C++ Advanced – Lesson 4

Supporting Encoders and Decoders

Lesson Objective

- Be able to implement CODEC support of the new Bitmap class
- The assignment for this lesson has mostly same inputs and outputs as assignment 3
    - With more sophisticated decoding/encoding

New bitmap features

- Our bitmap will now support
  - Client defined file format for decoding/encoding
  - File extension independent decoder determination
  - Object-oriented iterators for implementation independence and run-time decoration
  - Support for deferred loading and decoding
    - Allows load and decode on first use – handy for remote images and slow formats
  - Flexible iterator-based decoding

Reuse

- Limited reuse from last lesson
  - New design will be more OO and robust
- Compare implementation differences
  - New design more flexible for future changes
  - Gain flexibility without much additional work
  - Easy to add support for other bitmap formats such as GIF, TIFF, and JPG

CODECS in modern systems

- CODEC
  - Comes from Encoder/Decoder
  - Facilitates conversion of information to/from a different format
  - Used with audio, video, file compression and other applications
- May be dynamically added to multimedia applications
  - E.g., RealOne player, Windows Media Player, Winamp
- Some applications automatically find and install codecs from the web

Defining the Decoder

- We’ll separate decoders and encoders
  - Different implementations
  - Different usages
- Both reuse WindowsBitmapHeader class
- For genericity we’ll work with streams
  - Not files – why?
  - E.g., Add future support of HTTP stream

The Decoder

- Will report whether they support a given format
  - Can determine from first data chunk
  - Bitmap decoder only needs first 2 bytes
- Our decoder uses an iterator to get data to client
  - Client will iterate through converted data
    - Accessible via createIterator method
  - Decoding may be deferred until iteration
- MIME type (string identifying format)
  - E.g., “text/html”
  - We’ll use “image/x-ms-bmp”
    - This is the Windows Bitmap MIME type

The Decoder

- The decoder must report whether or not it supports a chunk of data
  - We'll send a 100 byte chunk of incoming data
  - This is a common approach
- The decoder can't be passed a stream for this check
  - Streams lose information as they are read
  - We'll use a string
  - Once the decoder reports OK, we'll give it the stream
    - Use seekg to reset the stream to the beginning

Defining the Encoder

- Similar but inverse of the Decoder
- Encoder will use encodeToStream
    - Instead of Decoder's createIterator
- Encoder created with an iterator and produces a stream
- Decoder created from stream and produces an iterator

Creating the CODEC library

- We’ll use a CodecLibrary singleton
  - Contains a table of encoders/decoders
  - Allows dynamic adding of encoders/decoders
- Library allows us to look up appropriate decoder/encoder by MIME type
  - But, for decoders we’ll usually won’t use MIME type
  - Instead, we’ll use decoder support determination method

CodecLibrary Code Example

```
class CodecLibrary {
public:
    static CodecLibrary& getInstance ();

    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);
};
```

CodecLibrary observations

- createXxx used instead of find or get to emphasis new object creation
- Decoder creation means the CODEC library can be responsible for getting first chunk from file
- Prototype design pattern used to create new instance
  - For decoders:
    - `clone (std::istream& sourceStream)`
  - For encoders:
    - `Clone(const HBitmapIterator& bitmapIterator)`

Creating the Bitmap Iterator

- NOT compatible with STL iterators.
  - Perhaps this is poorly named
  - Is a throwback to older times
- Should be scanline based
- Ideally one class
  - Not separate pixel and scan-line iterator
  - Create interface for iterator
    - Because different encoders provide different implementations
- What's an interface look like in C++?
  - Contains pure virtual methods only
    - Except destructor and constructor
  - Don't forget to make the destructor virtual

BitmapIterator Code

```
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;
};
```

BitmapIterator Observations

- No end() iterator
  - Closer to iterator pattern as described in "Design Patterns"
- Iterator knows width and height
  - Width and height don't change
  - Allows creation of stretching decorator
    - See next section
- In Assignment 4 you will provide various necessary iterator implementations
  - No client will know more than the interface

Creating Bitmap Iterator Decorators

- Apply effects on the fly by decorating an iterator
- For example
  - Brightness effect iterator
  - Invert color iterator
- Can repeat decorations and arbitrarily layer them

Brightness Decorator Code Example 1

```
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 ();
    }
```

Brightness Decorator Code Example 2

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

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

Brightness Decorator Code Example 3

```
 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;
};
```

Decorator Code Comments

- Note error in code
  - `else if (green < 0) { red = 0`
  - Incorrectly adjusts red
- Adjustments of code is complicated and redundent
  - Results in easy to make errors
- Replace if statements with adjustColorComponent method

Improved Brightness Decorator Code

```
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;
        }
    }
};
```

Make the code generic

- Note the general algorithm of last slide
  - Increment or decrement a value while keeping it restricted to a specified range
- Generic algorithms are good at solving this problem generally
- Specifying many parameters on function call can be awkward
  - We'll pass range as template parameters

Generic RangeAdd

```
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);
    }
};
```

Ranged Number solution

- More than one solution
- Create a ranged_number template class
  - Behaves exactly as normal numbers
  - But with automatic range enforcement
    - Std::clamp might be useful
- We have choice
  - Specify range as template parameters
  - Or as constructor parameters
- We'll prefer former as it makes range part of type

ranged_number code example

```
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;

    // ...
};
```

Summary of design/code changes

- Note path towards terse, expressive, safe code
- Path may end by making all subtle concepts first class
  - But we are constrained by time, language, our current understanding, as well as existing design and code
- In assignment 4 you will
  - Finish the ranged_number class
  - Implement several IBitmapIterator Decorators

Creating the Bitmap Class

- Not too much to do with the Bitmap class for assignment 4
  - Header should be reusable directly
- We’ll switch from STL style iterators to IBitmapIterator
- Modify your Bitmap class to work with IBitmapIterator