EasyBMP User Manual (Version 1.06)

Paul Macklin

email: macklin01@users.sourceforge.net
WWW: http://easybmp.sourceforge.net

May 2, 2017

Abstract

We define and document a simple, easy-to-use, cross-platform/cross-architecture Windows bitmap (BMP) library written in C++. The EasyBMP library will work for input and output on uncompressed 1, 4, 8, 16, 24, and 32 bpp (bits per pixel) Windows BMP files in just about any operating system on just about any 32-bit or higher architecture. EasyBMP has been tested on both little-endian (Pentium 3, Pentium 4, Celeron, Celeron M, Pentium M) and big-endian (Sun Sparc4) machines in Linux, Solaris, and Windows on 32-bit and 64-bit CPU's. Currently, GNU g++ (along with variants and/or frontends such as MinGW and Bloodshed), Intel's compiler, Borland's compiler, and Microsoft's MSVC++ compilers are fully supported.

Contents

1 What's New in this Release (Version 1.06)

Several major changes have been made since Version 1.03, including bugfixes, improved robustness, code cleanups, and feature enhancements.

1.1 Changes in Version 1.04

Version 1.04 focused on improving compatibility with the Borland compiler. Very few warnings are generated when compiling; those that remain can be safely ignored and point to quirks of Borland rather than the code.

This release also added some new functionality: the ability to suppress all EasyBMP warning and error messages. This new feature should be particularly useful for projects that have no need for terminal output.

1.2 Changes in Version 1.05

Version 1.05 improved compatibility by adding a copy constructor for the BMP class. A new feature was also added: bilinear image rescaling to a desired percentage, width, height, or square box fit.

1.3 Changes in Version 1.06

Version 1.06 fixed several minor bugs, particularly in the copy constructor. (Thank you to "redmaya" in South Korea!) The `cctype` and `cstring` includes were added to improve compliance and compatibility. Lastly, `GetPixel()` and `SetPixel()` functions were added for future use, where we hope to be more careful with the `const` keyword.

For further information on the changes made in EasyBMP, please see the EasyBMP_ChangeLog.txt file that is included with every release.

2 Introduction to the EasyBMP Library

During my studies at the University of Minnesota and the University of California, I needed a simple method to create and modify images. Because the Windows BMP file format is nearly universally readable, flexible, and simple, I decided to work with this format. (No compression to worry about, potential for 8 bits per color channel, etc.)

There are many excellent open- and closed-source BMP and image libraries available, and in no way do I claim that anything here is even equal to those libraries. However, as I looked about I noticed that quite a few existing libraries had one or more of the following properties:

- too feature-rich (and accordingly more difficult to learn);
- required extensive installation;
- relied upon Linux or Windows libraries for simple functions;
- were too poorly documented for the novice programmer;
- required programming changes when moving code from one platform to another.

At that point, I decided to create my EasyBMP library. My goals included easy inclusion in C++ projects, ease of use, no dependence upon other libraries (totally self-contained), and cross-platform compatibility.

2.1 Sample Application: Converting a Color Image to Grayscale

Here, we give a first sample application using the EasyBMP library. Notice that inclusion of the library is simple: we include the EasyBMP.h file. In this application, we see a simple example of opening an existing BMP file, reading its RGB information, and manipulating and writing that information to another BMP file. The commands are pretty straightforward. This example should illustrate how easy the library is for even the novice programmer.

```
#include "EasyBMP.h"
using namespace std;

int main( int argc, char* argv[] )
{
 if( argc != 3 )
 {
  cout << "Usage: ColorBMPtoGrayscale <input_filename> <output_filename>"
       << endl << endl;
  return 1;
 }

 // declare and read the bitmap
 BMP Input;
 Input.ReadFromFile( argv[1] );
```

```
 // convert each pixel to grayscale using RGB->YUV
 for( int j=0 ; j < Input.TellHeight() ; j++)
 {
  for( int i=0 ; i < Input.TellWidth() ; i++)
  {
   int Temp = (int) floor( 0.299*Input(i,j)->Red +
                           0.587*Input(i,j)->Green +
                           0.114*Input(i,j)->Blue );
   ebmpBYTE TempBYTE = (ebmpBYTE) Temp;
   Input(i,j)->Red   = TempBYTE;
   Input(i,j)->Green = TempBYTE;
   Input(i,j)->Blue  = TempBYTE;
  }
 }

 // Create a grayscale color table if necessary
 if( Input.TellBitDepth() < 16 )
 { CreateGrayscaleColorTable( Input ); }

 // write the output file
 Input.WriteToFile( argv[2] );

 return 0;
}
```

Additional code samples are available for download at

`http://easybmp.sourceforge.net`

3 What's EasyBMP Good for?

Lots of things! Okay, so we're a little biased. :-) EasyBMP was first used to easily load textures in a homebrew raytracer. Later on, however, its focus shifted to being a simple, intuitive way to get image data in and out of programs, particularly scientific applications. Some sample application ideas include:

1. If you have a scientific computation that runs for hours, days, or weeks, you could add a quick routine that outputs a snapshot of the simulation after every time step. This would provide an easy way to check the status of your long-running simulation without the overhead of starting up Matlab, Tecplot, etc. With X-forwarding, you could even check your simulation snapshot remotely over an SSH shell on free WiFi at Panera! :-)
2. You could write a small program that generates animation frames from your simulation data. Then, remotely start the program over a command line shell, let it run on its own, and collect the results later. Again, no overhead of Tecplot or Matlab, and no user interaction required! (You could then combine the movie frames into an AVI animation using EasyBMPtoAVI. See `http://easybmptoavi.sourceforge.net`.)

3. Import patient data (e.g., MRI imagery) into a patient-tailored simulation or for medical analysis.
4. Create arbitrary starting shapes for level set methods based on your own hand-drawn BMP files. This eliminates the artificial restriction of only being able to simulate shapes whose level set functions that can readily be described with a formula.
5. Interface your science applications with imaging equipment.
6. Use EasyBMP as a quick testbed for new image processing ideas.
7. Import snapshots from a webcam, compare them, and use EasyBMP to remotely detect changes in a room.
8. Import textures for raytracing and OpenGL programs.
9. Save screenshots of OpenGL programs. Or an X program. Or ...
10. Create nice graphics for a system monitoring utility.

4 Installing and Using the EasyBMP Library

Installing the EasyBMP library is easy. Simply copy all the `*.h` and `*.cpp` files to the directory of your project. Alternatively, copy all the header files anywhere in your compiler's path. You should have the following files:

1. `EasyBMP.h`
2. `EasyBMP_DataStructures.h`
3. `EasyBMP_BMP.h`
4. `EasyBMP_VariousBMPutilities.h`
5. `EasyBMP.cpp`

along with a code sample in the `sample` directory.

To use the EasyBMP library, simply include the `EasyBMP.h` file via

```
#include "EasyBMP.h"
```

Note that if you have copied all the EasyBMP source files to your compiler path, you may not need the quotes, but rather brackets:

```
#include <EasyBMP.h>
```

Compile your source code as you normally would along with the single `EasyBMP.cpp` file; you don't have to link to anything. For instance, to compile the code example above with g++, use

```
g++ -o ColorBMPtoGrayscale ColorBMPtoGrayscale.cpp EasyBMP.cpp
```

A sample project with a makefile is included in the `sample` directory. It demonstrates how to compile EasyBMP with a makefile. Please see the project website at `http://easybmp.sourceforge.net` for further compiling instructions, particularly for Microsoft Visual Studio and makefile tutorials.

5 A Few Words on the BMP file format

Any BMP file begins with a *file header*, which contains information on the file size and data location. The file header is followed by an *info header*, which has information on the dimensions and color depth of the file. All this information is stored in the first 54 bytes of the file.

After that, the data is stored. There are two basic storage schemes for BMP files. If the bit depth is 8 or less (256 colors or fewer), the colors are stored in a table of (red,green,blue,alpha) values immediately after the info header. In this *indexed* format, each pixel in the image refers to the position of a color in the color table. In essence, this storage technique is "painting by number." The benefit is that each pixel requires little space, but each of the 256 colors can attain the full range allowed on modern machines. (One of 256^3 colors with 256 values of transparency.)

The other scheme involves storing the red, green, and blue data at every pixel, along with (possibly) the alpha value. This format is very intuitive and allows for the full range of colors for all pixels. However, the resulting files are substantially larger than for 1, 4, and 8 bpp (bits per pixel) files.

An intermediate storage scheme is to allocate two consecutive bytes per every pixel (16 bpp bit depth). In this scheme, 5 bits are allocated to red and blue, and 5 or 6 bits are allocated to green. The green channel is given preference over the other channels because the human eye is most sensitive to green, followed by red and then blue; allocating the extra bit to green makes the most rational use of the space when considering the human viewer.

EasyBMP internally converts all files to 32 bpp for ease and consistency of handling. In particular, this makes it easy to write add-on functions that work on all bit-depth files. EasyBMP converts back to the original bit depth when saving the file.

The coordinate system of a BMP file has its origin in the top left corner of the image. The $(i,j)^{\text{th}}$ pixel is i pixels from the left and j pixels from the top. EasyBMP indexes pixels with the same coordinate system. See Figure 1.

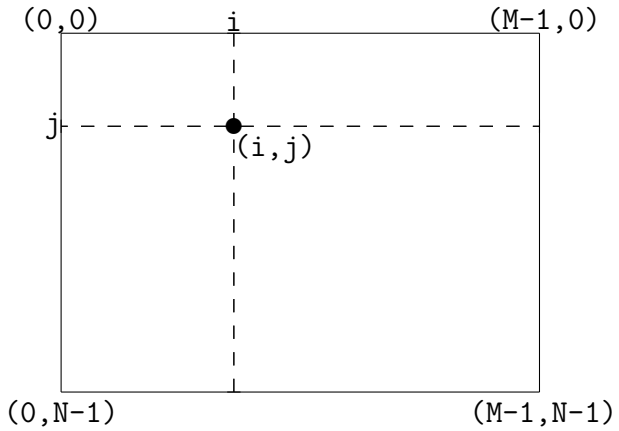

Figure 1: Coordinate system for an `M` × `N` BMP image. The black circle indicates the `(i,j)`$^{\text{th}}$ pixel in the coordinate system.

6 Basic Bitmap Operations

As of Version 0.55, EasyBMP has a unified interface for all bit depths. To initialize a new BMP object, simply declare it:

Example:

```
// Declare a new bitmap object
BMP AnImage;
```

When you declare a BMP image, you will have a 1×1 blank 24 bpp (bits per pixel) bitmap image. Next, set the size and bit depth of the image. You can do this either by reading an existing bitmap image or setting this information manually, as below:

Example:

```
BMP AnImage;
// Set size to 640 × 480
AnImage.SetSize(640,480);
// Set its color depth to 8-bits
AnImage.SetBitDepth(8);
// Declare another BMP image
BMP AnotherImage;
// Read from a file
AnotherImage.ReadFromFile("sample.bmp");
```

To check the bit depth, width, and height of a BMP object, use:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
cout << "File info:" << endl;
cout << AnImage.TellWidth() << " x " << AnImage.TellHeight()
     << " at " << AnImage.TellBitDepth() << " bpp" << endl;
```

EasyBMP also provides a simple routine to compute and display the number of colors:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
cout << "colors:  " << AnImage.TellNumberOfColors() << endl;
```

Note that for a 32 bpp file, we don't regard two colors that differ only in the alpha channel as different colors; this function will state that 32 bpp and 24 bpp files have the same number of colors.

The bit depth and dimensions of a bitmap can be changed at any time:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
// Change the bit-depth
AnImage.SetBitDepth(8);
AnImage.SetBitDepth(24);
// Change the size
AnImage.SetSize(1024,768);
```

Note that whenever the bit depth is changed, any existing color table is erased. Likewise, whenever the size is changed, all pixels are deleted.

To access pixels, use `RGBApixel* operator()(int,int)`:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
// show the color of pixel (14,18)
cout << "(" << (int) AnImage(14,18)->Red   << ","
            << (int) AnImage(14,18)->Green << ","
            << (int) AnImage(14,18)->Blue  << ","
            << (int) AnImage(14,18)->Alpha << ")" << endl;
// Change this pixel to a blue-grayish color
AnImage(14,18)->Red   = 50;
AnImage(14,18)->Green = 50;
AnImage(14,18)->Blue  = 192;
AnImage(14,18)->Alpha = 0;
```

You can also access pixels using `RGBApixel GetPixel(int,int)` and `bool SetPixel(int,int,RGBApixel)`:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
// show the color of pixel (14,18)
RGBApixel Temp = AnImage.GetPixel(14,18);
cout << "(" << (int) Temp.Red   << ","
            << (int) Temp.Green << ","
            << (int) Temp.Blue  << ","
            << (int) Temp.Alpha << ")" << endl;
// Change this pixel to a blue-grayish color
Temp.Red = 50; Temp.Green = 50; Temp.Blue = 192; Temp.Alpha = 0;
AnImage.SetPixel(14,18,Temp);
```

Lastly, to save to a file, use `WriteToFile(char*)`:

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
AnImage.WriteToFile("copied.bmp");
```

7 Advanced Usage: Modifying the Color Table

In EasyBMP, the BMP class will automatically generate a "Windows standard" color table whenever one is required (for 1, 4, and 8 bpp files). However, it is possible to access and modify individual colors in the color table. If you want to see what the n^{th} color in the color table and change another, use the following:

Example:

```
BMP SomeImage;
// Set the bit depth to 8 bpp
SomeImage.SetBitDepth(8);
// Get the 40th color
RGBApixel SomeColor = SomeImage.GetColor(40);
// Display the color
cout << (int) SomeColor.Red   << ","
     << (int) SomeColor.Green << ","
     << (int) SomeColor.Blue  << ","
     << (int) SomeColor.Alpha << endl;
// Change the 14th color to red
RGBApixel NewColor;
NewColor.Red   = 255;
NewColor.Green = 0;
NewColor.Blue  = 0;
NewColor.Alpha = 0;
SomeImage.SetColor(14,NewColor);
```

You can reset the color table to the "Windows standard" color table at any time:

Example:

```
BMP SomeImage;
// Set the bit depth to 8 bpp
SomeImage.SetBitDepth(8);
// Change the 14th color to red
RGBApixel NewColor;
NewColor.Red   = 255;
NewColor.Green = 0;
NewColor.Blue  = 0;
NewColor.Alpha = 0;
SomeImage.SetColor(14,NewColor);
// Reset the color table
SomeImage.CreateStandardColorTable();
```

To maintain previously-provided functionality, EasyBMP still provides a grayscale color table generator:

Example:

```
BMP SomeImage;
// Set the bit depth to 8 bpp
SomeImage.SetBitDepth(8);
// Create a grayscale color table
CreateGrayscaleColorTable( SomeImage );
```

Notice that in the example, the argument is a BMP object.

If you want to modify a color table for a BMP file, you can do so by using the SetColor(int,RGBApixel) member function. To avoid messy error messages, any color table operation should do nothing when applied to a 16, 24, or 32 bpp file. Consider this example:

Example:

```
void CreateRedColorTable( BMP& InputImage )
{
 int BitDepth = InputImage.TellBitDepth();
 if( BitDepth != 1 && BitDepth != 4 && BitDepth != 8 ){ return; }
 int NumberOfColors = IntPow(2,BitDepth); int i;
 ebmpBYTE StepSize;
 if( BitDepth != 1 )
 { StepSize = 255/(NumberOfColors-1); }
 else
 { StepSize = 255; }
 for( i=0 ; i < NumberOfColors ; i++)
 {
  RGBApixel Temp;
  Temp.Red   = i*StepSize;
  Temp.Green = 0;
  Temp.Blue  = 0;
  Temp.Alpha = 0;
  InputImage.SetColor(i,Temp);
 }
}
```

To call this new function, you would do this:

Example:

```
BMP RedImage;
RedImage.ReadFromFile("sample.bmp");
RedImage.SetBitDepth(8);
CreateRedColorTable( RedImage );
```

8 Tools for Horizontal and Vertical Resolution

In some circumstances, it may be useful to specify the horizontal and vertical resolution of the machine where a bitmap image was generated. The BMP provides for this possibility with the `biXPelsPerMeter` and `biYPelsPerMeter` data members, although most files simply set these to 0, indicating that the system default should be used instead.

EasyBMP now supports setting these numbers via the `SetDPI()` function:

Example:

```
BMP SomeImage;
SomeImage.ReadFromFile("sample.bmp");
// Set horizontal resolution to 64 DPI and vertical to 32 DPI
SomeImage.SetDPI(64,32);
```

The default vertical and horizontal resolutions are 96 dpi, and they are indicated by the globals

Example:

```
int DefaultXPelsPerMeter = 3780;
int DefaultYPelsPerMeter = 3780;
```

You can change the default behavior by setting these global variables to your own preferred values. The conversion is pretty simple:

$$\begin{aligned} \texttt{DefaultXPelsPerMeter} &= (\texttt{DesiredDefaultDPI}\text{ dots / inch}) \cdot \left(\frac{1\text{ inches}}{2.54\text{ cm}}\right) \cdot \left(\frac{100\text{ cm}}{1\text{ m}}\right) \\ &= \frac{\texttt{DesiredDefaultDPI}}{0.0254}\text{ dots / meter.} \end{aligned}$$

The `DefaultYPelsPerMeter` can be changed similarly.

9 EasyBMP and Warning Messages

You can turn the error and warning messages on or off in EasyBMP; this is most useful for applications where terminal output is not desired.

Example:

```
// turn the error messages off
SetEasyBMPwarningsOff();

// The following line would ordinarily give an out-of-range warning.
SomeImage( SomeImage.TellWidth() , SomeImage.TellHeight() )->Red = 0;

// turn the messages back off
SetEasyBMPwarningsOn();

// query the current message status
cout << "EasyBMP warning status:  " << GetEasyBMPwarningState() << endl;
```

10 EasyBMP and Metadata

EasyBMP currently supports reading BMP files with metadata by skipping it; in particular, EasyBMP does not preserve or save metadata. In the future, EasyBMP will preserve metadata occurring before and after the pixel data, as well as allow for the writing of additional metadata.

11 Extra Goodies: Various Bitmap Utilities

We have provided several sample utilities to make the library more immediately useful. We shall detail some of these goodies here. :-).

The first several utilities deal with getting file information from existing files.

- `void DisplayBitmapInfo( char* szFileNameIn )`: This routine displays the bitmap information from an existing bitmap. All information is given. (width and height of image, bit depth, etc.)
- `BMFH GetBMFH( char* szFileNameIn )`: This returns a BMFH based on the file. See Section A for more information on the data structure.
- `BMIH GetBMIH( char* szFileNameIn )`: This returns a BMIH based on the file. See Section A for more information on the data structure.
- `int GetBitmapColorDepth( char* szFileNameIn )`: This routine returns the bit depth of the file.

Other provided functions are "cut 'n' paste" functions: they copy pixels from one BMP object to another, with or without transparency.

- ```
  void PixelToPixelCopy( BMP& From, int FromX, int FromY,
                         BMP& To, int ToX, int ToY)
  ```
  This function copies the (FromX,FromY) pixel of the BMP object From to pixel (ToX,ToY) of the BMP object To.
- ```
  void PixelToPixelCopyTransparent( BMP& From, int FromX, int FromY,
                                    BMP& To, int ToX, int ToY,
                                    RGBApixel& Transparent )
  ```
  This function copies the (FromX,FromY) pixel of the BMP object From to pixel (ToX,ToY) of the BMP object To, and it treats the input pixel as transparent if its color is Transparent. Here's an example:

Example:

```
BMP Image1;
BMP Image2;
Image1.ReadFromFile("Blah.bmp");
Image2.SetSize(10,10);
RGBApixel TransparentColor;
TransparentColor.Red = 255;
TransparentColor.Green = 255;
TransparentColor.Blue = 255;
PixelToPixelCopyTransparent(Image1,3,5,Image2,0,0,TransparentColor);
```

Note that the alpha channel is ignored when considering transparency.

- `void RangedPixelToPixelCopy( BMP& From, int FromL , int FromR, int FromB, int FromT, BMP& To, int ToX, int ToY )`

  This function copies a range of pixels from one image to another. It copies the rectangle `[FromL , FromR]` × `[FromB , FromT]` in image `From` to the rectangle whose top left corner is `(ToX , ToY)` in image `To`. When using this function, don't forget that the top left corner of the image is `(0,0)` in the coordinate system! Also, `FromB` denotes the bottom edge of the rectangle, so `FromB` > `FromT`. However, if the algorithm detects that you accidentally reversed these numbers, it will automatically swap them for you. Lastly, if the rectangle you chose to copy from image `From` overlaps the boundary of image `To`, it will truncate the the copy selection, rather than give a nasty segmentation fault. :-)

- ```
  void RangedPixelToPixelCopyTransparent(
       BMP& From, int FromL , int FromR, int FromB, int FromT,
       BMP& To, int ToX, int ToY ,
       RGBApixel& Transparent )
  ```

  This function does the same thing as the previous function, but with support for transparency. As in the example for the pixel-to-pixel copy above, you specify a transparent color of type `RGBApixel`.

To maintain backward compatibility, we provide a function to generate a grayscale color table.

- `bool CreateGrayscaleColorTable( BMP& InputImage );` This function creates a grayscale color table for any 1, 4, or 8 bpp `BMP` object. It returns `true` when successful and `false` otherwise. An example is given in Section 7.

The last function rescales an image:

- `bool Rescale( BMP& InputImage, char mode, int NewDimension );`

  There are several different modes of operation available, which are probably best explained by example. All scaling is done by bilinear interpolation. Note that the `mode` input is not case-sensitive.

Example:

```
BMP AnImage;
AnImage.ReadFromFile("sample.bmp");
// rescale to 42% of original size
Rescale( AnImage , 'p', 42 );

// rescale (preserving aspect ratio) to width 100
Rescale( AnImage , 'W', 100 );

// rescale (preserving aspect ratio) to height 100
Rescale( AnImage , 'H', 100 );

// rescale (preserving aspect ratio) to fit in a 100 x 100 box
Rescale( AnImage , 'f', 100 );
```
```

12 Hidden Helper Functions

EasyBMP has a few helper functions that are intended primarily for internal use, but may also be helpful for your code.

- `double Square( double number )`: This routine gives the square of the double `number`. It is much faster than using the standard library `pow(double,2.0)`.
- `int IntSquare( int number )`: This does the same as the above, but with faster integer operations.
- `int IntPow( int base, int exponent )`: Instead of calling `pow(x,n)` where `x` and `n` are integers, use this function. It's much faster. Beware against using negative exponents, as they aren't supported.

13 Known Bugs and Quirks

As of Version 1.06, there is one known quirk: there is no `operator=` operator defined for the `BMP` class. This means that you can't do code operations like this:

Example:

```
BMP Sample1;
Sample1.ReadFromFile("sample1.bmp");
BMP Sample2;
Sample2.ReadFromFile("sample2.bmp");
Sample1 = Sample2;
```

AFAIK, the biggest implication of this is that you won't want to make functions that return a `BMP` object. Instead, if you want to modify a `BMP` object in a function, you should pass it by reference to the function. An example of this can be seen in the `CreateRedColorTable(BMP&)` sample function given in an earlier section.

`operator=` functionality may be added to EasyBMP at a later date, but only if they do not interfere with stability and robustness.

The other "quirk" in EasyBMP is that writing 8 bpp images is very slow. This is a consequence of the fact that 8 bpp images can be operated on just like 24 bpp images, and so the write routine must search for the best fit color for each pixel when saving. This may be improved in future versions of EasyBMP.

14 Future Changes

Future changes may include:

1. an optimal color table `GenerateOptimalColorTable()` generator function;
2. eliminating the use of ceil and floor functions, as well conversions from double to int wherever possible.
3. adding support for metadata before and after the pixel data
4. improving file write speeds for 8 bpp files.

15 Obtaining Support for EasyBMP or Contacting EasyBMP

If you need support, have a feature request, or have other feedback on EasyBMP, please open a new support tracker item at `http://easybmp.sourceforge.net`. (See the "support" link on that page.) If you also desire email support, you can email Paul Macklin at `macklin01@users.sourceforge.net` or indicate your email address in your tracker item. However, please still open a tracker item in such a case.

Lastly, Paul would love to hear back from people who have successfully used EasyBMP in their own projects.

A Classes and BMP Data Types

Here, we detail the various classes and data types and how to interface with them.

A.1 Miscellany

Some of the data types that are used in the construction of more complex data types are:

Type:	Info:
`ebmpBYTE`	an unsigned character of 8 bits
`ebmpWORD`	an unsigned short of 16 bits
`ebmpDWORD`	an unsigned long of 32 bits
`BMFH`	a specific header format for a BMP file
`BMIH`	provides additional information on the BMP file

For additional information on the `BMFH` and `BMIH` classes, I highly recommend that you visit

`http://www.fortunecity.com/skyscraper/windows/364/bmpffrmt.html`.

A.2 RGBApixel

This data structure is exactly as their its suggests: a single pixel of (red,green,blue,alpha) data. This data structure is used both for individual pixels within an image and the color table in the palette. Here are the details on the data structure:

Member:	Function:
`Blue`	blue pixel info of type `BYTE`
`Green`	green pixel info of type `BYTE`
`Red`	red pixel info of type `BYTE`
`Alpha`	alpha pixel info of type `BYTE`

A.3 BMP

The `BMP` class consists of all the necessary pixel information for a Windows bitmap file, along with file I/O routines.

- `int BitDepth`: This gives the number of bits per pixel, i.e., the color depth. This data member is private and can only be accessed through `TellBitDepth` and `SetBitDepth`.

- `int Width`: This gives the width of the bitmap in pixels. This data member is private and can only be accessed through `TellWidth` and `SetSize`.
- `int Height`: This gives the height of the bitmap in pixels. This data member is private and can only be accessed through `TellHeight` and `SetSize`.
- `RGBApixel** Pixels`: This is the actual `Width` × `Height` array of `RGBApixel`'s. This data member is private and can only be accessed through `operator()`.
- `RGBApixel* Colors`: This is the table of colors, stored as `RGBApixel`'s. If the `BMP` object is 24-bits or 32-bits, then `Colors = NULL`. This data member is private and can only be accessed through `GetColor` and `SetColor`.
- `int XPelsPerMeter:` This records the horizontal resolution in pixels per meter. This private data member can only be accessed through `SetDPI()`. This value defaults to 3780, i.e., 96 dpi.
- `int YPelsPerMeter:` This records the vertical resolution in pixels per meter. This private data member can only be accessed through `SetDPI()`. This value defaults to 3780, i.e., 96 dpi.
- `BYTE* MetaData1:` This is a placeholder for future metadata before the pixel data. This data member is private. This is currently a placeholder.
- `int SizeOfMetaData1:` This private data member gives the size of `MetaData1` in bytes. This is currently a placeholder.
- `BYTE* MetaData2:` This is a placeholder for future metadata after the pixel data. This data member is private. This is currently a placeholder.
- `int SizeOfMetaData2:` This private data member gives the size of `MetaData2` in bytes. This is currently a placeholder.
- `void SetDPI( int HorizontalDPI, int VerticalDPI ):` This function sets the private data members `XPelsPerMeter` and `YPelsPerMeter` such that the image has horizontal and vertical resolutions `HorizontalDPI` and `VerticalDPI` dots per inch, respectively.
- `int TellBitDepth( void ):` This function outputs the bit depth of the `BMP` object.
- `int TellWidth( void ):` This function outputs the width of the `BMP` object.
- `int TellHeight( void ):` This function outputs the height of the `BMP` object.
- `int TellNumberOfColors( void ):` This function outputs the number of colors of the `BMP` object.
- `BMP():` This constructor creates a 1 × 1, 24 bpp `BMP` object.
- `BMP( BMP& Input ):` This the copy constructor, which is used when `BMP` objects are passed by value, rather than reference.
- `~BMP():` This is the destructor. You should never call this; it is automatically called when a `BMP` object goes out of scope.
- `RGBApixel GetPixel( int i, int j ) const:` This is used to get the contents of the `(i,j)` pixel while respecting `const`, using a syntax similar to the color table functions below. Primarily for future use.
- `bool SetPixel( int i, int j, RGBApixel NewPixel ):` This is used to set the contents of the `(i,j)` pixel in a syntax similar to the color table functions below. Primarily for future use.
- `RGBApixel* operator()(int i, int j):` This returns a pointer to the `(i,j)` pixel.

Example:

```
BMP Sample;
Sample.SetSize(10,10);
Sample(3,4)->Red = 255;
Sample(3,4)->Alpha = 0;
Sample(3,4)->Blue = Sample(3,4)->Red;
```

If `operator()` is used on a pixel that is out of range, it warns the user and automatically truncates the range.

- `bool SetSize( int NewWidth, int NewHeight )`: Use this to change the size of the object to `NewWidth` × `NewHeight`. See the example above. If `NewWidth` or `NewHeight` is non-positive, this function gives a warning and does nothing. Returns true or false to indicate success or failure.
- `bool SetBitDepth( int NewDepth )`: This function changes the bit depth to `NewDepth` bits per pixel. It also automatically creates and/or resizes the color table, if necessary. If `NewDepth` is not one of 1, 4, 8, 24, or 32, the function gives an error and does nothing. Returns true or false to indicate success or failure.
- `bool WriteToFile( char* FileName )`: This function writes the current `BMP` object to the file `FileName`. Returns true or false to indicate success or failure.
- `bool ReadFromFile( char* FileName )`: This function reads the file `FileName` into the current `BMP` object. Returns true or false to indicate success or failure.
- `RGBApixel GetColor( int ColorNumber )`: This function returns color number `ColorNumber` in the color table of the current `BMP` object. If `ColorNumber` is out of range or the `BMP` does not have a color table (e.g., in the case of a 24-bit file), this function gives a warning and returns a black pixel.
- `bool SetColor( int ColorNumber , RGBApixel NewColor)`: This function sets color number `ColorNumber` in the color table of the current `BMP` object to `NewColor`. If `ColorNumber` is out of range or the `BMP` does not have a color table (e.g., in the case of a 24-bit file), this function gives a warning and does nothing. Returns true or false to indicate success or failure.
- `bool CreateStandardColorTable( void )`: This function creates a "Windows standard" color table for the `BMP` object. If the bit depth is 24 or 32, the function gives a warning and does nothing. Returns true or false to indicate success or failure.