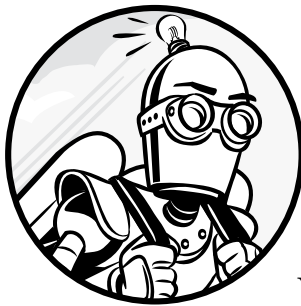# 5

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 log-
ging 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 func-
tion 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 dispatch-
ing 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 anti-pattern. 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.

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.

FURTHER READING

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