Lesson Four: Supporting Encoders and Decoders

The assignment that concludes this lesson will have mostly the same inputs and outputs as assignment 3. What varies is that we'll now be using a much more sophisticated means of decoding and encoding bitmap graphics. At the end of this lesson, our bitmap handling will support the following.

- Client defined file format support for both decoding (reading) and coding or encoding (writing).
- File extension independent decoder determination for maximum flexibility and resilience.
- Object-oriented iterators for implementation independence and run-time decoration.
- Support for deferred loading and decoding. This allows us to load and decode bitmaps on their first use, which can be convenient for remote images and slow formats that are slow to decompress.
- Flexible, iterator-based encoding and decoding.

Required Reading

- *Design Patterns: Element of Reusable Object-Oriented Software*: The Decorator (175), Prototype (117) and Iterator (257) Patterns.

In designing and implementing our more robust bitmap support, we will find limited reuse from the last lesson. This is because our design this time will be far more thorough and more object-oriented overall. For the most part, our last design followed the basic paradigm of development that the C++ standard library uses. This paradigm tends to push most things into compile time and to work primarily with simple, concrete classes. This works as a primitive layer on which to build more sophisticated things in many cases, but it's rare that this sort of design mentality survives the requirements of realistic full-scale systems (which, for example, often need to support many file formats). The difference between these two paradigms will become apparent throughout this lesson.

At the end of the lesson, you should reflect on the application built in Lesson Three that performs an almost identical task (for our test), and appreciate the many improvements we've made. Our new system is designed for the future. With not much more work, the new system could be a stable infrastructure for encoding and decoding a variety of bitmap image file formats (such as GIF, TIFF and JPG) for many years.

Lesson Objective

By the end of this lesson, you will be able to implement the CODEC support and the new Bitmap class.

The Design and Role of CODECs in Modern Systems

The industry term ***CODEC*** comes from "code/decode" (similar to *MODEM* which comes from "Modulator/Demodulator"). Both are extremely generic terms. A CODEC facilitates the conversion of information to and from a different format. You may have seen CODECs mentioned when working with audio, video, file compression, or any of a wide variety of applications.

Most modern systems allow CODECs to be dynamically added. New CODECs can be dropped into many multimedia applications, such as Windows Media Player, without having to get a new version of the application, and often without even having to restart the application. The applications (and in some cases the OS) maintain a registry of CODECs that is used to look up the appropriate CODEC for incoming information. This has certainly simplified the variety of formats that are out there. In the earlier days of computing, file formats were more deeply rooted in applications. Now some applications and operating systems even support automatic location and installation of new CODECs via the Web when an unknown file is opened.

Key Term

- CODEC

One of the important properties of the design of most systems that work with CODECs is that they are not dependent on the concept of a file or file extension. Most systems (including the Windows OS) look up the appropriate CODEC based on the raw incoming information. This is why the newer versions of Microsoft Windows can load a Windows Bitmap Format image, even if it is renamed to have a JPG extension. Instead of determining the CODEC to use by extension, it determines based on header information.

We won't delve too deeply into the existing technologies, but will write our own, based on the overall idea and intention of drop-in file format support. This is because this course is limited to the C++ Standard Library and can't reap the benefits of what's available in the industry.

Defining the Decoder

We'll separate encoders and decoders for obvious reasons: they each will have dramatically different implementations. Their only arguable codependence is at the conceptual level and perhaps the sharing of a few constants. Having each reuse the WindowsBitmapHeader class will simplify things to the point where the only remaining constant would be the double word alignment value—which you can make efforts to share by creating a constant in a shared namespace or simply redefine as a frozen class attribute (since this would be less exposed, it might be a better idea).

For the decoder's genericity, it will need to work with streams instead of files, of course. Certainly, we'd like to be able to support a future that includes HTTP stream of a bitmap located on a server. We'll also need our decoders to report whether or not they will be able to support a given format. To do this, we can pass in the first chunk of the incoming information to the decoder. For our bitmap decoder, it needs only to read the first two bytes of this chunk to determine whether it can support the format. Our decoder needs a way to get the converted information to the client. An iterator should do just the trick, allowing the client to iterate through the converted information—though this doesn't imply that we'll have to convert the information before the iteration starts. We can actually defer the decoding until the iteration, and only decode pixel by pixel as the client iterates across scan lines. When a client requests to increment to the next scan line, we simply read any double word alignment bytes; when a client requests to increment to the next pixel, we simply read in

In reality, we'd probably end up creating a more object-oriented (and interface based) stream to support true independence of location. Clients are sure to be able to implement any possible stream and would not be bound to iostream's

the next RGB color value.

implementation choices if we took an interface-based approach.

One other issue that both decoders and encoders will have to address is reporting which MIME type they support. A MIME type is a string, such as "text/html," that is a standardized way of referring to a particular format. Windows Bitmaps use "image/x-ms-bmp" as their mime type, so our WindowsBitmapDecoder implementation (clearly the decoder will be an interface) will have to report that as its MIME type.

One of the more complicated aspects of implementing our decoders will be the ability for a decoder to report whether or not it supports a chunk of information. We can create a documented standard that all bitmaps will be given a 100-byte chunk of the incoming information (or the total file size if it is less than 100 bytes) to determine whether or not they can support it. This is actually a common approach. Most, if not all, file formats are recognizable within the first 100 bytes.

Unfortunately, the decoder cannot be passed a stream for support determination because streams lose information as you read it, so the next decoder in the list would get a partially-eaten stream. Instead, we'll have to give the first chunk to the stream as a string. Once a decoder reports that it supports the information in the string, the entire stream can be handed to it.

Defining the Encoder

The encoder is for the most part identical to the decoder—just inversed. The only major difference is a simplification: it needn't be able to determine support for incoming information, because there is no incoming information, only outgoing. For the actual writing out to a stream, the encoder will need an iterator. This is precisely the inverse of the decoder. It also implies that—rather than a `createIterator` method—the encoder would have an `encodeToStream` method. Where the decoder is created with a stream and produces an iterator, the encoder is created with an iterator and produces (or rather, encodes to) a stream.

Creating the CODEC Library

Up till this point we have ignored how, exactly, the decoders and encoders are created. Based on the conversation, the decoders and encoders are tightly bound to a stream and a single bitmap to decode. So it is clear that upon creation, the decoder will need a reference to a stream. But how exactly does this creation occur?

(Recall from the *C++: Intermediate* course that in factory-like situations we can use the prototype pattern when we need to allow clients to dynamically define and parameterize implementations or subtypes to create in factory-like situations.) In *C++: Intermediate*, we used a registry of prototypes that could be searched based on the tag they claimed to support. With our CODECs, we have a similar situation.

If we use the prototype pattern, we'll end up with a CodecLibrary class that allows us at least to look up the appropriate encoder and decoder for a MIME type. Of course, we generally don't want to look up decoders by MIME type, but instead, by their support determination method. Also, both decoders and

encoders are tightly coupled to their incoming data; the decoder should be created along with a stream when it is found, and the encoder should be created along with an iterator when it is found. All of this information leads to the class definition shown in example 4.1.

Code Example 4.1

```
class CodecLibrary {
        public:
                void registerEncoder (HBitmapEncoder const& decoder);
                void registerDecoder (HBitmapDecoder const& decoder);

                // provide a mime type version and an
                // auto determination version of createDecoder
                HBitmapDecoder createDecoder (std::istream& sourceStream);
                HBitmapDecoder createDecoder (std::string const& mimeType,
                        std::istream& sourceStream);

                HBitmapEncoder createEncoder (std::string const& mimeType,
                        HBitmapIterator const& bitmapIterator);
};
```

Note that `create` was used rather than `find` or `get` because it emphasizes that the object being returned will be new each time, and that the parameters you're passing in are parameters to its creation.

The repercussions of the decoders creation means that the CODEC Library can be responsible for getting the first chunk from the file. This shouldn't be complicated, but the fact that the first chunk will be removed from the stream will make the clone method need to take the form of `clone (std::string const& firstChunk, std::istream& sourceStream)`. It will also make implementing the decoder a bit tricky, as it will need to work initially from a string, then secondly from a stream. Of course, for our WindowsBitmapDecoder, the entire header can be read before this distinction need be made, since its size is only 54 bytes. After that, it can provide a `getNextByte` method to abstract itself from when and how the switching between the string and the stream occurs. Note that stringstreams would have to be used to achieve all this cleanly, particularly since the WindowsBitmapHeader you are creating will require a stream, not a string.

Creating the Bitmap Iterator

We still have yet to rigorously define our bitmap iterator. We know that it should be scan-line based, and we know that we'd ideally like it in one class, not a separate pixel and scan-line iterator (since the scan-line iterator is fairly useless outside of a means to get to new pixel iterators). Making a single iterator will require more distinction in the method names. We'll also have to create an interface for this iterator, since different encoders will provide different implementations.

Code Example 4.2

```
class IBitmapIterator {
   public:
                virtual void nextScanLine () = 0;
                virtual bool isEndOfImage () const = 0;
                virtual void nextPixel () = 0;
                virtual bool isEndOfScanLine () const = 0;

                virtual Color getColor () const = 0;

                virtual int getBitmapWidth () const = 0;
                virtual int getBitmapHeight () const = 0;
};
```

As you can see, we've effectively removed the need for an "end" iterator. Although it would be possible to provide an end iterator, it would have to be a different implementation of the interface, and employ some trickery. The semantics would also be especially confusing because of the need to differentiate between the end of the scan line and the end of the image as a whole. The iterator we've provided is much closer to those in the examples in *Design Patterns: Elements of Reusable-Object Oriented Software* and, not coincidentally, those provided in the Java standard libraries. The idea of an "end" iterator is not common or suggested with most object-oriented systems. The "end" iterator comes from thinking of iterators more like pointers (as covered in the last lesson) or being overly oriented towards objects that behave like primitive types.

It may not be apparent to you why the bitmaps width and height are reported by the iterator. First, the information is a constant throughout the life of an iterator. If it changed from scan line to scan line, it would not be a valid bitmap. Second, reporting the width and height in the iterator allows us to potentially create a stretching decorator for the iterator, as will be apparent in the next section. The width and height will also be required by most things that work with the color information. For example, our encoder would require the width and height to be passed to it along with the iterator upon creation otherwise.

It will be your job in assignment 4 to provide the various necessary implementations of iterators. These will, of course, be internal iterators—though no client will be aware of anything beyond the basic IBitmapIterator interface.

Creating Bitmap Iterator Decorators

It is possible to apply effects on the fly by decorating an iterator. For example, we can create a brightness-effect decorator, or an invert-color decorator. As you learned in the past with the decorator pattern, it is possible to repeat decorations and arbitrarily layer them—all the while still making the original decorated object still seem the same.

As an example, consider a brightness decorator. For brightness, all we need to do is increase or decrease each color component (red, green, and blue) an equal amount. When a color component reaches a threshold point (0 or 255), we can no longer increment it, and the color as a whole will become

closer to black-and-white (which occurs when the red, green, and blue all have the same value). Our brightness decorator can be defined as shown in example 4.3.

Code Example 4.3

```
class BrightnessDecorator : public IBitmapIterator {
        public:
                BrightnessDecorator (HBitmapIterator const& originalIterator)
                        : originalIterator (originalIterator), brightnessAdjustment (0) {
                }

                void setBrightnessAdjustment (int brightnessAdjustment) {
                        this->brightnessAdjustment = brightnessAdjustment;
                }

                int getBrightnessAdjustment () const {
                        return this->brightnessAdjustment;
                }

                void nextScanLine () {
                        originalIterator->nextScanLine ();
                }

                bool isEndOfImage () const {
                        return originalIterator->isEndOfImage ();
                }

                void nextPixel () {
                        originalIterator->nextPixel ();
                }

                bool isEndOfScanLine () const {
                        return originalIterator->isEndOfScanLine ();
                }

                Color getColor () const {
                        Color const oldColor = originalIterator->getColor ();
                        int red = oldColor.getRed () + brightnessAdjustment;
                        if (red > 255) {
                                red = 255;
                        } else if (red < 0) {
                                red = 0;
                        }

                        int green = oldColor.getGreen () + brightnessAdjustment;
```

```
                        if (green > 255) {
                                green = 255;
                        } else if (green < 0) {
                                red = 0;
                        }
                        int blue = oldColor.getBlue () + brightnessAdjustment;
                        if (blue > 255) {
                                blue = 255;
                        } else if (blue < 0) {
                                blue = 0;
                        }

                        return Color (red, green, blue);
                }

        private:
                int brightnessAdjustment;
                HBitmapIterator originalIterator;
};
```

Of course, there's an error in the code in example 4.3. Since the adjustment of the red, green, and blue components is complicated and redundant—resulting in a high likelihood of error—the `else if (green < 0)` statement incorrectly adjusts red. This is just the sort of error that would be likely to happen, and just the reason to prefer idiomatic code. If we replace the `if` statements (which you probably copied and pasted) with an `adjustColorComponent` method, we'd gain a bit more isolation and readability of code, as in example 4.4 below.

Code Example 4.4

```
class BrightnessDecorator : public IBitmapIterator {
        public:
                // ...

                Color getColor () const {
                        Color adjustedColor = bitmapIterator->getColor ();
                        adjustedColor.setRedLevel (
                                adjustColorcomponent (adjustedColor.getRed ());
                        adjustedColor.setGreenLevel (
                                adjustColorcomponent (adjustedColor.getGreen ());
                        adjustedColor.setBlueLevel (
                                adjustColorcomponent (adjustedColor.getBlue ());

                        return adjustedColor;
                }
```

```
        private:
                static int adjustColorComponent (int colorComponent) {
                        int adjustedColorComponent = colorComponent +
                                brightnessAdjustment;
                        if (adjustedColorComponent > 255) {
                                adjustedColorComponent = 255;
                        } else if (adjustedColorComponent < 0) {
                                adjustedColorComponent = 0;
                        }
                }
};
```

Of course, this same sort of operation is bound to come up again. It's not that uncommon to need to increment or decrement a value while keeping it restricted by a specified range. This is just the sort of thing generic algorithms specialize in. If we create a generic rangedAdd algorithm, we can make sure we only ever have to write this code once. Since specifying many parameters in the function call can be awkward and complicated, we can pass the range in as template parameters.

Code Example 4.5

```
template <class Number, Number lowerLimit, Number upperLimit>
Number rangedAdd (Number firstNumber, Number secondNumber) {
        Number result = firstNumber + secondNumber;
        if (result > upperLimit) {
                result = upperLimit;
        } else if (result < lowerLimit) {
                result = lowerLimit;
        }

        return result;
}

class BrightnessDecorator : public IBitmapIterator {
        public:
                // ...

        private:
                static int adjustColorComponent (int colorComponent) {
                        return rangedAdd<int, 0, 255> (colorComponent +
                                brightnessAdjustment);
                }
};
```

Of course, there's more than one solution. Another, safer method would be to create a `ranged_number` template class that allowed us to work with a ranged number exactly as we worked with normal numbers, but with automatic range enforcement. If we did this, we'd have the choice of specifying the range in the template parameters or as constructor parameters. The former way is preferable because it allows us to think of the range as part of the type itself, not on a per-object basis. This also allows us to be more idiomatic and `typedef` a properly named class from the parameterized template, as in example 4.6, below.

Code Example 4.6

```
template <class Number, Number lowerLimit, Number upperLimit>
class ranged_number {
        public:
                // operators and methods to make the class behave
                // just like an actual number, with the addition of
                // restricting the range.

        private:
                Number number;
};

class BrightnessDecorator {
        public:
                // ...

                Color getColor () const {
                        Color const oldColor = BitmapIterator->getColor ();

                        ColorComponent const red = oldColor.getRedLevel () +
                                brightnessAdjustment;

                        ColorComponent const green = oldColor.getGreenLevel () +
                                brightnessAdjustment;

                        ColorComponent const blue = oldColor.getBlueLevel () +
                                brightnessAdjustment;

                        return Color (red, green, blue);
                }

        private:
                typedef ranged_number <int, 0, 255> ColorComponent;

                // ...
};
```

These examples should clearly show the path of improvement towards terse, expressive, and safe code. The path can always end simply by making all subtle concepts first-class, but this isn't always possible in the language and time constraints. It also can be unrealistic if you can't get the entire world to switch over to your ranged number class or even smaller classes, like a `ColorComponent` class. If a class is not used ubiquitously, but only on your side of the equation, you have to constantly translate between two different views of the system. This can be worthwhile in some cases, but simply a burden in others. It all depends on how much gain you get. If you use the ranged number class to implement the red, green, and blue attributes of the `Color` class, it might not be appropriate to make clients aware of this fact—especially if you anticipate changing the `ranged_number` class to make it better fit your needs over time. If you use it merely in the implementation but accept an integer, you still allow ranged numbers to be passed in, as they will automatically be converted to plain integers.

Of course, this will never be as clean and simple as if you could require all clients to use your `typedef` of ranged numbers directly; but, as we said, you can't always expect your fine-grained first-class concepts. Few clients would want to buy into a framework that made them commit to using your ranged number class in any and all cases of ranged numbers. They might even have their own solution for such problems, or when they combine your framework with several others in creating a product, they'd have a mess of confusion. Making sure you don't always expose your reinventions or perfections of the wheel to the outside helps your framework be conceptually compatible with the rest of the frameworks out there. This doesn't mean you should always take a lowest-common denominator approach, but it does caution you to exercise a bit of pragmatism in what you require from your client.

One final note on the ranged number class: You might think we could just have created an unsigned character. This approach would not have guaranteed our range at all; certain automatic conversion rules might actually translate negative numbers to positive numbers. Storustrup explicitly cautions against using unsigned numbers as a means of restricting ranges, as follows:

> The unsigned integer types are ideas for uses that treat storage as a bit array. Using an unsigned instead of an `int` to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by implicit conversion rules.
> —*The C++ Programming Language*, 3rd edition, page 73

In assignment 4, you will finish the `random_number` class and create several `IBitmapIterator` decorators to achieve brightening and inverting effects. You will then be able to decorate an iterator from the bitmap class (which we will discuss in the next section) before giving it to an encoder.

Creating the Bitmap Class

We still needn't spend too much time on the Bitmap class, especially since we haven't yet fitted it to a domain. Right now our essential concerns revolve around our CODEC system, and our Bitmap itself is not a true graphic so much as a collection. We won't be able to reuse our old Bitmap class completely, but that's okay because the actual loading is now done in the decoder—leaving our new Bitmap class simpler (in terms of implementation). For clarity, we'll want to remove the support for our standard library style

bidirectional iterators from the interface. This shouldn't be a problem since our clients only need read-only access currently.

The only decision we need to make is how to implement copying from the iterator to our scan line collection. Other than that, the BitmapIterator of which we'll have to provide an implementation should pose few problems. In the last lesson, you used the copy standard algorithm to perform this kind of task. But, in our system we have an iterator that works based on polymorphism; for this reason, and due to naming improvements, it does not use the increment, decrement, dereference, and assignment operators. This alone is not what prevents us from using it, though. Additionally, our iterator has a different design philosophy. There is no "end" iterator. Instead, ending is checked for by the `isEndOfScanLine` and `isEndOfImage` methods. Trying to incorporate an "end" iterator instead would have been complex—not only for the implementation but for the client.

It would be possible to create adapters for our iterator to make it work with standard algorithms if we had strong cases for using many standard algorithms, but the essence of our read-only bitmap iterator limits what algorithms we can use on it. Obviously we can't transform, and primarily we'd only be copying. So, if we still want to maintain idiomatic code we can simply name our isolated method that copies appropriately.

The bitmap class will remain simple until we fix it in a domain. In Lesson Five, we will use it for rendering vector graphics scenes, which will only stress the encoding side of things. In Lesson Six, though, we will introduce basic bitmap graphics support into our vector graphics framework, which will stress this class and make the case for a new BitmapGraphic class.