C++ Advanced

Introduction to Bitmaps

Lesson Objectives

- Learn to design and implement bitmap graphics
- Be able to use the bitmap file format
  - Used by web browers and a wide variety of other software
- Be able to implement these classes:
  - Bitmap
  - Windows Bitmap Header
  - Binary Primitive
    - Byte, Word, DoubleWord

The Windows Bitmap File Format

- There are many bitmap file formats
- We'll study the windows bitmap format
  - Initially developed for windows
  - Now used on many platforms
- Flat file format
- Consists of header and raw data
- Header describes the bitmap
  - Width, height, color depth
  - File info (such as compression information)

Color Depth

- Defines the number colors an image or screen can display
- E.g., 24-bit RGB
  - 3 bytes one each for red, green, and blue
  - 16,777,216 possible colors
  - Resolution similar to human eye
- Other depths (16 bit, 8 bit, 4 bit) more complex
  - Add a color table to map to 24 bit colors
- We'll only support 24 bit

Compression

- Most commonly uses RLE compression
  - Replaces repeated values with value and count
- Windows Bitmap Format supports compression
  - For 8-bit and 4-bit color depths only

Printing Information

- Print options include
  - Horizontal-pixels-per-meter
  - Vertical-pixels-per-meter
  - Used to determine an appropriate printer resolution
- We won't be printing so we'll ignore printing options

Header Format Overview

Name	Size and Values	Description
File Header (14 bytes)		
firstIdentifier	Byte: must be equal to B	Used to determine that this file is a windows bitmap file
secondIdentifier	Byte: must be equal to M	
fileSize	DoubleWord	The actual size of the entire file in bytes
reserved	DoubleWord; must be 0	Unused
rawImageByteOffset	DoubleWord	Specifies the offset from the beginning at which the raw bitmap data begins. This value will always be equal to the size of the entire header plus any color table

Header Format Overview

Name	Size and Values	Description
	Info Header (40 bytes)	
infoHeaderBytes	DoubleWord; must be 40	For verification
bitmapWidth	DoubleWord	
bitmapHeight	DoubleWord	
numberOfPlanes	Word; must be 1	Unused
bitsPerPixel	Word; [1, 4, 8, 16, 24]	Color Depth
compressionType	DoubleWord	Describes raw image data compression
compressedImageSize	DoubleWord; 0 if N/A	The size of the compressed data
horizontalPixelsPerMeter	DoubleWord	Used for printing. Often ignored
verticalPixelsPerMeter	DoubleWord	
numberOfColors	DoubleWord; 0 if N/A	If color table used
numberOfImportantColors	DoubleWord; 0 if all	

Windows Bitmap Format

- After the header is optional color table
  - We won't use
- File Header
  - Minimal data to process – tells us we do have a bitmap

Reading the Header

- Could pass address of struct instance
  - No type checking
  - Potential byte ordering problems
- We'll do something better

Byte Ordering

- Consider the DoubleWord 0xAABBCCDD
- On LittleEndian platform stored as 0xDDCCBBAA
- On BigEndian platform stored as 0xAABBCCDD
- Windows Bitmap Format uses LittleEndian
- We'll need to swap bytes if on BigEndian system

Binary Mode

- Modes of reading data from streams (e.g., a file stream)
- For fstream
  - In text mode end-of-line character is translated to account for platform differences
  - In binary mode automatic translation is not done
  - All other read/write is same
- To simplify reading we'll use Byte, Word, and DoubleWord first class concepts

Creating Byte, Word, and DoubleWord classes

- Make them behave as primitive types
  - C++ standard does not specify exact size of short, int, and long - depends on system too
  - C++11 has the solution: fixed width integer types eg. int8_t, uint8_t
    - Sizes of 8, 16, 32, 64
    - Guaranteed to be exactly N bits wide
    - C++17 added Byte, but this is not a 'math' type, only a collection of 8 bits
  - We'll use unsigned types

Byte, Word, and DoubleWord cont'd

- Need to be able to use them as math types too
  - Will provide conversions to standard types
- Explicit keyword prevents any unwanted automatic (implicit) conversions
  - Forces the programmer to acknowledge that conversions happening or won't compile
- Can be templatized on the properly sized uintN_t type
  - Then using Word = SizedWord<uint16_t>, etc

Reading

- We'll read byte by byte
  - No guarantee that short is only two bytes
  - Just that it's at least 2 bytes
- Implement Byte class
- Use Byte class in Word and DoubleWord implementations
- Best off using iostream get and put methods

Word implementation

```
namespace Binary {
   class Word {
      public:
         Word (unsigned short value = 0)
            : value (value) {
         }
         operator unsigned short () const {
            return value;
         }
         Word& operator= (Word const& original) {
            this->value = original.value;
            return *this;
         }
         Word& operator= (unsigned short value) {
            this->value = value;
            return *this;
         }
      private:
         unsigned short value;
   };
}
```

Reading/Writing

- Implementation will vary by platform
  - Little Endian or Big Endian
- Clients request file Word read
  - File may be Little Endian or Big Endian
  - Client results in platform native format
  - Our code does translation
- Byte class has not to care about ordering
- Word and DoubleWord will handle byte ordering

Application Program Interface (API)

- static Word readLittleEndian(std::istream& sourceStream);
- void writeLittleEndian(std::ostream& destinationStream) const;
- static Word readBigEndian(std::istream& sourceStream);
- void writeBigEndian(std::ostream& destinationStream) const;

API Continued

- Note static and non static methods
- Read methods are class methods (static)
  - They create a Word instance
- Write methods ask a particular instance to persist itself
  - They are instance methods
- In this assignment you will implement Byte, Word, and DoubleWord classes
  - Allow implementation to work Little Endian and Big Endian systems

Example configuration

- Use preprocessor ifdef

```
void Word::writeLittleEndian (std::ostream&
  destinationStream) const {
#ifdef Little_Endian_
    writeNativeOrder (destinationStream);
#else
    writeSwappedOrder (destinationStream);
#endif
}
```

- Define Little_Endian_ flag in build options

WindowsBitMapHeader class

```
class WindowsBitmapHeader {
public:
   // appropriate accessors, constructors, etc. . .

   // . . .
   void writeFileHeader (std::ostream& destinationStream) const {
      firstIdentifier.write (destinationStream);
      secondIdentifier.write (destinationStream);
      fileSize.writeLittleEndian (destinationStream);
      reserved.writeLittleEndian (destinationStream);
      rawImageByteOffset.writeLittleEndian (destinationStream);
   }
```

WindowsBitMapHeader class

```
private:
    // file header
    static Binary::Byte firstIdentifier;
    static Binary::Byte secondIdentifier;
    Binary::DoubleWord fileSize;
    static Binary::DoubleWord reserved;
    static Binary::DoubleWord rawImageByteOffset;

    // info header
    static Binary::DoubleWord infoHeaderBytes;
    Binary::DoubleWord bitmapWidth;
    Binary::DoubleWord bitmapHeight;
    static Binary::Word numberOfPlanes;
    static Binary::Word bitsPerPixel;
    static Binary::DoubleWord compressionType;
    static Binary::DoubleWord compressedImageSize;
    static Binary::DoubleWord horizontalPixelsPerMeter;
    static Binary::DoubleWord verticalPixelsPerMeter;
    static Binary::DoubleWord numberOfColors;
    static Binary::DoubleWord numberOfImportantColors;
};
```

Class notes

- Some members are static
  - These are members whose value does not change between instances (for now)
  - This also documents this design decision
- Note simplicity of writeFileHeader
  - Comes from making binary primitives first-class concepts

WriteFileHeader method

```
void WindowsBitmapHeader::readFileHeader (std::istream& sourceStream) {
    verifyEquality (firstIdentifier, Byte::read (sourceStream), "firstIdentifier");
    verifyEquality (secondIdentifier, Byte::read (sourceStream), "secondIdentifier");
    this->fileSize = DoubleWord::readLittleEndian (sourceStream);
    verifyEquality (reserved, DoubleWord::readLittleEndian (sourceStream), "reserved");
    verifyEquality (rawImageByteOffset, DoubleWord::readLittleEndian (sourceStream),
        "rawImageByteOffset");
}
```

- verifyEquality is template function
  - Will work with Byte, Word, and DoubleWord
- You will need to provide accessors for attributes client requires
- When implementing width/height setters
  - Recalculate size of file

Implementing the Bitmap class

- Represents the bitmap
- Provides loading and iteration
- We'll defer insertions and removals
- Our bitmap concept
  - 1 dimensional array of colors
  - Iteration on array by scan line
- Why not 2 dimensional?
  - Most algs designed around one dimension
  - We can word efficiently 1 scan line/iteration

Our bitmap design

- Our bitmap class will work like a standard collection

```
class Bitmap {
public:
    typedef std::list<Color> ScanLine;

private:
    typedef std::list<ScanLine> ScanLineCollection;

public:
    typedef ScanLineCollection::iterator ScanLineIterator;

    Bitmap (std::istream& sourceStream);

    ScanLineIterator begin (); // named this way for consistency.
    ScanLineIterator end ();

    int getWidth () const;
    int getHeight () const;

    // ...
};
```

Raw Data Format

- Stored as sets of 3 bytes
  - Red, Green, Blue
- Each scan line is DoubleWord aligned
  - Must be divisible by 4
  - All scan lines are thus padded with extra bytes
- For example if the image width is 10
  - Each scan line is 30 bytes (3 x 10)
  - 32 is closest larger number divisible by 4
    - Scan line is thus 32 bytes long or 8 DoubleWords

Iterators

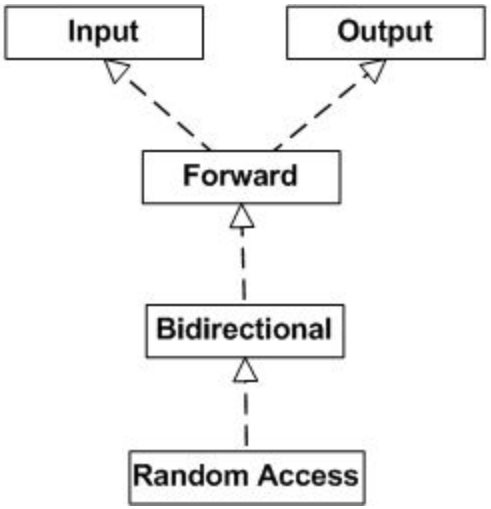
- Our bitmap is effectively no different than other standard library containers
- Generic algorithms are access elements in collections using iterators
- Iterators help us focus on the elements in a collection
- If we add iterators to our collection, we can use std library algorithms with it

Iterators

- No standard iterator base class or interface
- Instead, iterators defined by behavior
  - This is called duck typing
  - Templates provide compile time duck typing
  - Some languages (Ruby, Python) offer run time duck typing

Categories of Iterators

- Constant iterators
  - Used to iterate a collection for read only access
- Iterator Categories:



Input Iterator

- Single increment
- Dereferencing
- Pseudo interface (remember duck typing)

```
interface Input_iterator<Element> {
    input_iterator (input_iterator const&);

    input_iterator& operator++ ();
    input_iterator& operator++ (int);

    Element const& operator* () const;
    Element const* operator-> () const;

    bool operator== (input_terator const&) const;
    bool operator!= (input_iterator const&) const;
}
```

Output iterators

- Single increment
- Assignment
- Modify-only access
- Write values with forward movement

```
interface output_iterator<Element> {
    output_iterator (output_iterator const&);

    output_iterator& operator++ ();
    output_iterator& operator++ (int);

    Element& operator* ();
}
```

Forward Iterators

- Provides input iterator and some forward iterators functionality

```
interface forward_iterator<Element> {
    forward_iterator ();
    forward_iterator (forward_iterator const&);
    forward_iterator& operator= (forward_iterator const&);

    forward_iterator& operator++ ();
    forward_iterator& operator++ (int);

    Element const& operator* () const;
    Element const* operator-> () const;

    bool operator== (forward_iterator const&) const;
    bool operator!= (forward_iterator const&) const;
}
```

Iterators differences

- Forward iterators rely on an end
- Output iterators do not rely on end
- Input and output iterators do not allow multi-pass algorithms

Bidirectional Iterators

- Provide both post and pre-decrement
- Plus all capabilities of forward iterators

```
interface bidirectional_iterator<Element>
: public forward_iterator<Element> {
    // note that if this was real C++, rather than a pseudo
    // language one would have to overload additional operators
    // because they would be returning the supertype, rather than
    // bidirectional_iterator.

    bidirectional_iterator& operator-- ();
    bidirectional_iterator& operator-- (int);
}
```

Random-Access Iterators

- Has ability to jump ahead by more than one element

```
interface random_access_iterator<Element>
: public bidirectional_iterator<Element> {
    Element& operator[] (int index);

    random_access_iterator& operator+= (difference_type increment);
    random_access_iterator& operator-= (difference_type decrement);

    random_access_iterator operator- (difference_type decrement);
    difference_type operator- (random_access_iterator const&);

    bool operator< (random_access_iterator const&);
    bool operator> (random_access_iterator const&);
    bool operator<= (random_access_iterator const&);
    bool operator>= (random_access_iterator const&);

    friend random_access_iterator operator+ (
        difference_type increment, random_access_iterator const&);
}
```

Random Access Iterator features

- Provides indexing which allows iterator to be treated like an array style collection
- Operator+=() and operator-=()
  - Moves forward or backward over more than one element
- Operator+() and operator-()
  - Pointer arithmetic operators
- Comparison operators

Reverse iterators

- An adapter for bi-directional iterator
- Reverses increment and decrement semantics
- Allows standard algorithms to operate on elements forward or backward