing with implementation inheritance–laden C++ code, see Chapters 20 and 21 of* The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup.*

Defining Interfaces

Unfortunately, there's no `interface` keyword in C++. You have to define interfaces using antiquated inheritance mechanisms. This is just one of those archaisms you have to deal with when programming in a 40+ year-old language.

Listing 5-3 illustrates a fully specified `Logger` interface and a corresponding `ConsoleLogger` that implements the interface. At least four constructions in Listing 5-3 will be unfamiliar to you, and this section covers each of them.

```
#include <cstdio>

struct Logger {
  virtual❶ ~Logger()❷ = default;
  virtual void log_transfer(long from, long to, double amount) = 0❸;
};

struct ConsoleLogger : Logger ❹ {
  void log_transfer(long from, long to, double amount) override ❺ {
    printf("%ld -> %ld: %f\n", from, to, amount);
  }
};
```

Listing 5-3: A `Logger` interface and a refactored `ConsoleLogger`

To parse Listing 5-3, you'll need to understand the `virtual` keyword ❶, the virtual destructor ❷, the `=0` suffix and pure-virtual methods ❸, base class inheritance ❹, and the `override` keyword ❺. Once you understand these, you'll know how to define an interface. The sections that follow discuss these concepts in detail.

Base Class Inheritance

Chapter 4 delved into how the `exception` class is the base class for all other stdlib exceptions and how the `logic_error` and `runtime_error` classes derived from `exception`. These two classes, in turn, form the base classes for other derived classes that describe error conditions in even greater detail, such as `invalid_argument` and `system_error`. Nested exception classes form an example of a class hierarchy and represent an implementation inheritance design.

You declare derived classes using the following syntax:

```
struct DerivedClass : BaseClass {
  --snip--
};
```

To define an inheritance relationship for `DerivedClass`, you use a colon (`:`) followed by the name of the base class, `BaseClass`.

Derived classes are declared just like any other class. The benefit is that you can treat derived class references as if they were of base class reference type. Listing 5-4 uses a `DerivedClass` reference in place of a `BaseClass` reference.

```
struct BaseClass {}; ❶
struct DerivedClass : BaseClass {}; ❷
void are_belong_to_us(BaseClass& base) {} ❸

int main() {
  DerivedClass derived;
  are_belong_to_us(derived); ❹
}
```

Listing 5-4: A program using a derived class in place of a base class

The `DerivedClass` ❷ derives from `BaseClass` ❶. The `are_belong_to_us` function takes a reference-to-`BaseClass` argument `base` ❸. You can invoke it with an instance of a `DerivedClass` because `DerivedClass` derives from `BaseClass` ❹.

The opposite is not true. Listing 5-5 attempts to use a base class in place of a derived class.

```
struct BaseClass {}; ❶
struct DerivedClass : BaseClass {}; ❷
void all_about_that(DerivedClass& derived) {} ❸

int main() {
  BaseClass base;
  all_about_that(base); // No! Trouble! ❹
}
```

Listing 5-5: This program attempts to use a base class in place of a derived class. (This listing won't compile.)

Here, `BaseClass` ❶ doesn't derive from `DerivedClass` ❷. (The inheritance relationship is the other way around.) The `all_about_that` function takes a `DerivedClass` argument ❸. When you attempt to invoke `all_about_that` with a `BaseClass` ❹, the compiler yields an error.

The main reason you'd want to derive from a class is to inherit its members.

Member Inheritance

Derived classes inherit non-private members from their base classes. Classes can use inherited members just like normal members. The supposed benefit of member inheritance is that you can define functionality once in a base class and not have to repeat it in the derived classes. Unfortunately, experience has convinced many in the programming community to avoid member inheritance because it can easily yield brittle, hard-to-reason-about code compared to composition-based polymorphism. (This is why so many modern programming languages exclude it.)

The class in Listing 5-6 illustrates member inheritance.

```
#include <cstdio>

struct BaseClass {
  int the_answer() const { return 42; } ❶
```

```
  const char* member = "gold"; ❷
private:
  const char* holistic_detective = "Dirk Gently"; ❸
};

struct DerivedClass : BaseClass ❹ {};

int main() {
  DerivedClass x;
  printf("The answer is %d\n", x.the_answer()); ❺
  printf("%s member\n", x.member); ❻
  // This line doesn't compile:
  // printf("%s's Holistic Detective Agency\n", x.holistic_detective); ❼
}
```

```
The answer is 42 ❺
gold member ❻
```

Listing 5-6: A program using inherited members

Here, `BaseClass` has a public method ❶, a public field ❷, and a private field ❸. You declare a `DerivedClass` deriving from `BaseClass` ❹ and then use it in `main`. Because they're inherited as public members, `the_answer` ❺ and `member` ❻ are available on the `DerivedClass x`. However, uncommenting ❼ yields a compiler error because `holistic_detective` is private and thus not inherited by derived classes.

virtual Methods

If you want to permit a derived class to override a base class's methods, you use the `virtual` keyword. By adding `virtual` to a method's definition, you declare that a derived class's implementation should be used if one is supplied. Within the implementation, you add the `override` keyword to the method's declaration, as demonstrated in Listing 5-7.

```
#include <cstdio>

struct BaseClass {
  virtual❶ const char* final_message() const {
    return "We apologize for the incontinence.";
  }
};

struct DerivedClass : BaseClass ❷ {
  const char* final_message() const override ❸ {
    return "We apologize for the inconvenience.";
  }
};

int main() {
  BaseClass base;
  DerivedClass derived;
  BaseClass& ref = derived;
```

```
  printf("BaseClass:    %s\n", base.final_message()); ❹
  printf("DerivedClass: %s\n", derived.final_message()); ❺
  printf("BaseClass&:   %s\n", ref.final_message()); ❻
}
--------------------------------------------------------------------------
BaseClass:    We apologize for the incontinence. ❹
DerivedClass: We apologize for the inconvenience. ❺
BaseClass&:   We apologize for the inconvenience. ❻
```

Listing 5-7: A program using virtual members

The `BaseClass` contains a virtual member ❶. In the `DerivedClass` ❷, you override the inherited member and use the `override` keyword ❸. The implementation of `BaseClass` is used only when a `BaseClass` instance is at hand ❹. The implementation of `DerivedClass` is used when a `DerivedClass` instance is at hand ❺, even if you're interacting with it through a `BaseClass` reference ❻.

If you want to *require* a derived class to implement the method, you can append the `=0` suffix to a method definition. You call methods with both the `virtual` keyword and `=0` suffix pure virtual methods. You can't instantiate a class containing any pure virtual methods. In Listing 5-8, consider the refactor of Listing 5-7 that uses a pure virtual method in the base class.

```
#include <cstdio>

struct BaseClass {
  virtual const char* final_message() const = 0; ❶
};

struct DerivedClass : BaseClass ❷ {
  const char* final_message() const override ❸ {
    return "We apologize for the inconvenience.";
  }
};

int main() {
  // BaseClass base; // Bang! ❹
  DerivedClass derived;
  BaseClass& ref = derived;
  printf("DerivedClass: %s\n", derived.final_message()); ❺
  printf("BaseClass&:   %s\n", ref.final_message()); ❻
}
--------------------------------------------------------------------------
DerivedClass: We apologize for the inconvenience. ❺
BaseClass&:   We apologize for the inconvenience. ❻
```

Listing 5-8: A refactor of Listing 5-7 using a pure virtual method

The `=0` suffix specifies a pure virtual method ❶, meaning you can't instantiate a `BaseClass`—only derive from it. `DerivedClass` still derives from `BaseClass` ❷, and you provide the requisite `final_message` ❸. Attempting to instantiate a `BaseClass` would result in a compiler error ❹. Both `DerivedClass` and the `BaseClass` reference behave as before ❺❻.

NOTE *Virtual functions can incur runtime overhead, although the cost is typically low (within 25 percent of a regular function call). The compiler generates* virtual function tables (vtables) *that contain function pointers. At runtime, a consumer of an interface doesn't generally know its underlying type, but it knows how to invoke the interface's methods (thanks to the vtable). In some circumstances, the linker can detect all uses of an interface and* devirtualize *a function call. This removes the function call from the vtable and thus eliminates associated runtime cost.*

Pure-Virtual Classes and Virtual Destructors

You achieve interface inheritance through deriving from base classes that contain only pure-virtual methods. Such classes are referred to as *pure-virtual classes*. In C++, interfaces are always pure-virtual classes. Usually, you add virtual destructors to interfaces. In some rare circumstances, it's possible to leak resources if you fail to mark destructors as virtual. Consider Listing 5-9, which illustrates the danger of not adding a virtual destructor.

```
#include <cstdio>

struct BaseClass {};

struct DerivedClass : BaseClass❶ {
  DerivedClass() { ❷
    printf("DerivedClass() invoked.\n");
  }
  ~DerivedClass() { ❸
    printf("~DerivedClass() invoked.\n");
  }
};

int main() {
  printf("Constructing DerivedClass x.\n");
  BaseClass* x{ new DerivedClass{} }; ❹
  printf("Deleting x as a BaseClass*.\n");
  delete x; ❺
}
--------------------------------------------------------------------------------
Constructing DerivedClass x.
DerivedClass() invoked.
Deleting x as a BaseClass*.
```

Listing 5-9: An example illustrating the dangers of non-virtual destructors in base classes

Here you see a `DerivedClass` deriving from `BaseClass` ❶. This class has a constructor ❷ and destructor ❸ that print when they're invoked. Within `main`, you allocate and initialize a `DerivedClass` with `new` and set the result to a `BaseClass` pointer ❹. When you `delete` the pointer ❺, the `BaseClass` destructor gets invoked, but the `DerivedClass` destructor doesn't!

Adding virtual to the `BaseClass` destructor solves the problem, as demonstrated in Listing 5-10.

```
#include <cstdio>

struct BaseClass {
  virtual ~BaseClass() = default; ❶
};

struct DerivedClass : BaseClass {
  DerivedClass() {
    printf("DerivedClass() invoked.\n");
  }
  ~DerivedClass() {
    printf("~DerivedClass() invoked.\n"); ❷
  }
};

int main() {
  printf("Constructing DerivedClass x.\n");
  BaseClass* x{ new DerivedClass{} };
  printf("Deleting x as a BaseClass*.\n");
  delete x; ❸
}
--------------------------------------------------------------------------
Constructing DerivedClass x.
DerivedClass() invoked.
Deleting x as a BaseClass*.
~DerivedClass() invoked. ❷
```

Listing 5-10: A refactor of Listing 5-9 with a virtual destructor

Adding the virtual destructor ❶ causes the `DerivedClass` destructor to get invoked when you delete the `BaseClass` pointer ❸, which results in the `DerivedClass` destructor printing the message ❷.

Declaring a virtual destructor is optional when declaring an interface, but beware. If you forget that you haven't implemented a virtual destructor in the interface and accidentally do something like Listing 5-9, you can leak resources, and the compiler won't warn you.

NOTE *Declaring a protected non-virtual destructor is a good alternative to declaring a public virtual destructor because it will cause a compilation error when writing code that deletes a base class pointer. Some don't like this approach because you eventually have to make a class with a public destructor, and if you derive from that class, you run into the same issues.*

Implementing Interfaces

To declare an interface, declare a pure virtual class. To implement an interface, derive from it. Because the interface is pure virtual, an implementation must implement all of the interface's methods.

It's good practice to mark these methods with the `override` keyword. This communicates that you intend to override a virtual function, allowing the compiler to save you from simple mistakes.

Using Interfaces

As a consumer, you can only deal in references or pointers to interfaces. The compiler cannot know ahead of time how much memory to allocate for the underlying type: if the compiler could know the underlying type, you would be better off using templates.

There are two options for how to set the member:

Constructor injection With constructor injection, you typically use an interface reference. Because references cannot be reseated, they won't change for the lifetime of the object.

Property injection With property injection, you use a method to set a pointer member. This allows you to change the object to which the member points.

You can combine these approaches by accepting an interface pointer in a constructor while also providing a method to set the pointer to something else.

Typically, you'll use constructor injection when the injected field won't change throughout the lifetime of the object. If you need the flexibility of modifying the field, you'll provide methods to perform property injection.

Updating the Bank Logger

The `Logger` interface allows you to provide multiple logger implementations. This allows a `Logger` consumer to log transfers with the `log_transfer` method without having to know the logger's implementation details. You've already implemented a `ConsoleLogger` in Listing 5-2, so let's consider how you can add another implementation called `FileLogger`. For simplicity, in this code you'll only modify the log output's prefix, but you can imagine how you might implement some more complicated behavior.

Listing 5-11 defines a `FileLogger`.

```
#include <cstdio>

struct Logger {
  virtual ~Logger() = default; ❶
  virtual void log_transfer(long from, long to, double amount) = 0; ❷
};

struct ConsoleLogger : Logger ❸ {
  void log_transfer(long from, long to, double amount) override ❹ {
    printf("[cons] %ld -> %ld: %f\n", from, to, amount);
  }
};
```

```
struct FileLogger : Logger ❺ {
  void log_transfer(long from, long to, double amount) override ❻ {
    printf("[file] %ld,%ld,%f\n", from, to, amount);
  }
};
```

Listing 5-11: `Logger`, `ConsoleLogger`, and `FileLogger`

`Logger` is a pure virtual class (interface) with a default virtual destructor ❶ and a single method `log_transfer` ❷. `ConsoleLogger` and `FileLogger` are `Logger` implementations, because they derive from the interface ❸❺. You've implemented `log_transfer` and placed the `override` keyword on both ❹❻.

Now we'll look at how you could use either constructor injection or property injection to update `Bank`.

Constructor Injection

Using constructor injection, you have a `Logger` reference that you pass into the `Bank` class's constructor. Listing 5-12 adds to Listing 5-11 by incorporating the appropriate `Bank` constructor. This way, you establish the kind of logging that a particular `Bank` instantiation will perform.

```
--snip--
// Include Listing 5-11
struct Bank {
  Bank(Logger& logger) : logger{ logger }❶ { }
  void make_transfer(long from, long to, double amount) {
    --snip--
    logger.log_transfer(from, to, amount);
  }
private:
  Logger& logger;
};

int main() {
  ConsoleLogger logger;
  Bank bank{ logger }; ❷
  bank.make_transfer(1000, 2000, 49.95);
  bank.make_transfer(2000, 4000, 20.00);
}
--------------------------------------------------------------------------------
[cons] 1000 -> 2000: 49.950000
[cons] 2000 -> 4000: 20.000000
```

Listing 5-12: Refactoring Listing 5-2 using constructor injection, interfaces, and object composition to replace the clunky `enum class` approach

The `Bank` class's constructor sets the value of `logger` using a member initializer ❶. References can't be reseated, so the object that `logger` points to doesn't change for the lifetime of `Bank`. You fix your logger choice upon `Bank` construction ❷.

Property Injection

Instead of using constructor injection to insert a `Logger` into a `Bank`, you could use property injection. This approach uses a pointer instead of a reference. Because pointers can be reseated (unlike references), you can change the behavior of `Bank` whenever you like. Listing 5-13 is a property-injected variant of Listing 5-12.

```
--snip--
// Include Listing 5-10

struct Bank {
  void set_logger(Logger* new_logger) {
    logger = new_logger;
  }
  void make_transfer(long from, long to, double amount) {
    if (logger) logger->log_transfer(from, to, amount);
  }
private:
  Logger* logger{};
};

int main() {
  ConsoleLogger console_logger;
  FileLogger file_logger;
  Bank bank;
  bank.set_logger(&console_logger); ❶
  bank.make_transfer(1000, 2000, 49.95); ❷
  bank.set_logger(&file_logger); ❸
  bank.make_transfer(2000, 4000, 20.00); ❹
}
```

```
[cons] 1000 -> 2000: 49.950000 ❷
[file] 2000,4000,20.000000 ❹
```

Listing 5-13: Refactoring Listing 5-12 using property injection

The `set_logger` method enables you to inject a new logger into a `Bank` object at any point during the life cycle. When you set the logger to a `ConsoleLogger` instance ❶, you get a `[cons]` prefix on the logging output ❷. When you set the logger to a `FileLogger` instance ❸, you get a `[file]` prefix ❹.

Choosing Constructor or Property Injection

Whether you choose constructor or property injection depends on design requirements. If you need to be able to modify underlying types of an object's members throughout the object's life cycle, you should choose pointers and the property injector method. But the flexibility of using pointers and property injection comes at a cost. In the `Bank` example in this chapter, you must make sure that you either don't set `logger` to `nullptr` or that you check for this condition before using `logger`. There's also the question of what the default behavior is: what is the initial value of `logger`?

One possibility is to provide constructor and property injection. This encourages anyone who uses your class to think about initializing it. Listing 5-14 illustrates one way to implement this strategy.

```
#include <cstdio>

struct Logger {
  --snip--
};

struct Bank {
  Bank(Logger* logger) : logger{ logger } () ❶
  void set_logger(Logger* new_logger) { ❷
    logger = new_logger;
  }
  void make_transfer(long from, long to, double amount) {
    if (logger) logger->log_transfer(from, to, amount);
  }
private:
    Logger* logger;
};
```

Listing 5-14: A refactor of the `Bank` *to include constructor and property injection*

As you can see, you can include a constructor ❶ and a setter ❷. This requires the user of a `Bank` to initialize logger with a value, even if it's just `nullptr`. Later on, the user can easily swap out this value using property injection.

Summary

In this chapter, you learned how to define interfaces, the central role that virtual functions play in making inheritance work, and some general rules for using constructor and property injectors. Whichever approach you choose, the combination of interface inheritance and composition provides sufficient flexibility for most runtime polymorphic applications. You can achieve type-safe runtime polymorphism with little or no overhead. Interfaces encourage encapsulation and loosely coupled design. With simple, focused interfaces, you can encourage code reuse by making your code portable across projects.

EXERCISES

5-1. You didn't implement an accounting system in your `Bank`. Design an interface called `AccountDatabase` that can retrieve and set amounts in bank accounts (identified by a `long` id).

5-2. Generate an `InMemoryAccountDatabase` that implements `AccountDatabase`.

5-3. Add an `AccountDatabase` reference member to `Bank`. Use constructor injection to add an `InMemoryAccountDatabase` to the `Bank`.

5-4. Modify `ConsoleLogger` to accept a `const char*` at construction. When `ConsoleLogger` logs, prepend this string to the logging output. Notice that you can modify logging behavior without having to modify `Bank`.

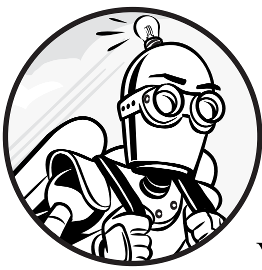
FURTHER READING

- *API Design for C++* by Martin Reddy (Elsevier, 2011)

6

COMPILE-TIME POLYMORPHISM

The more you adapt, the more interesting you are.
—Martha Stewart



In this chapter, you'll learn how to achieve compile-time polymorphism with templates. You'll learn how to declare and use templates, enforce type safety, and survey some of the templates' more advanced usages. This chapter concludes with a comparison of runtime and compile-time polymorphism in C++.

Templates

C++ achieves compile-time polymorphism through *templates*. A template is a class or function with template parameters. These parameters can stand in for any type, including fundamental and user-defined types. When the compiler sees a template used with a type, it stamps out a bespoke template instantiation.

Template instantiation is the process of creating a class or a function from a template. Somewhat confusingly, you can also refer to "a template instantiation" as the result of the template instantiation process. Template instantiations are sometimes called concrete classes and concrete types.

The big idea is that, rather than copying and pasting common code all over the place, you write a single template; the compiler generates new template instances when it encounters a new combination of types in the template parameters.

Declaring Templates

You declare templates with a *template prefix*, which consists of the keyword `template` followed by angle brackets `< >`. Within the angle brackets, you place the declarations of one or more template parameters. You can declare template parameters using either the `typename` or `class` keywords followed by an identifier. For example, the template prefix `template<typename T>` declares that the template takes a template parameter `T`.

NOTE *The coexistence of the* `typename` *and* `class` *keywords is unfortunate and confusing. They mean the same thing. (They're both supported for historical reasons.) This chapter always uses* `typename`.

Template Class Definitions

Consider `MyTemplateClass` in Listing 6-1, which takes three template parameters: `X`, `Y`, and `Z`.

```
template❶<typename X, typename Y, typename Z> ❷
struct MyTemplateClass❸ {
  X foo(Y&); ❹
private:
  Z* member; ❺
};
```

Listing 6-1: A template class with three template parameters

The `template` keyword ❶ begins the template prefix, which contains the template parameters ❷. This `template` preamble leads to something special about the remaining declaration of `MyTemplateClass` ❸. Within `MyTemplateClass`, you use `X`, `Y`, and `Z` as if they were any fully specified type, like an `int` or a user-defined class.

The `foo` method takes a `Y` reference and returns an `X` ❹. You can declare members with types that include template parameters, like a pointer to `Z` ❺. Besides the special prefix beginning ❶, this template class is essentially identical to a non-template class.

Template Function Definitions

You can also specify template functions, like the my_template_function in Listing 6-2 that also takes three template parameters: X, Y, and Z.

```
template<typename X, typename Y, typename Z>
X my_template_function(Y& arg1, const Z* arg2) {
  --snip--
}
```

Listing 6-2: A template function with three template parameters

Within the body of my_template_function, you can use arg1 and arg2 however you'd like, as long as you return an object of type X.

Instantiating Templates

To instantiate a template class, use the following syntax:

```
tc_name❶<t_param1❷, t_param2, ...> my_concrete_class{ ... }❸;
```

The *tc_name* ❶ is where you place the template class's name. Next, you fill in your template parameters ❷. Finally, you treat this combination of template name and parameters as if it were a normal type: you use whatever initialization syntax you like ❸.

Instantiating a template function is similar:

```
auto result = tf_name❶<t_param1❷, t_param2, ...>(f_param1❸, f_param2, ...);
```

The *tf_name* ❶ is where you put the template function's name. You fill in the parameters just as you do for template classes ❷. You use the combination of template name and parameters as if it were a normal type. You invoke this template function instantiation with parentheses and function parameters ❸.

All this new notation might be daunting to a newcomer, but it's not so bad once you get used to it. In fact, it's used in a set of language features called named conversion functions.

Named Conversion Functions

Named conversions are language features that explicitly convert one type into another type. You use named conversions sparingly in situations where you cannot use implicit conversions or constructors to get the types you need.

All named conversions accept a single object parameter, which is the object you want to cast *object-to-cast*, and a single type parameter, which is the type you want to cast to *desired-type*:

```
named-conversion<desired-type>(object-to-cast)
```