Chapter 9: Abstraction and Classes

Software keeps getting bigger. Society keeps digitizing and automating more and more aspects of life, and the scope and complexity of software systems are ever increasing. For computer scientists, this is a thrilling prospect: even after decades of booming growth, the field is still expanding and applications abound. But for software engineers - the brave souls who actually write the code – this can be daunting.

In the early days of programming, software was considerably less complicated because the tasks we used to ask of computers are nowhere near as complex as those we ask today. Operating systems worked on less powerful hardware and with considerably fewer peripherals. The earliest web browsers didn't need to support a wide array of HTML, CSS, JavaScript, XML, SVG, and RSS formats. Video games didn't need to take advantage of the latest-and-greatest 3D hardware and weren't criticized for not having the most up-to-date shading engine. But nowadays, the expectations are higher, and software is growing more complicated.

Unfortunately, increasing the size of a software system greatly increases the system's complexity and opens all sorts of avenues for failure. Combating software complexity is therefore extraordinarily important as it allows software systems to grow robustly. This section of this book is dedicated entirely to techniques for combating complexity through a particular technique called *abstraction*. Abstraction is a subtle but important aspect of software design, and in many ways the difference between good programmers and excellent programmers is the ability to design robust and intuitive abstractions in software.

Many textbooks jump directly into a discussion of what abstraction is all about and how to represent it in software, but I feel that doing so obscures the fundamental reasons underlying abstraction. This chapter discusses how software engineering is different from other engineering disciplines, why software complexity is particularly dangerous, and how abstraction can dramatically reduce the complexity of a software system. It then introduces the `class` keyword and how to represent abstraction directly in source code.

The Complexity of Software

What exactly does it mean to talk about the complexity of a software system? One of the first metrics that may come to mind is the number of lines of code in the program. This is akin to measuring the complexity of a chip by the number of transistors on it or a bridge by the number of welds required: while in general system complexity rises with lines of code, a program with ten thousand lines of code is not necessarily ten times more complicated than a system with one thousand lines of code. However, number of lines of code is still a reasonable metric of software complexity. To give you a sense for how massive some projects can be, here is a list of various software projects and the number of lines of code they contain:

1 – 10	Hello, World!
10 – 100	Most implementation of the STL `stack` or `queue`.
100 – 1,000	Most of the worked examples in this book.
1,000 – 10,000	Intensive team project in a typical computer science curriculum.
10,000 – 100,000	Most Linux command-line utilities.
100,000 – 1,000,000	Linux `g++` Compiler
1,000,000 – 10,000,000	Mozilla Firefox
10,000,000 – 100,000,000	Microsoft Windows 2000 Kernel
100,000,000 - 1,000,000,000	Debian Linux Operating System

The number of lines of code in each of these entries is ten times more than in the previous example. This means that there are ten times as many lines of code in Firefox than in `g++`, for example. And yes, you did read this correctly – there are many, *many* projects that clock in at over a million lines of code. The Debian Linux kernel is roughly 230 million lines of code as of this writing. It's generally accepted that no single programmer can truly understand more than fifty thousand lines of code, which means that in all but the simplest of programs, no one programmer understands how the entire system works.

So software is complex – so what? That is, why does it matter that modern software systems are getting bigger and more complicated? There are many reasons, of which two specifically stand out.

Every Bit Counts

In a software system, a single incorrect bit can spell disaster for the entire program. For example, suppose you are designing a system that controls a nuclear reactor core. At some point in your program, you have the following control logic:

```
if (MeltdownInProgress())
{
   SetReactorPower(0);
   EmergencyShutoff();
}
```

Let's suppose that `MeltdownInProgress` returns a `bool` that signals whether the reactor is melting down. On most systems, `bool`s are represented as a sequence of ones and zeros called *bits*. The value `false` is represented by those bits all being zero, and `true` represented by any of the bits being nonzero. For example, the value `00010000` would be interpreted as `true`, while `00000000` would be `false`. This means that the difference between `true` and `false` is a single bit. In the above example, this means that the difference between shutting down the reactor in an emergency and continuing normal operation is a *single bit in memory*.

In the above example, our "single incorrect bit" was the difference between the boolean values `true` and `false`, but in most systems the "single incorrect bit" will be something else. It might, for example, be an negative integer where a positive integer was expected, an iterator that is past the end of a container, or an unsorted `vector` when the data was expected to be sorted. In each of these cases, the erroneous data is likely to be only a handful of bits off from a meaningful piece of data, but the result will be the same – the program won't work as expected.

Interactions Grow Exponentially

Suppose you have a program with n lines of code. Let's consider an "interaction" to be where data is manipulated by two different lines of code. For example, one interaction might be creating an `int` in one line and then printing it to the console in the next, or a function passing one of its parameters into another function. Since every line of code might potentially manipulate data created by any of the other n lines of code, if you sum up this count for all n lines of code there are roughly n^2 pairs of interacting lines. This which means that the number of possible interactions in a software project increases, in the worst case, as the *square* of the number of lines of code.

Let's consider a slightly different take on this. Suppose you are working on a software project with n lines of code and you're interested in adding a new feature. As mentioned above, any changes you make might interact with any of the existing n lines of code, and those n lines of code might interact with all of your new lines of code. If the changes you make somehow violate an assumption that exists in some other module, then changes in your relatively isolated region of the code base might cause catastrophic failures in entirely different parts of the code base.

However, the situation is far worse than this. In this example we considered an interaction to be an interaction between two lines of code. A more realistic model of interactions would consider interactions between *arbitrar-*

ily many lines of code, since changes made in several different points might converge together in a point to form a result not possible if a single one of the changes didn't occur. In this case, if the code base has n lines of code, the maximum number of interactions (sets of two or more lines of code) is roughly 2^n. That's a staggeringly huge number. In fact, if we make the liberal assumption that there are 10^{100} atoms in the universe (most estimates put the figure at much less than this), then even a comparably small software system (say, three thousand lines of code) has more possible interactions than there are atoms in the universe.

In short, the larger a software system gets, the greater the likelihood than an error occurs and, consequently, the more difficult it is to make changes. In short, software is *chaotic*, and even minuscule changes to a code base can completely cripple the system.

Abstraction

One of the most powerful techniques available to combat complexity is *abstraction*, a means of simplifying a complex program to a manageable level. Rather than jumping headfirst into a full-on definition of abstraction with examples, let's look at abstraction by means of an example. Consider a standard, run-of-the-mill stapler. Certainly you understand how to use a stapler: you place the papers to staple under the arm of the stapler, then depress the handle to staple the pages together. You've undoubtedly encountered more than one stapler in your life, yet (barring unfortunate circumstances) you've probably figured out how to work all of them without much trouble. Staplers come in all shapes and sizes, and consequently have many different internal mechanisms, yet switching from one type of stapler to another poses little to no problem to you. In fact, you probably don't think much about staplers, even when you're using them.

Now consider the companies that make staplers – Swingline or McGill, for example. These companies expend millions of dollars designing progressively better staplers. They consider all sorts of tradeoffs between different types of springs and different construction materials. In fact, they probably expend more effort in a single day designing staplers than you will ever spend thinking about staplers in your entire life. But nonetheless, at the end of the day, staplers are simple and easy to use and bear no markings to indicate the painstaking labor that has gone into perfecting them. This setup, where a dedicated manufacturer designs a complex but easy-to-use product, is the heart of abstraction.

Formally speaking, an *abstraction* is a description of an object that omits all but a few salient details. For example, suppose we want to describe a particular coffee mug. Here are several different descriptions of the coffee mug, each at different levels of abstraction:

- Matter.
- An object.
- A beverage container.
- A coffee mug.
- A white coffee mug.
- A white ceramic coffee mug.
- A white ceramic coffee mug with a small crack in the handle.
- A white ceramic coffee mug with a small crack in the handle whose manufacturer's logo is emblazoned on the bottom.

Notice how these descriptions move from least specific (matter) to most specific (A white ceramic coffee mug with a small crack in the handle whose manufacturer's logo is emblazoned on the bottom). Each of the descriptions describe the same coffee mug, but each does so at a different level of detail. Depending on the circumstance, different levels of detail might be appropriate. For example, if you were a physicist interested in modeling universal gravitation, the fact that the coffee mug is made of matter might be sufficient for your purposes. However, if you wanted to paint a picture of the mug, you would probably want to pick the last description, since it offers the most detail. If you'll notice, as the descriptions become more and more detailed, more and more information is revealed about the object. Starting from the first of these descriptions and moving downward, the

picture of the coffee mug becomes more clear. Describing the coffee mug as "matter" hardly helps you picture the mug, but as you go down the list you begin to notice that the mug is white, has a small crack, and has a logo printed on the bottom.

What does this have to do with our previous discussion on staplers? The answer is simple: the reason that staplers are so easy to use despite the complex mechanisms that make them work is because the very notion of a stapler is an abstraction. There are many ways to build a stapler, some of which have handles to staple documents, and others which use proximity sensors to detect paper and insert the staples automatically. Although these devices have little mechanism or structure in common, we would consider both of them staplers because they staple paper. In other words, what is important to us as stapler users is the fact that staplers fasten paper together, not their inner workings. This may seem like a strange line of reasoning, but it's one of the single most important concepts to grasp as a computer scientist. The rest of this chapter is dedicated to exploring what abstraction means from a programming perspective and how abstraction can combat complexity. But first, let's discuss some of the major concepts and terms pertaining to abstraction at a high level.

The Wall of Abstraction

In our previous example with staplers, there was a clear separation of complexity between the stapler manufacturer and the end user. The manufacturer took painstaking care to ensure that the stapler works correctly, and end users just press a handle or feed paper near a sensor. This separation is fundamental to combating complexity, and is given an appropriately impressive name: *the wall of abstraction*.

The wall of abstraction is the information barrier between a *device* and *how it works*. On one side of the wall is the *manufacturer*, whose task is to provide a device capable of meeting certain requirements. To the manufacturer, the single most important task is producing a device that works correctly and reliably. On the other side of the wall of abstraction is the *end user*, who is interested in using the device but who, unless curious, does not particularly care how the device works. In computer science, we refer to these two roles as the *client*, who uses the device, and the *implementer*, who is tasked with making it work correctly.

When using a stapler, you don't care how the stapler works because it's an unnecessary mental burden. You shouldn't need to know what type of metal the casing is made from, nor should you have to worry about what type of spring pushes the staples up to the front of the stapler. The same is true of almost every device and appliance in use today. Do you know exactly how a microwave works? How about an internal combustion engine? What about an iPhone? Each of these devices is extraordinarily complicated and works on nuanced principles of physics, materials science, chemical engineering, and in some cases electrical and software engineering. The magic of all of these devices is that *we don't need to know how they work*. We can trust that a team of dedicated engineers understand their inner workings, and can focus instead on using them.

The fact that the wall of abstraction separates the implementer in the client necessarily means that an abstraction shields clients from unnecessary implementation details. In that sense, the wall of abstraction is a barrier that prevents information about the device's internals from leaking outside. That the wall of abstraction is an information barrier has profound consequences for software design. Before we discuss how abstraction can reduce system complexity, let us focus on this aspect in more detail.

Abstractions are Imprecise

Earlier in this chapter we discussed abstraction in the context of a coffee mug by exploring descriptions of a coffee mug at various levels of abstraction. Suppose that we have an actual coffee mug we are interested in designing an abstraction for; for example, this mug here:

The highest-level description of a coffee mug in the original list was the extraordinarily vague "matter." That is, all of the properties of the coffee mug are ignored except for the fact that it is matter. This means the implementer (us) holds a coffee mug, but the client (the person reading the description "matter") knows only that our object is composed of matter. Because the wall of abstraction prevents information from leaking to the client, that our object is made of matter is the only information the client has about the coffee mug. This means that the client can't tell if our object is a coffee mug, Jupiter's moon Ganymede, or a fried egg. In other words, our description was so vague that the client knows nothing about what object we have.

Let's now move to a lower level of abstraction, the description that the mug is "a beverage container." The client can now tell that our mug is not Ganymede, nor is it a fried egg, but we haven't yet excluded other possibilities. Many things are beverage containers – punch bowls, wine glasses, thermoses, etc. – and the client cannot figure out from our limited description that the object is a coffee mug. However, without knowing that the object is a coffee mug, at this level of abstraction the client could safely assume that our object could store water.

Now let's consider an even more precise description: we describe the coffee mug as "a coffee mug." Now what can the client infer about our mug? Because we have said that the object is a mug, they know that it's a coffee mug, true, but there are many properties of the object that they don't know. For example, how big is the mug? What color is it? What design, if any, adorns the mug? A client on the other side of the wall of abstraction still can't paint a particularly vivid picture of the mug we're describing, but they now know a good deal more about the mug than they did with either of the two previous descriptions.

This above discussion hits on a major point: abstractions are imprecise. When describing the coffee mug at various levels of detail, we always truthfully represented our coffee mug, but our descriptions never were sufficient to unambiguously convince the client of what object we were describing. In fact, if we had instead been describing a different coffee mug, like this one here:

then all of our descriptions would still have been perfectly honest.

Abstractions, by their very nature, make it impossible for the client to know exactly what object is being described. This is an incredible blessing. Suppose, for example, that you bring your car in for routine maintenance. The mechanic informs you that your radiator is nearing the end of its lifespan, so you pay for a replacement. Once the mechanic replaces the radiator and you drive off into the sunset, the car that you are driving is not the same car that you drove in with. One of its fundamental components has been replaced, and so an integral part of the car's system is not the same as it used to be. However, you feel like you are driving the same car when you leave because from your perspective, nothing has changed. The accelerator and brakes still work as they used to, the car handles like it used to, and in fact almost every maneuver you perform with the car will execute exactly the same way that the car used to. Viewing this idea through the lens of abstraction, the reason for this is that your conception of the car has to do with its observable behavior. The car you are now driving has a different "implementation" than the original car, but it adheres to the same abstraction as the old car and is consequently indistinguishable from the original car. Without looking under the hood (breaking the wall of abstraction), you wouldn't be able to notice the difference.

Interfaces

The wall of abstraction sits at the boundary between two worlds – the world of the implementer, where the workings of the mechanism are of paramount importance, and the world of the client, where the observable behavior is all that matters. As mentioned earlier, the wall of abstraction is an information barrier that prevents implementation details and usage information from crossing between the client and implementer. But if the client is unaware of the implementation of the particular device, how can she possibly use it? That is, if there is a logical barrier between the two parties, how can the client actually use the implementer's device?

One typical way to provide the client access to the implementation is via an *interface*. An interface is a set of commands and queries that can be executed on the device, and is the way that the client interacts with the object. For example, an interface might let the client learn some properties of the object in question, or might allow the client to ask the device to perform some task. In software engineering, an interface typically consists of a set of attributes (also called properties) that the object is required to have, along with a set of actions that the object can perform. For example, here's one possible interface for a stapler:

- Attributes:
  - Number of staples left.
  - Size of staples being used.
  - How many sheets of paper are in the stapler.
  - The maximum number of sheets of paper that the stapler can staple.
- Actions:
  - Add more staples.
  - Put paper into the stapler.
  - Staple the papers together.

Interfaces are fascinating because they provide a particularly elegant means for a implementer to expose an object to a client. The implementer is free to build the device in question as she sees fit, provided that all of the operations specified in the interface work correctly. That is, someone implementing a stapler that adheres to the above interface can use whatever sort of mechanism they feel like to build the stapler, so long as it is possible to look up how many staples are left, to add more staples to the stapler, etc. Similarly, the client needs only learn the operations in the interface and should (theoretically) be able to use any device that conforms to that interface. In software engineering terminology, we say that the implementer *implements* the interface by providing a means of transforming any request given to the interface to a request to the underlying device. For example, if the Swingline corporation decided to create a new stapler, they might build a concrete stapler and then implement the interface for staplers as follows:

- Attributes:
  - Number of staples left: **Open the cover and count the number of staples.**
  - Size of staples being used: **Open the cover and look at the size of the staples.**
  - How many sheets of paper are in the stapler: **Count the sheets of paper on the base plate.**
  - The maximum number of sheets of paper that the stapler can staple: **25**
- Actions:
  - Add more staples: **Open the cover and insert more staples.**
  - Put paper into the stapler: **Place the paper over the base plate**.
  - Staple the papers together: **Depress the handle until it clicks, then release the handle.**

However, we could also implement the stapler inteface in a different way if we were using an electronic stapler:

- Attributes:
  - Number of staples left: **Read the digital display**.
  - Size of staples being used: **Read the digital display**.
  - How many sheets of paper are in the stapler: **Read the digital display**.
  - The maximum number of sheets of paper that the stapler can staple: **75**
- Actions:
  - Add more staples: **Open the cover, remove the old roll of staples, and snap the new roll in place**.
  - Put paper into the stapler: **Place the paper into the loading assembly**.
  - Staple the papers together: **Press the "staple" button**.

Notice that these two staplers have the same interface but entirely different actions associated with each item in the interface. This is a concrete example of abstraction in action - because the interface only describes some specific attributes and actions associated with the stapler, anything that can make these actions work correctly can be treated as a stapler. The actual means by which the interface is implemented may be different, but the general principle is the same.

An extremely important point to note is the relation between interfaces and abstractions. Abstraction is a general term that describes how to simplify systems by separating the role of the client and the implementer. Interfaces are the means by which abstractions are actually modeled in software systems. Whenever one speaks of abstraction in programming, it usually refers to designing an interface. In other words, an object's interface is a concrete description of the abstraction provided for that object.

Encapsulation

When working with interfaces and abstractions, we build a wall of abstraction to prevent implementation details about an object from leaking to the client. This means that the client does not necessarily need to know how the particular object is implemented, and can just rely on the fact that some implementer has implemented the interface correctly. But while an interface captures the idea that a client doesn't *have* to know the particular implementation details, it does not express the idea that a client *shouldn't* know the particular implementation details. To understand this, let's return to our discussion of staplers. If an implementer provides a particular stapler that implements the stapler interface, then anyone using that stapler can just use the interface to the stapler to get all of their required functionality. However, there's nothing stopping them from disassembling the stapler, looking at its component parts, etc. In fact, given a physical stapler, it's possible to do things with that stapler that weren't initially anticipated. You could, for example, replace the stapler handle with a pneumatic compressor to build a stapler gun, which might make the stapler more efficient in a particular application. However, you could also remove the spring inside the stapler which forces the staples to the front of the staple tray, rendering the stapler useless.

In general, allowing clients to bypass interfaces and directly modify the object described by that interface is dangerous. The entire purpose of an interface is to let implementers build arbitrarily complicated systems that can

be operated simply, and if a client bypasses the interface he'll be dealing with an object whose workings could easily be far beyond his comprehension. In the case of a stapler, bypassing the interface and looking at the stapler internals isn't likely to cause any problems, but you would certainly be asking for trouble if you were to start poking around the internals of the Space Shuttle. This violation, where a client bypasses an interface, is called *breaking the wall of abstraction*.

The above examples have hinted at why breaking the wall of abstraction is a bad idea, but we haven't explicitly spelled out any reasons why in general it can be dangerous. Let us do this now. First, breaking the wall of abstraction allows clients to severely hurt themselves by tweaking a complex system. In a complex system like a car engine, certain assumptions have to hold about the relationship between the parts of the car in order for the car to work correctly. That is, fuel shouldn't be injected into the engine except in certain parts of the cycle, the transmission shouldn't try shifting gears until the clutch has been released, etc. Consequently, most cars provide an interface into the engine that consists of a gas and brake pedal, whose operation controls all of the relevant parts of the engine. If you were to ignore these controls and instead try to drive the car by manually adjusting fuel intake and the brake pressure, barring special training, you would almost certainly either cause an explosion or irreversibly destroy the engine. Remember that abstraction protects both the client and the implementer – the client doesn't need to know about the inner workings of the object, and the implementer doesn't need to worry that the client can make arbitrary changes to the object; all operations on the object must come through the interface. Breaking the wall of abstraction violates both these assumptions and can hurt both parties.

The second major reason against breaking the wall of abstraction is to ensure system flexibility. As mentioned earlier, abstractions are by nature imprecise, and multiple different implementations might satisfy a particular interface. If the client is allowed to break the wall of abstraction and look at a particular part of the implementation, then that implementation is in essence "locked in place." For example, suppose that you provide a traditional stapler to a client. That client then decides to use the stapler in a context where the exact position and orientation of the stapler hinge is important; perhaps the client has trained a robot to use the stapler by learning to feed paper into the stapler whenever the hinge is at a particular angle. Earlier in this chapter, we discussed how interfaces make it possible to change an object's implementation without befuddling the user. That is, if the client only uses the operations listed in an interface, then any object implementing that interface should be substitutable for any other. The problem with breaking the wall of abstraction is that this is no longer possible. Consider, for example, what happens if we try to replace the mechanical stapler from this setup with an electric stapler. Electric staplers tend not to have hinges, and so if we swapped staplers the robot designed to feed paper into the stapler would no longer be able to insert paper. In other words, because the robot assumed that some property held true of the implementation that was documented nowhere in the interface, it became impossible to ever change the implementation.

To summarize – peering behind an interface and looking at the underlying implementation is a bad idea. It allows clients to poke systems in ways that were never intended, and it locks the particular implementation in place.

If an abstraction does not allow clients to look at the implementation under any circumstance, that abstraction is said to be *encapsulated*. In other words, it is as though the actual implementation is trapped inside a giant capsule, and the only way to access the object is by issuing queries and commands specified by the interface. Encapsulation is uncommon in the real world, but some analogies exist. For example, if you visit a rare book collection, you cannot just go in and take any book off the shelf. Instead, you have to talk to a librarian who will then get the book for you. You have no idea where the book comes from – perhaps it's sitting on a shelf in the back, or perhaps the librarian has to get a courier to fetch it from some special vault – but this doesn't concern you because at the end of the day you (hopefully) have the book anyway.

Encapsulated interfaces are extraordinarily important in software because they represent a means for entirely containing complexity. The immense amount of implementation detail that might be necessary to implement an interface is abstracted away into a small set of commands that can be executed on that interface, and encapsulation prevents other parts of the program from inadvertently (or deliberately) modifying the implementation in

unexpected ways. Later in this chapter, when we discuss classes, you will see how C++ allows you to build encapsulated interfaces.

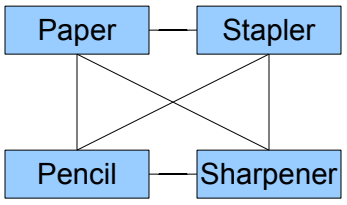
The Math: Why Abstraction Works

We've talked about abstraction and how it lets clients operate with complex objects without knowing their full implementation. The implicit claim throughout this chapter has been that this greatly reduces the complexity of software systems. Amazingly, given a suitable definition of system complexity, we can *prove* that increasing the level of abstraction in a system reduces the maximum complexity possible in the system.

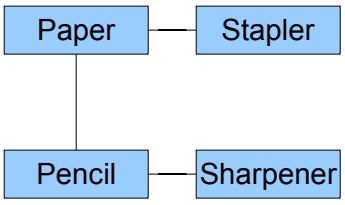
In this discussion, we'll need to settle on a definition of a system's complexity. If a system consists of different interfaces, we will define the maximum complexity of that system to be the maximum number of interactions between these interfaces. For our purposes, we'll consider an interaction between interfaces to be a set of two or more interfaces. It can be shown that there are $2^n - n - 1$ possible interactions between interfaces, which is an absolutely huge number. In fact, in a system with ten interfaces, there are 1,013 possible interactions between those interfaces. The reason for this is the first term in this quantity (2^n), which grows extremely fast. It grows so quickly that we can ignore the last two terms in the sum and approximate the maximum complexity of a system as 2^n.

Now, suppose that we introduce a new abstraction into a system that reduces the total number of interfaces in the system by 10. This means that the new system has n - 10 interfaces, and consequently its maximum complexity is 2^{n-10}. If we take the ratio of the maximum complexity of the new system to the maximum complexity of the old system, we get 2^{-10}, which is just under one one-thousandth. That is, the maximum complexity of the new system will be roughly one one-thousandth that of the original system. This result has extremely important implications for software design. It is possible to build reasonably simple software systems that are hundreds of millions of lines of code simply by minimizing the number of interfaces present in the software system. This caps the maximum complexity of the system by limiting the number of possible interactions.

But the above logic is terribly misleading. In practice, software systems rarely get even close to reaching the maximum number of possible interactions. Maximum complexity only occurs if every combination of objects has a well-defined interaction, and this is rarely the case. For example, in a simulation with a stapler, pen, and pencil sharpener, you are unlikely to ever have the stapler and pencil sharpener interact, and if you do it is extremely unlikely that you will have all three objects have some specific behavior when interacting all at the same time. A more realistic measure of complexity is the number of ways in which *pairs* of objects can interact. This is a desirable choice for several reasons. First, it corresponds to an elegant graphical measure of complexity. If we list all of the components in a system and add lines between pairs of objects that interact with each other, the complexity of that system is then the number of lines in the picture. For example, here are two diagrams of ways that common office supplies might interact with one another. The first system is clearly more complex than the second since there are more interactions defined between the components.



Setup one: Everything interacts with everything else.



Setup two: Only meaningful interactions defined.

Second, in practice, interactions between two or more objects can usually be simplified down into multiple instances of interactions between pairs of objects. For example, if three billiard balls all collide, we could consider the interaction between the three balls as three separate interactions of the pairs of balls. Only in unusual circumstances is such a decomposition not possible. Finally, considering interactions only of pairs rather than of triples or quadruples tends to correspond more accurately to how systems are actually built. It is conceptually simpler to think about how a single piece of a system interacts with each of its neighbors in isolation than it is to think about how that pieces interacts with all of its neighbors simultaneously.

Even with this more restrictive definition of complexity, reducing the number of interfaces in a system still produces larger reductions in complexity. It can be shown that the number of possible pairs of interacting objects is slightly less than n^2. This means that if we make a linear reduction in the number of objects in the system, we get a quadratic decrease in the maximum complexity in that system. That is, removing ten interfaces isn't going to drop the maximum complexity by ten interactions – it will be a considerably bigger number.

Classes

The single most important difference between C++ and its predecessor C is the notion of a *class*. Classes are C++'s mechanism for encoding and representing abstraction, pairing interfaces with implementations, and enforcing encapsulation. The entire remainder of this book will be dedicated to exploring how to create, modify, maintain, use, and refine classes and class definitions.

In the previous discussion on abstraction, we discussed abstractly the notions of interfaces and encapsulation. Before we discuss the class mechanism, let's consider an extended example that illustrates exactly why abstraction and interfaces are so important. In particular, we will explore how one might represent an FM radio tuner in C++ code. We won't actually create a working FM radio in software – that would require specialized hardware – but the example should demonstrate many of the reasons why classes are so important.

Designing an FM Radio

In our example, we will create a data structure that stores information about an FM radio. Since the properties of an FM radio can't be represented with a single variable, we'll create a struct called `FMRadio` which will hold all of our data. What should this struct contain? At a bare minimum, we will need to know what frequency the radio is tuned in to. We also probably want to specify a volume control, so that listeners can turn up high-energy music or turn down shouting news pundits. We can represent this information as follows:

```
struct FMRadio
{
    double frequency;
    int    volume;
};
```

Here, the frequency field stores the frequency in MHz. For example, if you were listening to 88.5 KQED San Francisco, this field would have the value 88.5. Similarly, listening to 107.9 The End Sacramento would have the field hold 107.9. I've arbitrarily chosen to store the volume as an int that holds a value between 0 and 10, inclusive. Volume zero completely mutes the radio, while volume ten is as much power as the speaker can deliver. This means that if I wanted to configure my radio to listen to "This American Life" at a reasonably quiet level, I could write

```
FMRadio myRadio;
myRadio.frequency = 88.5; // 88.5 MHz (KQED)
myRadio.volume    = 3;    // Reasonably quiet
```

Now, let's consider one more extension to the radio. Most radios these days let the user configure up to six different "presets," saved stations that listeners can adjust the radio to quickly. Most car radios have this feature, al-

though older FM radios do not. The presets are numbered one through six, and at any time a particular preset might be empty (the listener hasn't programmed this preset yet) or set to a particular frequency. As an example, I frequently commute between Palo Alto and Sacramento, and enjoy listening to NPR on the drive. Both Sacramento and San Francisco have stations that broadcast NPR content, and about halfway between Palo Alto and Sacramento one of the stations fades out dramatically while the other one comes in much more strongly. To make it easier to switch between the stations, I programmed my car radio's presets so that preset one is the San Francisco station (88.5) and preset two is the Sacramento station (89.3).

We'd like to add this functionality to FMRadio, but what's the best way to do so? If we want to store a list of six different settings, we could do so with a `vector`, but run into a problem because a `vector` always enforces the restriction that there must be an element at every position. Because some of the presets might not be configured, we might run into trouble if we stored the elements in a `vector` because it would be difficult to determine whether a particular position in the `vector` was empty or filled with a valid station. Instead, we'll implement the `FMRadio` using a `map` that maps from the preset number to the station at the preset. If a particular value between 1 and 6 is not a key in the `map`, then the preset has not been configured; if it is a key, then its value is the preset. This leads to the following version of `FMRadio`:

```
struct FMRadio
{
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

If I then wanted to program my radio as described above, I would do so as follows:

```
FMRadio myRadio;
myRadio.presets[1] = 88.5;
myRadio.presets[2] = 89.3;
```

Abusing the FM Radio

The definition of `FMRadio` from above seems reasonably straightforward. It has three fields that correspond to some attribute of the radio. Unfortunately, however, using this `FMRadio` in any complex software system can cause problems. The reason is that there are certain restrictions on what values the fields of the `FMRadio` can and cannot be, but there is no means of enforcing those restrictions. For example, in the United States, all FM radio frequencies are between 87.5 and 108.0 MHz. Consequently, the `frequency` field should never be set to any value out of this range, since doing so would be meaningless. Similarly, we've stated that we don't want the volume field to leave the range 0 to 10, but nothing prevents clients of `FMRadio` from doing so. Finally, the presets field has to obey two restrictions: that the keys are integers between 1 and 6, and that the values are `doubles` restricted to the range of valid frequencies.

Now, suppose that someone who does not have this intimate knowledge of the FM radio class we've designed comes along and writes the following code:

```
FMRadio myRadio;
myRadio.frequency  = 110.0; // Problem: Invalid frequency
myRadio.volume     = 11;    // Problem: Volume out of range
myRadio.presets[0] = 85.0;  // Problem: Bad preset index, invalid frequency
```

All of the above operations are illegal on FM radios, but this code compiles and runs just fine. Moreover, there is no indication at runtime that this code isn't going to work correctly. Everything that the client has done is perfectly legal C++, and the compiler has no idea that something bad might happen in the future. To give a context of where things can go wrong, suppose that we have a function that adjusts the power level to some system peri-

pheral to tune in to the proper frequency. Because all legal frequencies are between 87.5 and 108.0 MHz, the code adjusts the power level to a floating-point value such that the power is 0.0 at the lowest possible frequency (87.5 MHz) and 1.0 at the highest frequency (108.0 MHz). This code is shown below, assuming the existence of a SetDevicePower function that actually sets the device power:

```
void TuneReceiver(FMRadio radio)
{
    /* Compute the fraction of the maximum power that this
     * frequency requires.
     */
    double powerLevel = (radio.frequency - 87.5) / (108.0 - 87.5);
    SetDevicePower(powerLevel);
}
```

I'll leave double-checking that the above computation gives the fraction of the power to the transmitter as an exercise to the reader. In the meantime, think about what will happen if we write the following code:

```
FMRadio myRadio;
myRadio.frequency  = 110.0;
myRadio.volume     = 11;
myRadio.presets[0] = 85.0;
TuneReceiver(myRadio);
```

We now have a fairly serious problem on our hands. Because the radio frequency is 110.0 MHz, a value out of the valid FM radio range, the code inside of `TuneReceiver` is going to set the power level to a nonsensical value. In particular, since (110.0 – 87.5) / (108.0 – 87.5) ≈ 1.095, the code will turn the receiver on at roughly 110% of the maximum power it's supposed to receive. If we're lucky, the code inside `TuneReceiver` will have a check that this value is out of range, and the program will report an error. If we're unlucky and the code actually drives too much power into the receiver, we might overload the device and set it on fire. In other words, because the client of the `FMRadio struct` set a single field to a nonsensical value, it's possible that our program will crash or cause a physical device malfunction. This is clearly unacceptable, and we will need to do something about this.

Modifying the FM Radio

Of course, that's not all of the problems we might encounter when working with the `FMRadio`. Suppose, for example, that we write the following function, which sets the radio's frequency to the preset at the given position if possible, and does not change the frequency otherwise. The code is as follows:

```
void LoadPreset(FMRadio& radio, int preset)
{
    /* Check whether this preset exists. */
    map<int, double>::iterator itr = radio.presets.find(preset);

    /* If not, don't do anything. */
    if (itr == radio.presets.end())
        return;

    /* Otherwise, change the radio frequency. */
    radio.frequency = itr->second;
}
```

Now, suppose that for some reason (efficiency, perhaps) that we decide to change the FMRadio struct so that the presets are implemented as an array of doubles. We arbitrarily say that any preset that has not been programmed will be represented by having the value 0 stored in a particular slot. That is, given my NPR travel presets, the preset array would look like this:

Value	88.5	89.3	0	0	0	0
Index	0	1	2	3	4	5

This requires us to change the definition of the `FMRadio` interface to use a raw array instead of an STL `map`. The updated definition is shown here:

```
struct FMRadio
{
    double frequency;
    int    volume;
    double presets[6];
};
```

We now have a serious problem. Almost of the code that we've written previously that uses the `FMRadio`'s preset field will fail to compile. For example, our earlier code for `LoadPreset` will call `presets.find`, which does not exist in a raw array. This means that this single change might require us to rewrite huge amounts of code. In a small project, this is a mere annoyance, but in a large system on the order of millions of lines of code might be so time-consuming as to render the change impossible.

What Went Wrong?

The above discussion highlighted two problems with the `FMRadio struct`. First, `FMRadio` provides no means for enforcing its invariants. Because the aspects of the FM radio were represented in `FMRadio` by raw variables, any part of the program can modify those variables without the `FMRadio` getting a chance to intervene. In other words, the `FMRadio` expects that certain relations hold between its fields, but has no mechanism for enforcing those relations. Second, because the `FMRadio` is represented in software as a particular implementation of an FM radio, code that uses the FM radio necessarily locks the FM radio into that particular implementation. Using the terminology from the earlier in this chapter, this implementation of the FM radio provides no *abstraction* and no *encapsulation*. The `FMRadio` interface *is* its implementation – that is, the operations that clients can perform on the `FMRadio` are manipulations of the fields that compose the `FMRadio`. Changing the implementation thus changes the interface, which is why changing the fields breaks existing code. Similarly, because the interface of `FMRadio` is the set of all possible manipulations of the data members, clients can tweak the `FMRadio` in any way they see fit, even if such manipulations break internal invariants.

This is the reality of what C++ programming is like without classes. Code bases are more brittle, bugs are more likely, and changes are more difficult. As we now change gears and see how to represent an FM radio using classes, keep this starting point in mind. By the time you finish this chapter, the FM radio will be significantly more robust than it is now.

Introduction to Classes

In C++, a class is an *interface* paired with an *implementation*. Like `structs`, classes define new types that can be created and used elsewhere in the program.

Because classes pair an implementation and an interface, the structure of an individual class can be partitioned into two halves – the *public interface* specifying how clients interact with the class, and the *private implementation*, which specifies how functions in the public interface are implemented. Rather than diving head-first into a full-blown class definition, we'll investigate each of these parts individually. We will focus first on how to declare the class, and worry about the implementation later.

Defining a Public Interface

Let's return to the example of the FM radio. We are interested in designing an abstraction that represents an FM radio, then expressing the radio in software. In particular, we want our radio to have three pieces of data: the current frequency (in MHz), the volume (from 0 to 10), and the presets. As we saw in the failed experiment with `struct FMRadio`, we cannot simply give clients direct access to the fields that ultimately implement these properties. How, then, can we design an FM radio that contains some data but which does not allow clients to directly modify the data? The answer is a subtle yet beautiful trick that is ubiquitous in modern software. We will create a set of functions that set and query the value of these data members. We then prevent the client from directly accessing the data members that these functions manipulate. The major advantage of this approach is that every operation that could potentially read or modify the data must go through this small set of functions. Consequently, the implementer retains full control over what operations manipulate the class's implementation.

Now, let's see how one might express this in C++. We will rewrite our `FMRadio struct` from earlier to convert it into a fully-fledged C++ class. To begin, we use the C++ `class` keyword to indicate that we're defining a new class called `FMRadio`. This is shown here:

```
class FMRadio
{
};
```

Currently, this class is empty and is useless. We'll thus start defining the public interface by which clients of `FMRadio` will interact with the class. In C++, to define a class's public interface, we use the `public` keyword to indicate the start of the interface, and then list the functions contained in that public interface. This leads us to the following code:

```
class FMRadio
{
public:

};
```

That is, the public keyword, followed by a colon. Any definitions that follow the public keyword will be included as part of the class's public interface. But what functions should we put in here? Let's begin by letting the client query and set the radio's frequency. To do this, we'll define two member functions called `getFrequency` and `setFrequency`. This is shown here:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);
};
```

These functions are called member functions of the `FMRadio` class. Although they look like regular function prototypes, because these functions are defined inside of `FMRadio`, they are local to that class. In fact, calling the function `getFrequency` by itself will result in a compile-time error because there is no global function called `getFrequency`. Instead, we've defined a function that can be invoked on an object of type `FMRadio`. To see how this works, let's create a new object of type `FMRadio`. This is syntactically identical to the code for creating instances of a `struct` type – we put the name of the type, followed by the variable name. This is shown here:

```
FMRadio myRadio; // Declare a new variable of type FMRadio
```

Now that we have this `myRadio` object, we can ask it for its frequency by invoking the `getFrequency` member function. This is shown here:

```
FMRadio myRadio;
double f = myRadio.getFrequency(); // Query the radio for its frequency
```

Note that this code will not run as written, because we have not yet implemented the `getFrequency()` function; we'll see how to do that later in this chapter. However, this syntax should seem familiar, as it's the same syntax we used to invoke functions on STL containers, stream objects, and strings. In fact, all of those objects are instances of classes. You're on the road to learning how these complex objects are put together!

Let's continue designing our interface. We also want a means for the client to set and read the radio volume. Along the same lines as before, we can add a pair of member functions to `FMRadio` to grant access to this data. This is shown here:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);
};
```

Clients can then read and write the volume by writing code like this:

```
FMRadio myRadio;

myRadio.setVolume(8);
cout << myRadio.getVolume() << endl;
```

Again, this code will not run because we haven't implemented either of these functions. Don't worry, we're almost at the point where we'll be able to do this.

Let us now consider the final piece of the `FMRadio` interface – the code for manipulating presets. With the previous two properties (volume and frequency) we were working with a single entity, but we now must design an interface to let clients read and write multiple different values. Moreover, some of these values might not exist, since the presets might not yet be programmed in. To design a good interface, we should consider what clients would like to do with presets. We should certainly allow clients to set each of the presets. Additionally, clients should be able to check whether a certain preset has been programmed in. Finally, clients should be able to read back the presets they've programmed in, assuming they exist. We can represent each of these operations with a member function, leading to this interface for the `FMRadio` class:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);
};
```

We now have an interface for our `FMRadio` class. Now, let's see how we specify the implementation of this interface.

Writing a Class Implementation

A C++ class represents an abstraction, which consists of an interface into some object. We've just seen how to define the interface for the class, and now we must provide an implementation of that interface. This implementation consists of two parts. First, we must define what variables we will use to implement the class. This is akin to choosing the fields we put in a `struct` for the information to be useful. Second, we must provide an implementation of each of the member functions we defined in the class's public interface. We will do each of these in a separate step.

If you'll recall, one old version of `FMRadio` was a struct that looked like this:

```
struct FMRadio
{
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

This is a perfectly fine implementation of an `FMRadio` since it allows us to store all of the information we could possibly need. We'll therefore modify our implementation of the `FMRadio` class so that it is implemented using these three fields. However, we want to do this in a way that prevents clients of `FMRadio` from accessing the fields directly. For this purpose, C++ provides the `private` keyword, which indicates that certain parts of a class are completely off-limits to clients. This is shown here:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

When referring to elements of a `struct`, one typically uses the term *field*. In the context of classes, these variables are called *data members*. That is, `frequency` is a *data member* of `FMRadio`, and `getVolume` is a *member function*.

Because we've marked these data members private, the C++ compiler will enforce that no client of the `FMRadio` class can access them. For example, consider the following code:

```
FMRadio myRadio;
myRadio.frequency = 110.0; // Problem: Illegal; frequency is private
```

This code will cause a compile-time error because the frequency data member is private. To write code to this effect, clients would have to use the public interface, in particular the `setFrequency` member function, as shown here:

```
FMRadio myRadio;
myRadio.setFrequency(110.0); // Legal: setFrequency is public
```

All that's left to do now is implement the member functions on the `FMRadio` class. Implementing a member function is syntactically similar to implementing a regular function, though there are a few differences. One obvious syntactic difference is the means by which we specify the name of the function. If we are interested in implementing the `getFrequency` function of `FMRadio`, for example, then we would begin as follows:

```
double FMRadio::getFrequency()
{
    /* ... implementation goes here ... */
}
```

Notice that the name of the function is `FMRadio::getFrequency`. The double-colon operator (`::`) is called the *scope resolution operator* and tells C++ where to look for the function we want to implement. You can think of the syntax `X::Y` as meaning "look inside `X` for `Y`." When implementing member functions, it is extremely important that you make sure to use the full name of the function you want to implement. If instead we had written the following:

```
double getFrequency() // Problem: Legal but incorrect
{
    /* ... implementation goes here ... */
}
```

Then C++ would think that we were implementing a regular function called `getFrequency` that has no relationship whatsoever to the `getFrequency` function inside of `FMRadio`.

Now that we've seen how to tell C++ that we're implementing the function, what code should we write inside of the function? We know that the function should return the FM radio's current frequency. Moreover, the frequency is stored inside of a data member called `frequency`. Consequently, we can write the following code for `FMRadio::getFrequency`:

```
double FMRadio::getFrequency()
{
    return frequency;
}
```

This may look a bit confusing, so let's take a second to think about what's going on here. This function is a single line, `return frequency`. If you'll notice, nowhere in the `getFrequency()` function did we define a variable called `frequency`, but this function still compiles and runs correctly. The reason is as follows – inside of a member function, all of the class's data members can be accessed by name. That is, when implementing the `getFrequency` function, we can freely access and manipulate any or all of the class's data members by referring to them by name. We don't need to indicate that `frequency` is a data member, nor do we have to specify *which* `FMRadio`'s `frequency` data member we're referring to. By default, C++ assumes that all data members are the data members of the receiver object, and so the line return frequency means "return the value of the `frequency` data member of the object on which this function was invoked."

At this point, let us more formally define what the public and private access specifiers actually mean. If a member of a class is marked public, then *any* part of the code can access and manipulate it. Thus if you have a public member function in the class interface, all code can access it. If a class member is marked private, then the only

pieces of the code that can access that member are the member functions of the class. That is, private data can only be read and written by the implementations of the class's member functions. In this sense, the public and private keywords are C++'s mechanism for defining interfaces and enforcing encapsulation. A class's interface is defined by all of its public members, and its implementation by the implementations of those public member functions along with any private data members. The compiler enforces encapsulation by disallowing class clients from directly accessing private data, and so the implementation can assume that any access to the class's private data goes through the public interface.

Let's conclude this section by implementing the remaining pieces of the `FMRadio` class. First, let's implement the `setFrequency` function, which sets the radio's frequency to a particular value. If you'll recall, all FM radio frequencies must be between 87.5 MHz and 108.0 MHz. Thus, we'll have this function verify that the new frequency is in this range, and will then set the frequency to be in that range if so. Here's one possible implementation:

```
void FMRadio::setFrequency(double newFreq)
{
    assert(newFreq >= 87.5 && newFreq <= 108.0);
    frequency = newFreq;
}
```

This is a remarkably simple two lines of code. We first assert that the frequency is in range, and then set the frequency data member of the class to the new value. What's so fantastic about this code is that it allows us to enforce the restriction that the frequency be constrained to the range spanned by 87.5 MHz to 108.0 MHz. Because the only way that clients can change the frequency data member is through the setFrequency function, we can prevent the frequency from ever being set to a value out of range. We'll discuss this in more detail when we talk about class invariants.

Using the implementation of the `get/setFrequency` functions as a basis, we can easily implement the `get/setVolume` functions. This is shown here:

```
int FMRadio::getVolume()
{
    return volume;
}

void FMRadio::setVolume(int newVol)
{
    assert(newVol >= 0 && newVol <= 10);
    volume = newVol;
}
```

This pattern of pairing a `get*` function along with a `set*` function is extremely common, and you will undoubtedly see it in any major C++ project you work on. We'll detail exactly why it is such a useful design later in this chapter.

The final three functions we wish to implement are the `setPreset`, `presetExists`, and `getPreset` functions. These functions are in some ways similar to the `get/setVolume` functions, but differ in that the values they read and write might not exist. We'll begin with `setPreset`, which is shown here:

```
void FMRadio::setPreset(int index, double freq)
{
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index] = freq;
}
```

The `presetExists` function can be implemented quite simply by returning whether the `map` contains the specified key. However, there's one detail we didn't consider – what happens if the index is out of bounds? That is, what do we do if the client asks whether preset 0 exists, or whether preset 137 exists? We could implement `presetExists` to so that it returns false in these cases (since there are no presets in those slots), but it seems more reasonable to have the function assert that the value is in bounds first. The reason is that if a client is querying whether a preset that is out of the desired range exists, it almost certainly represents a logical error. Using `assert` to check that the value is in bounds will let us debug the program more easily. This leads to the following implementation of `presetExists`:

```
bool FMRadio::presetExists(int index)
{
    assert(index >= 1 && index <= 6);
    return presets.find(index) != presets.end();
}
```

Finally, we'll implement the `getPreset` function. Since there is no meaningful value to return if the preset doesn't exist, we'll have this function verify that the preset is indeed valid before returning it. This is shown here:

```
double FMRadio::getPreset(int index)
{
    assert(presetExists(index));
    return presets[index];
}
```

Notice that in this function, we invoked the `presetExists` member function. As with private data members, C++ lets you call member functions of the receiver object without having to explicitly specify which object you are referring to. That is, the compiler is smart enough to tell that the call to `presetExists(index)` should be interpreted as "call the `presetExists` function on the receiver object, passing in the value `index`." This also brings up another important point: it is perfectly legal to use a class's public interface in its implementation. In fact, doing so is often a wise idea. If we had implemented `getPreset` without calling `presetExists`, we would have to duplicate a reasonable amount of code, which is in general a very bad idea.

Comparing `classes` and `structs`

Earlier in this chapter, we saw how representing the `FMRadio` as a `struct` led to all sorts of problems. The `struct` had no means of enforcing invariants, and any change to the `struct`'s fields could break a potentially unbounded amount of code. We discussed earlier at a high level how abstraction and encapsulation can prevent these problems from occurring. Does the class mechanism, which is designed to represent these ideas in software, prevent the aforementioned problems from happening? Let's take a few minutes to see whether this is the case.

Classes Enforce Invariants

An *invariant* is a property of a set of data that always holds true for that data. For example, one possible invariant might be that a certain value always be even, while another could be that the difference between two values is less than fourteen. In our example with `FMRadio`, our class had several invariants:

- The radio's frequency is always between 87.5 MHz and 108.8 MHz.
- The radio's volume is a value between 0 and 10, inclusive.
- The radio's presets are numbered between 1 and 6, and are valid frequencies.

The `struct` version of `FMRadio` failed to enforce any of these invariants because clients could go in and directly modify the fields responsible for holding the data. In the class version, however, any access to the data

members that represent these quantities must go through the appropriate `set*` and `get*` functions. This allows the implementation to double-check that all of the invariants hold before modifying the class's data members. For example, let's review the implementation of `setPreset`:

```
void FMRadio::setPreset(int index, double freq)
{
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index] = freq;
}
```

This is the only function in the interface that allows clients to modify the radio's presets. Before the function writes a value to one of the presets, it verifies that the index and frequency are in range. Consequently, if the data member is written to, it is only after the implementer has had a chance to inspect the value and confirm that it is indeed in range. In other words, by restricting access to the data members and instead providing a set of functions that modify the data members, the implementer can prevent clients from modifying the implementation in a way that violates the class invariants.

Classes Enforce Encapsulation

Recall that when we implemented the `FMRadio` as a `struct`, changing any of the fields would break existing code. The reason for this is that any manipulations of the `struct` required direct access to the fields of the `struct`. When using classes, however, all operations on the class must go through an additional layer – the interface – which is independent of the current implementation. For example, consider the following code:

```
FMRadio myRadio;

myRadio.setVolume(10);
cout << myRadio.getVolume() << endl;
```

This isn't the most exciting code we've written, but it illustrates how a client might read and set the radio's volume. Now, suppose that we are implementing the `FMRadio` class so that it interacts with a real set of speakers. Initially, you might think that the speaker volume is controlled by modifying how much power the speakers receive; at lower power, the speakers output less sound. In reality, though, most speaker volumes are controlled by modifying how much *attenuation* the sound signal receives. That is, when a speaker is at full volume, the attenuation level is zero, and the speaker plays the sound at maximum volume. When the volume is zero (the sound is muted), the attenuation level is 100% and the speakers produce no sound. In other words, the volume control is represented by determining how much attenuation to insert. Consequently, whenever we want to increase the volume, we would decrease the attenuation, and vice-versa. Given this description, we therefore might change the implementation of the `FMRadio` class so that the volume is represented internally as an attenuation amount. Here's the modified class:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    attenuation; // 0 is no attenuation, 10 is maximum attenuation
    map<int, double> presets;
};
```

Because we've changed the internal representation of the `FMRadio`, we will need to change the implementation of the `get`/`setVolume` functions. Now, these functions are designed so that the user inputs an amount of volume, not an amount of attenuation, and so the functions will have to do a quick behind-the-scenes calculation to convert between the two. Here's one possible implementation of `setVolume`:

```
void FMRadio::setVolume(int newVol)
{
    assert(newVol >= 0 && newVol <= 10); // Unchanged
    attenuation = 10 - newVol; // Convert from volume to attenuation level
}
```

Here, the code for `setVolume` takes in a volume level from the client, then converts it into an attenuation level by subtracting the volume from ten. This means that volume 10 corresponds to 0 attenuation, volume 7 to 3 attenuation, etc.

Now, how might we go about changing the implementation of `getVolume`? This function must return a volume between 0 and 10 with 0 meaning no volume and 10 meaning maximum volume, but we've implemented the volume level internally as the attenuation. This means that the function must do a quick calculation to convert between the two. The resulting implementation is shown here:

```
int FMRadio::getVolume()
{
    return 10 - attenuation;
}
```

I'll leave it as an exercise to the reader to verify that this computation is correct. ☺

In this short discussion, we completely changed the internal implementation of the radio volume. But from a client's perspective, absolutely nothing has changed. Recall the client code we wrote earlier on for changing the radio volume:

```
FMRadio myRadio;

myRadio.setVolume(10);
cout << myRadio.getVolume() << endl;
```

This code is still perfectly legal, and moreover it produces the exact same output as before. Because this code only uses the class's public interface, the client cannot tell that calling `myRadio.setVolume(10)` actually sets

an internal field in the `FMRadio` to zero, nor can she tell that calling `myRadio.getVolume()` will perform a conversion behind-the-scenes. In other words, using the public interface allows clients of `FMRadio` to write code that will compile and run correctly even if the entire implementation of the `FMRadio` has changed.

Class Constructors

One of the recurring themes of this chapter has been that classes can enforce invariants. However, using only the techniques we've covered so far, there are some invariants that classes cannot enforce automatically. To see this, let's return to the `FMRadio` class. If you'll recall, when implementing `FMRadio` using a `struct`, we saw that one possible implementation of the preset list was to use an array of six `double`s, where an unprogrammed preset has value 0.0. Let's modify our original implementation of the `FMRadio` class so that we use this implementation strategy. The new class looks like this:

```
class FMRadio
{
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

This, of course, necessitates that we change our implementation of `presetExists`, since we no longer represent the preset list as a map. The new implementation is shown here:

```
bool FMRadio::presetExists(int index)
{
    assert(index >= 1 && index <= 6);
    return presets[index - 1] == 0.0; // -1 maps [1, 6] to [0, 5]
}
```

Given this implementation, what is the result of running the following code snippet?

```
FMRadio myRadio;
if (myRadio.presetExists(1))
    cout << "Preset 1: " << myRadio.getPreset(1) << endl;
else
    cout << "Preset 1 not programmed." << endl;
```

Intuitively, this program should print out that preset one is not programmed, since we just created the radio. Unfortunately, though, this program produces undefined behavior. Here is the output from several different runs of the program on my machine:

```
Preset 1: 3.204e+108
Preset 1 not programmed.
Preset 1: -1.066e-34
Preset 1: 4.334e+20
```

This certainly doesn't seem right! What's going on here?

The problem is that all of the data members of `FMRadio` are primitive types, and unless you explicitly initialize a primitive type, it will hold whatever value happens to be in memory at the time that it is created. In particular, this means that the presets array will be filled with garbage, and so the `presetExists` and `getPreset` functions will be working with garbage data. Garbage data is never a good thing, but it is even more problematic from the standpoint of class invariants. The `FMRadio` assumes that certain constraints hold for its data members, but those data members are initialized randomly. How can `FMRadio` ensure that it behaves consistently when it does not have control over its implementation? The answer is simple: it can't, and we're going to need to refine our approach to make everything work correctly.

A Step in the Right Direction: `init()`

One way that we could fix this problem is to create a new member function called `init` that initializes all of the data members. We then require all clients of the `FMRadio` class to call this `init` function before using the other member functions of the `FMRadio` class. Assuming that clients ensure to call `init` before using the `FMRadio`, this should solve all of our problems.

Let's take a minute to see how we might implement the `init` function. First, we need to modify the class's public interface, as shown here:

```
class FMRadio
{
public:
    void   init();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

We could then implement `init` as follows:

```
void FMRadio::init()
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
}
```

This is certainly a step in the right direction. We no longer have to worry about the presets array containing uninitialized values. But what of the other data members, frequency and volume? They too must be initialized to some meaningful value. We can therefore update the `init` function to set them to some reasonable value. For simplicity, let's set the frequency to 87.5 MHz (the minimum possible frequency) and set the volume to five. This is shown here:

```
void FMRadio::init()
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}
```

It may seem strange that we have to initialize frequency and volume inside of the `init` function. After all, why can't we do something like this?

```
class FMRadio
{
public:
    void   init();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency = 87.5;  // Problem: Not legal C++
    int    volume    = 5;     // Problem: Not legal C++
    double presets[6];
};
```

Unfortunately, this is not legal C++ code. There isn't a particularly good reason why this is the case, and in the next release of C++ this syntax will be supported, but for now we have to manually initialize everything in the `init` function.

Why `init()` is Insufficient

The approach we've outlined above seems to solve all of our problems. Every time that we create an `FMRadio`, we manually invoke the `init` function. This solves our problem, but puts an extra burden on the client. In particular, if a client does not call the `init` function, our object's internals will not be configured properly and any use of the object will almost certainly cause some sort of runtime error.

The problem with `init` is that it does not make logical sense. When you purchase a physical object, most of the time, that object is fully assembled and ready to go. When you buy a stapler, you don't buy the component parts and then assemble it; you buy a finished product. You don't purchase a car and then manually connect the transmission to the rest of the engine; you assume that the car manufacturer has done this for you. In other words, by the time that you begin using an object, you expect it to be assembled. From the standpoint of physical objects, this is because you are buying a logically complete object, not a collection of components. From the standpoint of abstraction, this is because it breaches the wall of abstraction if you are required to set up an object into a well-formed state before you begin using it.

None of the objects we've seen so far have required any function like `init`. The STL `vector` and `map` are initialized to sensible defaults before you begin using them, and `string`s default to holding the empty string without any explicit intervention by the user. But how do they do this? It's through the magic of a special member function called the constructor.

Class Constructors

A *constructor* is a special member function whose job is to initialize the object into a well-formed state before clients start manipulating that object. In this sense, constructors are like the `init` function we wrote earlier. However, constructors have the special property that they are called automatically whenever an object is constructed. That is, if you have a class that defines a constructor, that constructor is guaranteed to execute whenever you create an object of the class type.

Syntactically, a constructor is a member function whose name is the same as the name of the class. For example, the string constructor is a function named string::string, and in our `FMRadio` example, the constructor is a member function named `FMRadio::FMRadio`. Here is a refined interface for `FMRadio` that includes a class constructor:

```
class FMRadio
{
public:
    FMRadio();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

Notice that the constructor has no return type, not even void. This may seem strange at first, but will make substantially more sense once you see how and where the constructor is invoked.

Syntactically, one implements a constructor just as one would any other member function. The only difference is that the constructor does not have a return type, and so the syntax for implementing a constructor looks like this:

```
FMRadio::FMRadio()
{
    /* ... implementation goes here ... */
}
```

Constructors are like any other function, and so we can put whatever code we feel like in the body of the constructor. However, the constructor should ensure that all of the object's data members that need manually initialization are manually initialized. In our case, this means that we might implement the `FMRadio` constructor as follows:

```
FMRadio::FMRadio()
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}
```

Now, whenever we create an instance of the `FMRadio` type, the object will be set up correctly. That is, when we write code like this:

```
FMRadio myRadio;
if (myRadio.presetExists(1))
    cout << "Preset 1: " << myRadio.getPreset(1) << endl;
else
    cout << "Preset 1 not programmed." << endl;
```

The output will always be "Preset 1 not programmed." This is because in the line where we create the `myRadio` object, C++ automatically invokes the constructor, which zeros out all of the presets.

It is illegal to call a class's constructor; C++ will always do this for you. For example, the following code will not compile:

```
FMRadio myRadio;
myRadio.FMRadio(); // Problem: Cannot manually invoke constructor
```

This may seem like an unusual restriction, but is actually quite useful. Because the constructor is invoked when and only when the class is being constructed for the first time, you don't need to worry about unusual conditions where the class is being instantiated but meaningful data is already stored in the class. Additionally, this makes the role of the constructor explicitly clear – its job is to initialize the class to a meaningful state, nothing more. Second, as a consequence, constructors can never return values. The constructor is invoked automatically, not giving you a chance to store a returned value even if one were to exist.

Arguments to Constructors

In the above example, our `FMRadio` constructor takes in no parameters. However, it is possible to create constructors that take in arguments that might be necessary for initialization. For example, our `FMRadio` constructor arbitrarily sets the frequency to 87.5 MHz and the volume to 5 because we need these values to be in certain ranges. There's no particular reason why we should initialize these values this way, but in the absence of information about what the client wants to do with the object we cannot do any better. But what if the client could tell us what she wanted the frequency and volume to be? In that case, we could initialize the frequency and volume to the user's values, in essence creating a radio whose frequency and volume were already set up for the user. To do this, we can create a second `FMRadio` constructor that takes in a frequency and volume, then initializes the radio to those settings.

Syntactically, a constructor of this sort is a member function named `FMRadio` that takes in two parameters. This is shown here:

```
class FMRadio
{
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

We could then implement this function as follows:

```
FMRadio::FMRadio(double freq, int vol)
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}
```

Now that we have this constructor, how do we call it? That is, how do we create an object that is initialized using this constructor? The syntax for this is reasonably straightforward and looks like this:

```
FMRadio myRadio(88.5, 5);
```

That is, we write out the type of the object to create, the name of the object to create, and then a parenthesized list of the arguments to pass into the constructor.

You may be wondering why in the case of a zero-argument constructor, we do not need to explicitly spell out that we want to use the default constructor. In other words, why don't we write out code like this:

```
FMRadio myRadio(); // Problem: Legal but incorrect
```

This code is perfectly legal, but it does not do what you'd expect. There is an unfortunate defect in C++ that causes this statement to be interpreted as a function prototype rather than the creation of an object using the default constructor. In fact, C++ will interpret this as "prototype a function called `myRadio` that takes in no arguments and returns an `FMRadio`" rather than "create an `FMRadio` called `myRadio` using the zero-argument constructor." This is sometimes referred to as "C++'s most vexing parse" and causes extremely difficult to understand warnings and error messages. Thus, if you want to invoke the default constructor, omit the parentheses. If you want to invoke a parametrized constructor, parenthesize the arguments.

Another important point to remember when working with multiple constructors is that constructors cannot invoke one another. This is an extension of the rule that you cannot directly call a constructor. If you need to do the same work in multiple constructors, you can either duplicate the code (yuck!) or use a private member function, which we'll discuss later.

Classes Without a Nullary Constructor

A function is called *nullary* if it takes no arguments. For example, the first `FMRadio` constructor we wrote is a nullary constructor, since it takes no arguments. If you define a class and do not provide a constructor, C++ will automatically provide you a default nullary constructor that does absolutely nothing. This is why in the case of `FMRadio`, we needed to provide a nullary constructor to initialize the data members; otherwise they would initialize to arbitrary values. However, if you define a class and provide any constructors, C++ will not automatically generate a nullary constructor for you. This means that it is possible to construct classes that do not have a zero-argument constructor. For example, suppose that we remove the nullary constructor from `FMRadio`; this results in the following class definition:

```
class FMRadio
{
public:
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

Because this class does not have a nullary constructor, we cannot construct instances of it without passing in values for the frequency and volume. That is, the following code is illegal:

```
FMRadio myRadio; // Problem: No default constructor available
```

At first, this may seem like a nuisance. However, this aspect of class design is extremely valuable because it allows you to create types that must be initialized to a meaningful value. For example, suppose that you are designing a class that represents a robot-controlled laser, either for automated welding or delicate surgical procedures. When building such a laser, it is imperative that the laser know how much power to deliver and what points the beam should be directed at. These values absolutely must be initialized to meaningful data, or the laser might deliver megawatts of power at a patient or aim at random points firing the beam. If you wanted to represent the laser as a C++ class, you could force clients to specify this data before using the laser by making a `RobotLaser` class whose only constructor takes in both the laser power and laser coordinates. This means that clients could not create instances of `RobotLaser` without entering coordinates, reducing the possibility of a catastrophic failure.

Private Member Functions

Let's return once again to our `FMRadio` example, this time looking at the implementation of three functions: `setFrequency`, `setPreset`, and `presetExists`. The implementations of these functions are shown here:

```
void FMRadio::setFrequency(double newFreq)
{
    assert(newFreq >= 87.5 && newFreq <= 108.0);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq)
{
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index - 1] = freq;
}

double FMRadio::presetExists(int index)
{
    assert(index >= 1 && index <= 6);
    return presets[index - 1] == 0.0;
}
```

Notice that each highlighted line of code appears in two of the three functions. Normally this isn't too serious a concern, but in this particular case makes the implementation brittle and fragile. In particular, if we ever want to change the number of presets or the maximum frequency range, we'll need to modify multiple parts of the code accordingly or risk inconsistent handling of presets and frequencies. To unify the code, we might consider decomposing this logic into helper functions. However, since the code we're decomposing out is an implementation detail of the `FMRadio` class, class clients shouldn't have access to these helper functions. In other words, we want to create a set of functions that simplify class implementation but which can't be accessed by class clients. For situations like these, we can use a technique called private member functions.

Marking Functions private

If you'll recall from earlier, the `private` keyword indicates which parts of a class cannot be accessed by clients. So far we have restricted ourselves to dealing only with private data members, but it is possible to create member functions that are marked private. Like regular member functions, these functions can read and write private class data, and are invoked relative to a receiver object. Unlike public member functions, though, they can only be invoked by the class implementation. Therefore, private member functions are not part of the class's interface and exist solely to simplify the class implementation.

Declaring a private member function is similar to declaring a public member function - we just add the definition to the class's private data. In our `FMRadio` example, we will introduce two helper functions: `checkFrequency`, which asserts that a frequency is in the proper range, and `checkPreset`, which ensures that a preset index is in bounds. The updated class definition for `FMRadio` is shown here:

```
class FMRadio
{
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);

    double frequency;
    int    volume;
    double presets[6];
};
```

We can then implement these functions as follows:

```
void FMRadio::checkFrequency(double freq)
{
    assert(freq >= 87.5 && freq <= 108.8);
}

void FMRadio::checkPreset(int index)
{
    assert(index >= 1 && index <= 6);
}
```

Using these functions yields the following implementations of the three aforementioned functions:

```
void FMRadio::setFrequency(double newFreq)
{
    checkFrequency(newFreq);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq)
{
    checkPreset(index);
    checkFrequency(freq);
    presets[index - 1] = freq;
}

bool FMRadio::presetExists(int index)
{
    checkPreset(index);
    return presets[index - 1] == 0.0;
}
```

These functions are significantly cleaner than before, and the class as a whole is much more robust to change.

Simplifying Constructors with Private Functions

Private functions can greatly reduce the implementation complexity of classes with multiple constructors. Recall that our `FMRadio` class has two constructors, one which initializes the `FMRadio` to have a reasonable default frequency and volume, and one which lets class clients specify the initial frequency and volume. The implementation of these two functions is shown here:

```
FMRadio::FMRadio()
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}

FMRadio::FMRadio(double freq, int vol)
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}
```

These functions are extremely similar in structure, but because C++ does not allow you to manually call a class's constructor. How, then, can the two functions be unified? Simple – we introduce a private member function which does the initialization, then have the two constructors invoke this member function with the proper arguments. This is illustrated below:

```
class FMRadio
{
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);
    void   initialize(double freq, int vol);

    double frequency;
    int    volume;
    double presets[6];
};
```

```
void FMRadio::initialize(double freq, int vol)
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}

FMRadio::FMRadio()
{
    initialize(87.5, 5);
}

FMRadio::FMRadio(double freq, int vol)
{
    initialize(freq, vol);
}
```

As you can see, private member functions are extremely useful tools. We will continue to use them throughout the remainder of this book, just as you will undoubtedly use them in the course of your programming career.

Partitioning Classes Across Files

One of the motivations behind classes was to provide a means for separating out implementation and interface. We have seen this already through the use of the `public` and `private` access specifiers, which prevent clients from looking at implementation-specific details. However, there is another common means by which implementation is separated from interface, and that is the split between *header files* and *implementation files*. As you've seen before, almost all of the programs you've written begin with a series of `#include` directives which tell the compiler to fetch certain files and include them in your programs. Now that we've reached a critical mass and can begin writing our own classes, we will see how to design your own header files.

At a high level, header files provide a means for exporting a class interface without also exporting unnecessary implementation details. Programmers who wish to use your class in their code can `#include` the header file containing your class declaration, and the linker will ensure that the class implementation is bundled along with the final program. More concretely, a header file contains the class declaration (including both `public` and `private` members), and the implementation file contains the actual class implementation. For example, suppose that we want to export the `FMRadio` class we've just designed in a header/implementation pair. We'll begin by constructing the header file. Traditionally, header files that contain class declarations have the same name as the class and a .h suffix. In our case, this means that we'll be creating a file called `FMRadio.h` that contains our class definition. This is shown here:

File: *FMRadio.h*

```
#ifndef FMRadio_Included
#define FMRadio_Included

class FMRadio
{
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);
    void   initialize(double freq, int vol);

    double frequency;
    int    volume;
    double presets[6];
};

#endif
```

Notice that we've surrounded the header file with an include guard. In case you've forgotten, the include guard is a way to prevent compiler errors in the event that a client `#include`s the same file twice; see the chapter on the preprocessor for more information. Beyond this, though, the header contains just the class interface.

Now that we've built the .h file for our `FMRadio` class, let's see if we can provide a working implementation file. Typically, an implementation file will have the same name as the class, suffixed with the .cpp extension.* Appropriately, we'll name this file `FMRadio.cpp`. Unlike a .h file, which is designed to export the class declaration to clients, the .cpp file just contains the implementation of the class. Here's one possible version of the `FMRadio.cpp` file:

* It is also common to see the .cc extension. Older code might use the .C extension (capital C) or the .c++ extension.

File: FMRadio.cpp

```
#include "FMRadio.h"

FMRadio::FMRadio()
{
    initialize(87.5, 5);
}

FMRadio::FMRadio(double freq, int vol)
{
    initialize(freq, vol);
}

void FMRadio::initialize(double freq, int vol)
{
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}

void FMRadio::checkFrequency(double freq)
{
    assert(freq >= 87.5 && freq <= 108.8);
}

void FMRadio::checkPreset(int index)
{
    assert(index >= 1 && index <= 6);
}

double FMRadio::getFrequency()
{
    return frequency;
}

void FMRadio::setFrequency(double newFreq)
{
    checkFrequency(newFreq);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq)
{
    checkPreset(index);
    checkFrequency(freq);
    presets[index - 1] = freq;
}

bool FMRadio::presetExists(int index)
{
    checkPreset(index);
    return presets[index - 1] == 0.0;
}

double FMRadio::getPreset(int index)
{
    checkPreset(index);
    return presets[index - 1];
}
```

There are a few important aspects of this .cpp file to note. First, notice that at the top of the file, we `#include`d the .h file containing the class declaration. This is *extremely important* – if we don't include the header file for the class, when the C++ compiler encounters the implementations of the class's member functions, it won't have seen the class declaration and will flag all of the member function implementations as errors. Second, note that when `#include`-ing the .h file, we surrounded the name of the file in "double quotes" instead of <angle brackets.> If you'll recall, this is because the preprocessor treats include directives using <angle brackets> as instructions to look for standard library files, and since the classes you'll be writing aren't part of the standard library you'll want to use "double quotes" instead.*

Now that we've partitioned the class into a .h/.cpp pair, we can write programs that use the class without having to tediously copy-and-paste the class definition and implementation. For example, here's a short program which manipulates an `FMRadio`. Note that we never actually see the declaration or implementation of the `FMRadio` class; `#include`-ing the header provides the compiler enough information to let us use the `FMRadio`.

```
#include <iostream>
#include "FMRadio.h"
using namespace std;

int main()
{
    FMRadio myRadio;
    myRadio.setFrequency(88.5);
    myRadio.setVolume(8);
    /* ... etc. ... */
}
```

Throughout the remainder of this book, whenever we design and build classes, we will assume that the classes are properly partitioned between .h and .cpp files.

Chapter Summary

- Software systems are often on the order of millions of lines of code, far larger than even the most competent programmers can ever keep track of at once.
- A single incorrect value in a software system can cause that entire system to fail.
- The maximum number of possible interactions in a software system grows exponentially in the number of components of that system.
- Abstractions give a way to present a complex object in simpler terms.
- Abstractions partition users into clients and implementers, each with separate tasks. This separation is sometimes referred to as the wall of abstraction.
- Abstractions describe many possible implementations, and encapsulation prevents clients from peeking at that implementation.
- The way in which a client interacts with an object is called that object's interface.
- Abstraction reduces the number of components in a software system, reducing the maximum complexity of that system.

---

* If you ever have the honor of getting to write a new standard library class, please contact me... I'd love to offer comments and suggestions!

- C++ `structs` lack encapsulation because their implementation is their interface.
- The C++ class concept is a realization of an interface paired with an implementation.
- The members of a class that are listed public form that class's interface and are accessible to anyone.
- The members of a class that are listed private are part of the class implementation and can only be viewed by member functions of that class.
- Constructors allow implementers to enforce invariants from the moment the class is created.
- Private member functions allow implementers to decompose code without revealing the implementation to clients.
- Class implementations are traditionally partitioned into a .h file containing the class definition and a .cpp file containing the class implementation.
- Design class interfaces before implementations to avoid overspecializing the interface on an implementation artifact.

Practice Problems

1. In our discussion of abstraction, we talked about how interfaces and modularity can exponentially reduce the maximum complexity of a system. Can you think of any examples from the real world where introducing indirection makes a complex system more manageable? *(Hint: Think about a republican government, or layers of management in a company, etc...)*
2. What is the motivation behind functions along the lines of `getFrequency` and `setFrequency` over just having a public `frequency` data member?
3. When is a constructor invoked? Why are constructors useful?
4. What is the difference between a public member function and a private member function?
5. What goes in a class's .h file? In its .cpp file?
6. We've talked at length about the streams library and STL without mentioning much of how those libraries are implemented behind-the-scenes. Explain why abstraction makes it possible to use these libraries without full knowledge of how they work.
7. Explain why C strings have almost no abstraction. That is, think about the difference (or lack of a difference) between the way C strings are implemented and the way clients manipulate C strings.
8. Suppose that C++ were designed somewhat differently in that data members marked private could only be read but not written to. That is, if a data member called `volume` were marked private, then clients could read the value by writing `myObject.volume`, but could not write to the `volume` variable directly. This would prohibit clients of a class from modifying the implementation incorrectly, since any operations that could change the object's data members would have to go through the public interface. However, this setup has a serious design flaw that would make class implementations difficult to change. What is this flaw? *(Hint: Think back to the `volume/attenuation` example from earlier)*

9. Below is an interface for a class that represents a grocery list:

```
class GroceryList
{
public:
    GroceryList();

    void addItem(string quantity, string item);
    void removeItem(string item);

    string itemQuantity(string item);
    bool itemExists(string item);
};
```

   The `GroceryList` constructor sets the grocery list to contain no items. The `addItem` function adds a certain quantity of an item to the grocery list. For example, the call

```
gList.addItem("One Gallon", "Milk");
```

   would add the item "Milk" to the list with quantity "One Gallon." If the item already exists in the list, then `addItem` should replace the original quantity with the new quantity.

   The `removeItem` function should delete the specified item off of the shopping list. `itemExists` returns whether the specified item exists in the shopping list, and `itemQuantity` takes in an item and returns the quantity associated with it in the list. If the item doesn't exist, `itemQuantity` can either raise an `assert` error or return the empty string.

   Provide an implementation for the `GroceryList` class. You are free to use whatever implementation you feel is best, and can implement the member functions as you see fit. However, you might find it useful to use a `map<string, string>` to represent the items in the list.

10. What is the advantage of making a `GroceryList` class over just using a raw `map<string, string>`?

11. Does the `GroceryList` class need a constructor? Why or why not?

12. Give an example of a parameterized constructor you have encountered in the STL.

13. Why are parameterized constructors useful?

14. *Keno* is a popular gambling game with similarities to a lottery or bingo. Players place a bet and pick a set of numbers between 1 and 80, inclusive. The number of numbers chosen can be anywhere from one to twenty, with each having a different payoff scale. Once the players have chosen their numbers, twenty random numbers between 1 and 80 are chosen, and players receive a payoff based on how many numbers they picked that matched the chosen numbers. For example, if a player picked five numbers and all five were chosen, she might win around $1,000 for a one- or two-dollar bet. The actual payoffs are based on the probabilities of hitting *k* numbers out of *n* chosen, but this is irrelevant for our discussion.

    Suppose that you are interested in writing a program that lets the user play Keno. You are not interested in the payoffs, just letting the user enter numbers and reporting which of the user's numbers came up. To do this, you decide to write a class `KenoGame` with the following interfaces:

```
class KenoGame
{
public:
    KenoGame();

    void addNumber(int value);
    size_t numChosen();

    size_t numWinners(vector<int>& values);
};
```

The `KenoGame` constructor initializes the class however you see fit. `addNumber` takes in a number from the user and adds it to the set of numbers the user guessed. The `numChosen` member function returns how many numbers the user has picked so far. Finally, the `numWinners` function takes in a `vector<int>` corresponding to the numbers that were chosen and returns how many of the user's numbers were winners.

Write an implementation of the `KenoGame` class.

15. Refer back to the implementation of *Snake* from the chapter on STL containers. Design and implement a class that represents a snake. What operations will you support in the public interface? How will you implement it?