Multi-Dimension Image Processing Library

Generated by Doxygen 1.10.0

Chapter 1

Multi-Dimension Image Processing Library

This is an implementation for the **Advanced Programming Group Project**.

This project provides a comprehensive toolkit for 2D/3D image processing, including utilities for loading, manipulating, and saving both 2D and 3D image data. It is designed to facilitate advanced image processing tasks such as filtering, projection, slicing of volumetric data, and edge detection. This library is particularly suited for applications in medical imaging, scientific visualization, 3D data analysis, and real-time image enhancement.

For more details, please check:

- Webpage document: `Our Website` or `./Docs/html/index.html`
- User manual: `./Docs/Manual_Multi-Dimension_Image_Processing_Library.pdf`

1.1 Features

- **Image Loading and Saving**: Supports loading and saving images in various formats through the integration of the `stb_image` and `stb_image_write` libraries.
- **Volume Manipulation**: Allows the construction of a 3D volume from a series of 2D slices, along with the capability to save slices or projections of the volume back to disk.
- **Filtering**: Implements median, Gaussian, edge detection, and pixel-level filtering for noise reduction, smoothing, and edge enhancement of volumetric data.
- **Projections**: Offers functions for maximum, minimum, average, and median intensity projections, which are essential for visualizing structural information in volumetric data.
- **Slicing**: Provides functionality to extract arbitrary slices from a volume, facilitating the examination of cross-sectional data.
- **Edge Detection**: Incorporates edge detection algorithms such as Sobel and Prewitt, enabling the identification of edges within images for analysis and processing.

1.2 Project Structure

```
plaintextCopy code
<Multi-Dimension Image Processing Library>/
 CMakeLists.txt               - Project build script.
 README.md                    - Project documentation.
 Docs/                        - Documentation files.
 Images/                      - Sample images for testing.
 Include/                     - Header files.
    Algorithm.h
    Box2DFilter.h
    EdgeFilter.h
    Filters/                 - Filter headers.
    Gaussian2DFilter.h
    Gaussian3DFilter.h
    Image.h
    Median2DFilter.h
    Median3DFilter.h
    Padding.h
    PixelFilter.h
    Projection.h
    Slice.h
    Volume.h
    stb_image.h
    stb_image_write.h
 Output/                      - Output directory for processed images.
 Scans/                       - Input directory for volume scans.
 Source/                      - Source files.
    Algorithm.cpp
    Box2DFilter.cpp
    EdgeFilter.cpp
    Filters/                 - Filter source files.
    Gaussian2DFilter.cpp
    Gaussian3DFilter.cpp
    Image.cpp
    Median2DFilter.cpp
    Median3DFilter.cpp
    Padding.cpp
    PixelFilter.cpp
    Projection.cpp
    Slice.cpp
    Volume.cpp
    main.cpp
 Tests/                       - Unit tests.
```

1.3 Getting Started

1.3.1 3D Datasets

Download CT Scan datasets here:

https://imperiallondon-my.sharepoint.com/:u:/g/personal/tmd02_ic_ac_uk/EafXMuNsbcNGnRpa8K62FjkBvIKvCswl1riz7hPDHpHdSQ

1.3.2 Prerequisites

Ensure you have a modern C++ compiler that supports C++17 and the CMake build system installed on your system. This project also requires the `filesystem` library for directory operations.

1.3.3 Building the Project

We have scripts to automatically run the build commands, do the unit tests and start the command line user interface.

If you prefer using CMake and want to build the project manually, Please follow:

1. Clone the repository to your local machine.
2. Navigate to the project directory and create a build directory:
   ```
   mkdir build && cd build
   ```
3. Run CMake to configure the project:
   ```
   cmake ..
   ```
4. Compile the project:
   ```
   cmake --build .
   ```

The CMakeList files have turned on the compilation optimization by default.

1.4 Usage

We prefer users to operate through the interface we've provided. Please follow the instructions below to build and run the project.

Run Unit Tests

Make sure 3D datases are downloaded and put to `./Scans`.

```
./RunMain.sh
```

Run Main User Interface

```
./RunTest.sh
```

To use the library in your project, include the necessary header files from the `Include` directory and link against the compiled library.

Here's an example demonstrating how to load a image / volume from disk, apply various filters, and save a filtered file / maximum intensity projection:

```
#include "Volume.h"
#include "Image.h"
#include "Filters/Gaussian3DFilter.h"
#include "Filters/Gaussian2DFilter.h"
#include "Projection.h"
#include "Filters/EdgeFilter.h"
#include "Filters/PixelFilter.h"

int main() {
    // Demo for 2D images
    Image image;
    image.loadFromFile("/path/to/your/image");

    // Apply Greyscale filter to image
    PixelFilter pixelFilter("Grayscale");
    pixelFilter.apply(image);

    // Apply a 2D Gaussian filter with sigma = 2.0 and kernel size = 5
        Gaussian2DFilter gaussianFilter2D(5, 2.0);
    gaussianFilter2D.apply(image);

    // Apply Sobel edge filter
    EdgeFilter edgeFilter(FilterType::Sobel);
    edgeFilter.apply(image);

    // Save filtered image
    image.saveToFile("/path/and/name/of/your/image.png");


    // Demo for 3D images
    Volume volume;
    volume.loadFromDirectory("/path/to/your/images_directory");

    // Apply a 3D Gaussian filter with sigma = 2.0 and kernel size = 5
    Gaussian3DFilter gaussianFilter3D(2.0, 5);
    gaussianFilter3D.apply(volume);

    // Save a maximum intensity projection to disk
        volume.save("/path/to/your/directory", "x-y", "MIP");

    return 0;
}
```

1.5 Contributing

Contributions to enhance the functionality or performance of this library are welcome. Please follow the standard fork-branch-PR workflow.

1.6 Developers

This project is developed by:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

The group name is **Advanced Programming Group Radix Sort**.

1.7 License

This project is open source and available under the MIT License. See the LICENSE file for more details.

1.8 Acknowledgments

- Special thanks to Sean Barrett for the `stb_image` and `stb_image_write` libraries, which are used for image I/O operations in this project.
- Appreciation to all contributors and the Advanced Programming Group Radix Sort for their efforts and collaboration in building this comprehensive image processing toolkit.
- Thanks to generative AI tools like ChatGPT and Google Gemini for the help in coding.

Chapter 2

README

List of output images to upload to Output directory on GitHub

```
|-- 1-grayscale
|   |-- gracehopper.png
|   `-- tienshan.png
|-- 2-brightness
|   |-- gracehopper_minus100.png   # minus 100 from each channel
|   |-- gracehopper_plus100.png    # plus 100 to each channel
|   |-- stinkbug_minus50.png       # minus 50 from each channel
|   `-- stinkbug_plus50.png        # plus 50 to each channel
|-- 3-histogram
|   |-- vh_anatomy_HSL.png         # equalise histogram of L channel
|   |-- vh_anatomy_HSV.png         # equalise histogram of V channel
|   `-- vh_ct.png                  # equalise histogram of grayscale image
|-- 4-threshold
|   |-- stinkbug_80.png            # threshold grayscale at 80
|   |-- tienshan_HSL_127.png       # threshold L channel at 127
|   |-- tienshan_HSV_127.png       # threshold V channel at 127
|   `-- vh_ct_80.png               # threshold grayscale at 80
|-- 5-saltandpepper
|   |-- gracehopper_10.png         # 10% salt and pepper noise
|   |-- gracehopper_25.png         # 25% salt and pepper noise
|   |-- stinkbug_10.png            # 10% salt and pepper noise
|   `-- stinkbug_40.png            # 40% salt and pepper noise
|-- 6-blur
|   |-- box
|   |   |-- stinkbug_3x3.png       # 3x3 box filter
|   |   |-- stinkbug_5x5.png       # 5x5 box filter
|   |   |-- tienshan_3x3.png
|   |   |-- tienshan_5x5.png
|   |   |-- vh_anatomy_sp15_3x3.png
|   |   `-- vh_anatomy_sp15_5x5.png
|   |-- gaussian
|   |   |-- stinkbug_3x3.png       # 3x3 gaussian filter
|   |   |-- stinkbug_5x5.png       # 5x5 gaussian filter
|   |   |-- tienshan_3x3.png
|   |   |-- tienshan_5x5.png
|   |   |-- vh_anatomy_sp15_3x3.png
|   |   `-- vh_anatomy_sp15_5x5.png
|   `-- median
|       |-- stinkbug_3x3.png       # 3x3 median filter
|       |-- stinkbug_5x5.png       # 5x5 median filter
|       |-- tienshan_3x3.png
|       |-- tienshan_5x5.png
|       |-- vh_anatomy_sp15_3x3.png
|       `-- vh_anatomy_sp15_5x5.png
|-- 7-edgedetection
|   |-- prewitt
|   |   |-- dimorphos.png
|   |   |-- gracehopper.png
|   |   |-- stinkbug.png
|   |   |-- tienshan.png
|   |   |-- vh_anatomy.png
|   |   `-- vh_ct.png
|   |-- robertscross
|   |   |-- dimorphos.png
|   |   |-- gracehopper.png
|   |   |-- stinkbug.png
|   |   |-- tienshan.png
|   |   |-- vh_anatomy.png
|   |   `-- vh_ct.png
|   |-- scharr
|   |   |-- dimorphos.png
|   |   |-- gracehopper.png
|   |   |-- stinkbug.png
|   |   |-- tienshan.png
|   |   |-- vh_anatomy.png
|   |   `-- vh_ct.png
|   `-- sobel
|       |-- dimorphos.png
|       |-- gracehopper.png
|       |-- stinkbug.png
|       |-- tienshan.png
|       |-- vh_anatomy.png
|       `-- vh_ct.png
`-- 8-3D
    |-- confuciusornis
    |   |-- aip-gaussian_3x3x3.png              # average intensity projection, 3x3x3 gaussian filter
    |   |-- aip-gaussian_5x5x5.png              # average intensity projection, 5x5x5 gaussian filter
    |   |-- aip-median_3x3x3.png                # average intensity projection, 3x3x3 median filter
    |   |-- aip-median_5x5x5.png                # average intensity projection, 5x5x5 median filter
    |   |-- aip-nofilter.png                    # average intensity projection, no filter
    |   |-- mip-gaussian-3x3x3.png              # maximum intensity projection, 3x3x3 gaussian filter
    |   |-- mip-gaussian-5x5x5.png              # maximum intensity projection, 5x5x5 gaussian filter
    |   |-- mip-median-3x3x3.png                # maximum intensity projection, 3x3x3 median filter
    |   |-- mip-median-5x5x5.png                # maximum intensity projection, 5x5x5 median filter
    |   |-- mip-nofilter.png                    # maximum intensity projection, no filter
    |   |-- slice_xz_y420.png                   # slice at y=420
    |   |-- slice_yz_x400.png                   # slice at x=400
    |   |-- thinslab_10_70_nofilter_aip.png     # thin slab between index 10-70, average intensity
      projection
    |   `-- thinslab_10_70_nofilter_mip.png     # thin slab between index 10-70, maximum intensity
      projection
    `-- fracture
        |-- minip-gaussian_3x3x3.png            # minimum intensity projection, 3x3x3 gaussian filter
        |-- minip-gaussian_5x5x5.png            # minimum intensity projection, 5x5x5 gaussian filter
        |-- minip-median_3x3x3.png              # minimum intensity projection, 3x3x3 median filter
        |-- minip-median_5x5x5.png              # minimum intensity projection, 5x5x5 median filter
        |-- minip-nofilter.png                  # minimum intensity projection, no filter
        |-- mip-gaussian-3x3x3.png              # maximum intensity projection, 3x3x3 gaussian filter
        |-- mip-gaussian-5x5x5.png              # maximum intensity projection, 5x5x5 gaussian filter
        |-- mip-median-3x3x3.png                # maximum intensity projection, 3x3x3 median filter
        |-- mip-median-5x5x5.png                # maximum intensity projection, 5x5x5 median filter
        |-- mip-nofilter.png                    # maximum intensity projection, no filter
        |-- slice_xz_y138.png                   # slice at y=138
        |-- slice_yz_x275.png                   # slice at x=275
        |-- thinslab_276_476_nofilter_aip.png   # thin slab between index 276-476, average intensity
      projection
        `-- thinslab_276_476_nofilter_mip.png   # thin slab between index 276-476, maximum intensity
      projection
```

Chapter 3

REFERENCE

3.1 Websites

- https://en.wikipedia.org/wiki/HSL_and_HSV
- https://en.wikibooks.org/wiki/Color_Models:_RGB,_HSV,_HSL
- https://opencv.org/
- https://www.doxygen.nl/

3.2 AI Tools

- https://chat.openai.com/share/ffc0b1bb-e37c-48a7-87ad-39be46f41aaa
- https://chat.openai.com/share/e10d1dda-fcf9-4a2e-b039-a6935679ef62
- https://chat.openai.com/share/c512e3a5-c0d8-406f-a0bf-0e507a91fa22
- https://github.com/features/copilot

Chapter 4

README

CT scans should be unzipped locally in here into two directories: confuciusornis and fracture. Do not add the images to your repository, as they are too large.

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 7

File Index

7.1 File List

Here is a list of all files with brief descriptions:

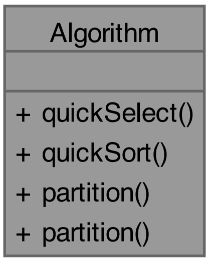
Chapter 8

Class Documentation

8.1 Algorithm Class Reference

```
#include <Algorithm.h>
```

Collaboration diagram for Algorithm:



Static Public Member Functions

- static unsigned char quickSelect (std::vector< unsigned char > &arr, int left, int right, int k)
- static void quickSort (std::vector< std::string > &arr, int low, int high)
- static int partition (std::vector< unsigned char > &arr, int left, int right, int pivotIndex)
- static int partition (std::vector< std::string > &arr, int low, int high)

8.1.1 Member Function Documentation

8.1.1.1 partition() [1/2]

```
int Algorithm::partition (
            std::vector< std::string > & arr,
            int low,
            int high )  [static]
```

Partitions the array of strings around the last element as pivot.

Rearranges the elements in the string vector such that all elements less than the pivot come before it and all elements greater come after it, with the pivot itself moving to its correct sorted position. Primarily used in the quicksort algorithm.

Parameters

arr	A reference to the vector of strings to be partitioned.
low	The start index of the subarray to be partitioned.
high	The end index of the subarray to be partitioned.

Returns

: The index where the pivot element is finally placed.

8.1.1.2 partition() [2/2]

```
int Algorithm::partition (
             std::vector< unsigned char > & arr,
             int left,
             int right,
             int pivotIndex )  [static]
```

Partitions the array around a pivot element.

This method rearranges the elements in the array such that all elements less than the pivot value come before the pivot, while all elements greater than the pivot come after it. The pivot element itself is moved to its final position in the sorted array. This partitioning is a key process in both quicksort and quickselect algorithms.

Parameters

arr	A reference to the vector of unsigned char to be partitioned.
left	The start index of the subarray to be partitioned.
right	The end index of the subarray to be partitioned.
pivotIndex	The index of the pivot element.

Returns

: The index where the pivot element is finally placed.

8.1.1.3 quickSelect()

```
unsigned char Algorithm::quickSelect (
             std::vector< unsigned char > & arr,
             int left,
             int right,
             int k )  [static]
```

Finds the k-th smallest element in an array using the Quickselect algorithm.

Quickselect is a selection algorithm to find the k-th smallest element in an unsorted array. It is related to the quicksort sorting algorithm. Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect uses a partition method to recursively divide the array.

Parameters

arr	A reference to the vector of unsigned char to be processed.
left	The starting index of the array from which to find the k-th smallest element.
right	The ending index of the array.
k	The order of the smallest element to find.

Returns

: The k-th smallest element in the specified array segment.

8.1.1.4 quickSort()

```
void Algorithm::quickSort (
            std::vector< std::string > & arr,
            int low,
            int high )  [static]
```

Sorts a vector of strings using the Quicksort algorithm.

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. This implementation uses the last element as the pivot.

Parameters

arr	A reference to the vector of strings to be sorted.
low	The starting index of the segment of the array to be sorted.
high	The ending index of the segment of the array to be sorted.

Returns

: None.

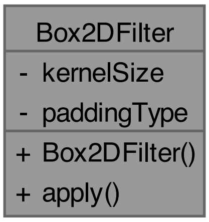
The documentation for this class was generated from the following files:

- Include/Algorithm.h
- Source/Algorithm.cpp

8.2 Box2DFilter Class Reference

```
#include <Box2DFilter.h>
```

Collaboration diagram for Box2DFilter:



Public Member Functions

- Box2DFilter (int kernelSize, PaddingType paddingType=PaddingType::ZeroPadding)
- void apply (Image &image) const

Private Attributes

- int kernelSize
- PaddingType paddingType

8.2.1 Constructor & Destructor Documentation

8.2.1.1 Box2DFilter()

```
Box2DFilter::Box2DFilter (
            int kernelSize,
            PaddingType paddingType = PaddingType::ZeroPadding )
```

Constructor for the Box2DFilter class.

Initializes a Box2DFilter object with a specified kernel size and padding type. The constructor checks that the kernel size is an odd number, ensuring a symmetric area around each pixel for the filter application. This symmetry is critical for the uniform application of the filter across the image. If the kernel size is not odd, an exception is thrown, as the filter operation requires a central pixel. The padding type determines how the filter handles pixels at the image boundaries, allowing for flexible handling of edge cases.

Parameters

kernelSize	The size of the kernel, which must be an odd number to ensure symmetry around each pixel.
paddingType	The padding strategy to use when the filter kernel overlaps the image boundaries.

Exceptions

std::invalid_argument	if the kernelSize is not an odd number, ensuring proper filter application.

8.2.2 Member Function Documentation

8.2.2.1 apply()

```
void Box2DFilter::apply (
             Image & image ) const
```

Applies a 2D box filter to an image.

This method implements the spatial averaging of pixel values across a specified neighborhood around each pixel in the image, based on the kernel size and padding type provided during the object's construction. The process involves iterating over each pixel in the image, collecting pixel values from its neighborhood as defined by the kernel size, and then calculating the average of these values. The result is a blurred or smoothed version of the original image, with the degree of blurring dependent on the size of the kernel. Edge pixels are handled according to the specified padding strategy, which determines how pixels outside the image boundaries are treated for the purposes of the filter.

Parameters

image	A reference to an Image object representing the image to be processed. The Image object must be initialized and loaded with data prior to calling this method. The method modifies the Image object in place, replacing its pixel data with the filtered results.

8.2.3 Member Data Documentation

8.2.3.1 kernelSize

```
int Box2DFilter::kernelSize  [private]
```

8.2.3.2 paddingType

```
PaddingType Box2DFilter::paddingType  [private]
```

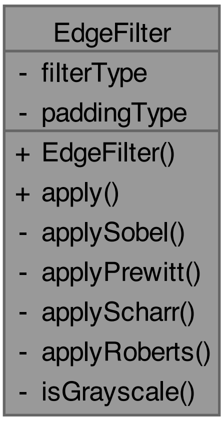
The documentation for this class was generated from the following files:

- Include/Filters/Box2DFilter.h
- Source/Filters/Box2DFilter.cpp

8.3 EdgeFilter Class Reference

```
#include <EdgeFilter.h>
```

Collaboration diagram for EdgeFilter:



Public Member Functions

- EdgeFilter (FilterType type, PaddingType paddingType=PaddingType::ZeroPadding)
- void apply (Image &image)

Private Member Functions

- void applySobel (Image &image) const
- void applyPrewitt (Image &image) const
- void applyScharr (Image &image) const
- void applyRoberts (Image &image) const
- bool isGrayscale (const Image &image) const

Private Attributes

- FilterType filterType
- PaddingType paddingType

8.3.1 Constructor & Destructor Documentation

8.3.1.1 EdgeFilter()

```
EdgeFilter::EdgeFilter (
            FilterType type,
            PaddingType paddingType = PaddingType::ZeroPadding )
```

Constructs an EdgeFilter object with the specified filter type and padding type.

This constructor initializes an EdgeFilter object with the specified edge detection filter type and padding type. The filter type determines the edge detection algorithm to be applied, while the padding type specifies the padding strategy to handle edge pixels during convolution. The constructed EdgeFilter object can be used to apply edge detection to grayscale images.

Parameters

type	The type of edge detection filter to be used (Sobel, Prewitt, Scharr, or Roberts).
paddingType	The padding strategy to be applied during convolution (ZeroPadding, ReplicatePadding, or ReflectPadding).

8.3.2 Member Function Documentation

8.3.2.1 apply()

```
void EdgeFilter::apply (
             Image & image )
```

Applies the configured edge detection filter to an image.

This method applies the selected edge detection algorithm to the provided Image object if the image is in grayscale format. It first checks if the image is suitable for edge detection by calling isGrayscale. If the image is not in grayscale, an error message is printed, and the function returns without modifying the image. Otherwise, it proceeds to apply the chosen edge detection algorithm, updating the image with the detected edges.

Parameters

image	A reference to an Image object on which the edge detection filter will be applied. The Image object must be initialized and loaded with image data prior to calling this method.

8.3.2.2 applyPrewitt()

```
void EdgeFilter::applyPrewitt (
             Image & image ) const  [private]
```

Applies the Prewitt edge detection algorithm to an image.

This method utilizes the Prewitt operator to detect edges in the provided grayscale image. Similar to the Sobel filter, it employs horizontal and vertical kernels to compute the gradient magnitude at each pixel, effectively highlighting the edges. The Prewitt operator is known for its simplicity and effectiveness in edge detection, making it suitable for various image processing tasks where edge delineation is required.

Parameters

image	A constant reference to an Image object to which the Prewitt filter will be applied. The image should be in grayscale format for the algorithm to function correctly.

8.3.2.3 applyRoberts()

```
void EdgeFilter::applyRoberts (
             Image & image ) const  [private]
```

Applies the Roberts Cross edge detection algorithm to an image.

The Roberts Cross operator is one of the earliest yet effective methods for edge detection, focusing on highlighting high-frequency components in the image. It computes the gradient magnitude using a pair of 2x2 convolution kernels. This simplicity makes it particularly fast and suitable for detecting edges in images with minimal computational resources. Despite its age, it remains effective for applications where quick edge detection is more critical than precision.

Parameters

image	A constant reference to an Image object to which the Roberts Cross filter will be applied. The algorithm expects the image to be in grayscale format for accurate edge detection.

8.3.2.4 applyScharr()

```
void EdgeFilter::applyScharr (
             Image & image ) const  [private]
```

Applies the Scharr edge detection algorithm to an image.

The Scharr method is an enhancement over the Sobel operator, offering better rotation symmetry and a higher weight to pixels directly adjacent to the center pixel. This method applies Scharr's specific kernels to calculate the gradient magnitude at each pixel of the provided grayscale image, aiming to accurately detect edges with a higher sensitivity to subtle changes in intensity, thereby producing a more defined edge map.

Parameters

image	A constant reference to an Image object to which the Scharr filter will be applied. It is essential that the image is in grayscale format to ensure the algorithm performs as expected.

8.3.2.5 applySobel()

```
void EdgeFilter::applySobel (
             Image & image ) const  [private]
```

Applies the Sobel edge detection algorithm to an image.

This method implements the Sobel edge detection algorithm, which uses two 3x3 kernels to calculate the gradient magnitude at each pixel in the provided grayscale image. The gradient magnitude represents the edge strength. The resulting edge map replaces the original image data. This method is suitable for highlighting horizontal and vertical edges in the image.

Parameters

image	A constant reference to an Image object to which the Sobel filter will be applied. The image must be in grayscale format.

8.3.2.6 isGrayscale()

```
bool EdgeFilter::isGrayscale (
             const Image & image ) const  [private]
```

Checks if an image is in grayscale format.

This method determines whether the provided Image object is a grayscale image by checking its number of color channels. An image is considered grayscale if it has only one channel.

Parameters

image	A constant reference to an Image object to be checked.

Returns

True if the image has only one channel, indicating it is a grayscale image; false otherwise.

8.3.3 Member Data Documentation

8.3.3.1 filterType

```
FilterType EdgeFilter::filterType  [private]
```

8.3.3.2 paddingType

```
PaddingType EdgeFilter::paddingType  [private]
```

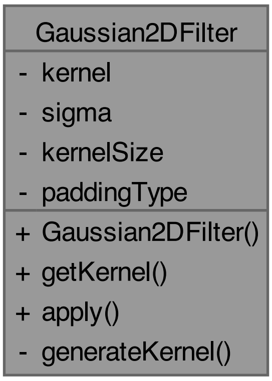
The documentation for this class was generated from the following files:

- Include/Filters/EdgeFilter.h
- Source/Filters/EdgeFilter.cpp

8.4 Gaussian2DFilter Class Reference

```
#include <Gaussian2DFilter.h>
```

Collaboration diagram for Gaussian2DFilter:



Public Member Functions

- Gaussian2DFilter (int kernelSize, double sigma=1.0, PaddingType paddingType=PaddingType::ZeroPadding)
- std::vector< std::vector< double > > getKernel () const
- void apply (Image &image) const

Private Member Functions

- void generateKernel ()

Private Attributes

- std::vector< std::vector< double > > kernel
- double sigma
- int kernelSize
- PaddingType paddingType

8.4.1 Constructor & Destructor Documentation

8.4.1.1 Gaussian2DFilter()

```
Gaussian2DFilter::Gaussian2DFilter (
              int kernelSize,
              double sigma = 1.0,
              PaddingType paddingType = PaddingType::ZeroPadding )
```

Constructor for the Gaussian2DFilter class.

Initializes a Gaussian2DFilter object with specified parameters for the Gaussian blur. The kernel size affects the blur extent by determining the size of the area around each pixel to consider in the blur calculation. The sigma value controls the spread of the blur. An odd kernel size ensures a symmetric area around each pixel. The constructor validates the kernel size to ensure it is odd, throwing an exception if it is not. This setup enables precise control over the blur effect applied to images.

Parameters

kernelSize	The size of the kernel for the Gaussian blur, which must be an odd number.
sigma	The standard deviation of the Gaussian distribution, determining the blur's spread.
paddingType	The type of padding to use when processing edges of the image.

Exceptions

std::invalid_argument	if kernelSize is not an odd number.

8.4.2 Member Function Documentation

8.4.2.1 apply()

```
void Gaussian2DFilter::apply (
            Image & image ) const
```

Applies the Gaussian blur to an image using the generated Gaussian kernel.

This method blurs the provided Image object by convolving it with the Gaussian kernel generated by `generateKernel`. It applies the blur separately to each channel of the image, accommodating images with multiple color channels. The method handles edge pixels according to the specified padding type, ensuring the blur extends to the edges of the image without artifacts. The blurred image replaces the original image data, resulting in a smoothly blurred version of the original image.

Parameters

image	A reference to an Image object representing the image to be blurred. The Image object must be initialized and loaded with data prior to calling this method.

8.4.2.2 generateKernel()

```
void Gaussian2DFilter::generateKernel ( )  [private]
```

Generates the Gaussian kernel based on the specified sigma and kernel size.

This method constructs the Gaussian kernel used for blurring the image. It calculates the value of each element in the kernel matrix based on the Gaussian function, ensuring the kernel is normalized so that its sum equals 1. This normalization is crucial for maintaining the original image's brightness level after the application of the blur. The kernel is stored internally within the Gaussian2DFilter object and used in the `apply` method to blur images.

8.4.2.3 getKernel()

```
std::vector< std::vector< double > > Gaussian2DFilter::getKernel ( ) const
```

Returns the Gaussian kernel used for blurring images.

This method provides access to the Gaussian kernel generated by the Gaussian2DFilter object. The kernel is a 2D matrix of double values representing the weights used in the convolution operation to apply the Gaussian blur. The kernel is normalized to ensure that the sum of its elements equals 1, preserving the image's brightness level during the blur.

Returns

A 2D vector of double values representing the Gaussian kernel.

8.4.3 Member Data Documentation

8.4.3.1 kernel

```
std::vector<std::vector<double> > Gaussian2DFilter::kernel  [private]
```

8.4.3.2 kernelSize

```
int Gaussian2DFilter::kernelSize  [private]
```

8.4.3.3 paddingType

```
PaddingType Gaussian2DFilter::paddingType  [private]
```

8.4.3.4 sigma

```
double Gaussian2DFilter::sigma  [private]
```

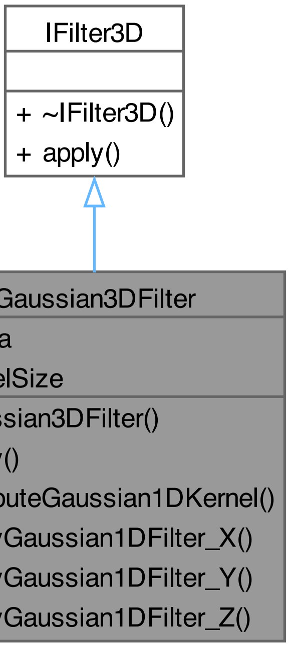
The documentation for this class was generated from the following files:

- Include/Filters/Gaussian2DFilter.h
- Source/Filters/Gaussian2DFilter.cpp

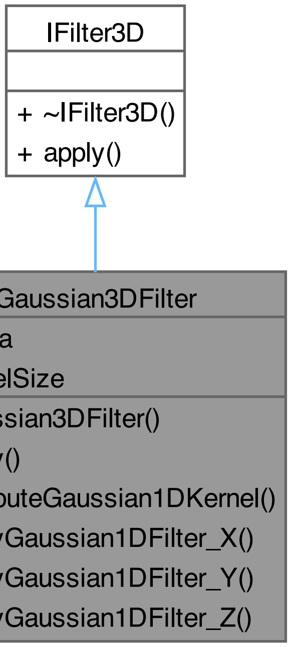
8.5 Gaussian3DFilter Class Reference

```
#include <Gaussian3DFilter.h>
```

Inheritance diagram for Gaussian3DFilter:


IFilter3D
+ ~IFilter3D()
+ apply()
Gaussian3DFilter
- sigma
- kernelSize
+ Gaussian3DFilter()
+ apply()
- computeGaussian1DKernel()
- applyGaussian1DFilter_X()
- applyGaussian1DFilter_Y()
- applyGaussian1DFilter_Z()

Collaboration diagram for Gaussian3DFilter:


IFilter3D
+ ~IFilter3D()
+ apply()
Gaussian3DFilter
- sigma
- kernelSize
+ Gaussian3DFilter()
+ apply()
- computeGaussian1DKernel()
- applyGaussian1DFilter_X()
- applyGaussian1DFilter_Y()
- applyGaussian1DFilter_Z()

Public Member Functions

- Gaussian3DFilter (double sigma, int kernelSize)
- void apply (Volume &volume) override

Public Member Functions inherited from IFilter3D

- virtual ∼IFilter3D ()=default

Private Member Functions

- std::vector< double > computeGaussian1DKernel () const
- void applyGaussian1DFilter_X (std::vector< unsigned char > &data, int width, int height, int depth)
- void applyGaussian1DFilter_Y (std::vector< unsigned char > &data, int width, int height, int depth)
- void applyGaussian1DFilter_Z (std::vector< unsigned char > &data, int width, int height, int depth)

Private Attributes

- double sigma
- int kernelSize

8.5.1 Constructor & Destructor Documentation

8.5.1.1 Gaussian3DFilter()

```
Gaussian3DFilter::Gaussian3DFilter (
             double sigma,
             int kernelSize )
```

Constructor for the Gaussian3DFilter class.

Initializes a Gaussian3DFilter object with a specified standard deviation (sigma) and kernel size. The kernel size determines the extent of the neighborhood around each voxel to be considered for filtering, and it must be an odd number to ensure a central voxel. Sigma determines the spread of the Gaussian kernel and thereby the extent of smoothing. This constructor validates the kernel size and throws an exception if the kernel size is not odd.

Parameters

sigma	The standard deviation of the Gaussian distribution used for the kernel.
kernelSize	The size of the kernel. It must be an odd number.

Exceptions

std::invalid_argument	if kernelSize is not an odd number.

8.5.2 Member Function Documentation

8.5.2.1 apply()

```
void Gaussian3DFilter::apply (
             Volume & volume )  [override], [virtual]
```

Applies the Gaussian filter to the entire volume.

This method orchestrates the application of the Gaussian filter to a 3D volume, smoothing the volume along all three axes (X, Y, and Z) sequentially. It achieves this by calling the applyGaussian1DFilter_X, applyGaussian1↩ DFilter_Y, and applyGaussian1DFilter_Z methods in succession, each applying the Gaussian kernel along one axis. The process results in a volume that is uniformly smoothed, reducing noise while preserving important structural information.

Parameters

volume	A reference to the Volume object representing the 3D data to be filtered.

Implements IFilter3D.

8.5.2.2 applyGaussian1DFilter_X()

```
void Gaussian3DFilter::applyGaussian1DFilter_X (
             std::vector< unsigned char > & data,
             int width,
             int height,
             int depth )  [private]
```

Applies the Gaussian filter along the X-axis of the volume.

This method convolves the volume data with the Gaussian kernel along the X-axis, effectively smoothing the volume along this direction. It uses the precomputed 1D Gaussian kernel for convolution, applying the kernel to each voxel's neighborhood along the X-axis and updating the voxel's value based on the weighted sum of its neighbors. The process results in a volume that is blurred along the X-axis while retaining its structure along the Y and Z axes.

Parameters

data	A reference to a vector of unsigned char representing the volume data to be filtered.
width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.

8.5.2.3 applyGaussian1DFilter_Y()

```
void Gaussian3DFilter::applyGaussian1DFilter_Y (
            std::vector< unsigned char > & data,
            int width,
            int height,
            int depth )  [private]
```

Applies the Gaussian filter along the Y-axis of the volume.

Similar to applyGaussian1DFilter_X, but the convolution is performed along the Y-axis. This method smooths the volume data along the Y-axis, applying the Gaussian kernel to each voxel's neighborhood in this direction. The resulting volume exhibits blurring along the Y-axis, with its characteristics along the X and Z axes preserved.

Parameters

data	A reference to a vector of unsigned char representing the volume data to be filtered.
width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.

8.5.2.4 applyGaussian1DFilter_Z()

```
void Gaussian3DFilter::applyGaussian1DFilter_Z (
            std::vector< unsigned char > & data,
            int width,
            int height,
            int depth )  [private]
```

Applies the Gaussian filter along the Z-axis of the volume.

This method extends the Gaussian smoothing process to the Z-axis, convolving the volume data with the Gaussian kernel along this direction. It smooths the volume along the Z-axis, applying the kernel to each voxel's neighborhood and updating the voxel's value accordingly. The operation blurs the volume along the Z-axis, maintaining its dimensions along the X and Y axes.

Parameters

data	A reference to a vector of unsigned char representing the volume data to be filtered.
width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.

8.5.2.5 computeGaussian1DKernel()

```
std::vector< double > Gaussian3DFilter::computeGaussian1DKernel ( ) const  [private]
```

Computes a 1D Gaussian kernel.

This member function calculates the coefficients of a 1D Gaussian kernel based on the class's sigma (standard deviation) and kernel size. The coefficients are computed using the Gaussian formula and are then normalized to ensure that the sum of the kernel coefficients is 1. This normalization is essential for maintaining the original amplitude of the signal after filtering. The computed kernel is symmetric and centered, suitable for convolution operations in one dimension.

Returns

: A vector of doubles containing the normalized coefficients of the 1D Gaussian kernel.

8.5.3 Member Data Documentation

8.5.3.1 kernelSize

```
int Gaussian3DFilter::kernelSize  [private]
```

8.5.3.2 sigma

```
double Gaussian3DFilter::sigma  [private]
```

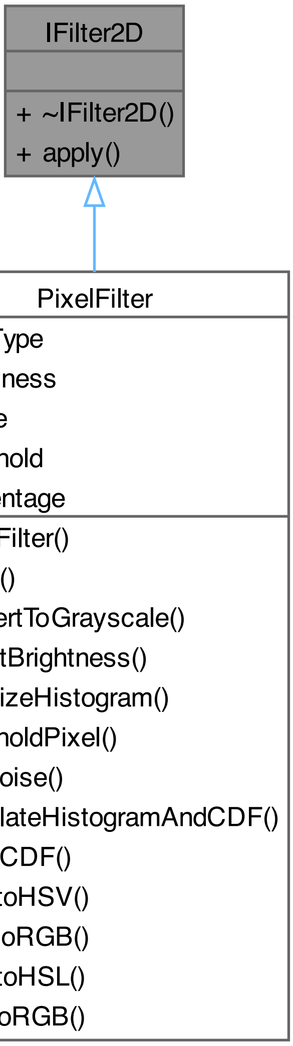
The documentation for this class was generated from the following files:

- Include/Filters/Gaussian3DFilter.h
- Source/Filters/Gaussian3DFilter.cpp

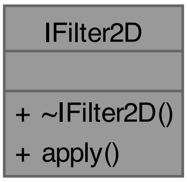
8.6 IFilter2D Class Reference

```
#include <Filter.h>
```

Inheritance diagram for IFilter2D:



Collaboration diagram for IFilter2D:



Public Member Functions

- virtual ~IFilter2D ()=default
- virtual void apply (Image &image)=0

8.6.1 Constructor & Destructor Documentation

8.6.1.1 ~IFilter2D()

```
virtual IFilter2D::~IFilter2D ( )  [virtual], [default]
```

Destructor for IFilter2D.

The destructor is declared as virtual to ensure that the destructor of any derived class is called when an object of the derived class is deleted through a pointer to the base class.

8.6.2 Member Function Documentation

8.6.2.1 apply()

```
virtual void IFilter2D::apply (
            Image & image )  [pure virtual]
```

Applies a filter to a 2D image.

This method applies a filter to a 2D image, modifying the image data in place.

Parameters

image	The image to which the filter will be applied.

Implemented in PixelFilter.

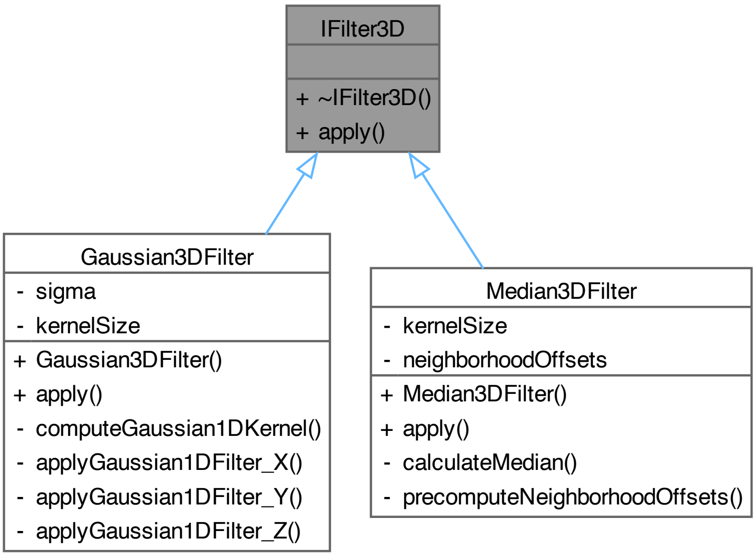
The documentation for this class was generated from the following file:

- Include/Filters/Filter.h

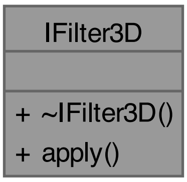
8.7 IFilter3D Class Reference

```
#include <Filter.h>
```

Inheritance diagram for IFilter3D:



Collaboration diagram for IFilter3D:



Public Member Functions

- virtual ~IFilter3D ()=default
- virtual void apply (Volume &volume)=0

8.7.1 Constructor & Destructor Documentation

8.7.1.1 ∼IFilter3D()

```
virtual IFilter3D::∼IFilter3D ( )  [virtual], [default]
```

Destructor for IFilter3D.

The destructor is declared as virtual to ensure that the destructor of any derived class is called when an object of the derived class is deleted through a pointer to the base class.

8.7.2 Member Function Documentation

8.7.2.1 apply()

```
virtual void IFilter3D::apply (
            Volume & volume )  [pure virtual]
```

Applies a filter to a 3D volume.

This method applies a filter to a 3D volume, modifying the volume data in place.

Parameters

volume	The volume to which the filter will be applied.

Implemented in Gaussian3DFilter, and Median3DFilter.

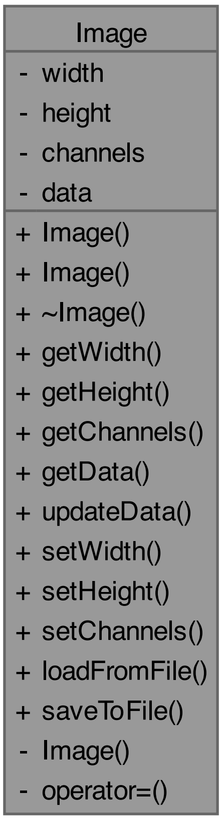
The documentation for this class was generated from the following file:

- Include/Filters/Filter.h

8.8 Image Class Reference

```
#include <Image.h>
```

Collaboration diagram for Image:



Public Member Functions

- Image ()
- Image (int width, int height, int channels, unsigned char ∗data)
- ∼Image ()
- int getWidth () const
- int getHeight () const
- int getChannels () const
- unsigned char ∗ getData () const
- void updateData (unsigned char ∗data)
- void setWidth (int width)
- void setHeight (int height)
- void setChannels (int channels)
- bool loadFromFile (const std::string &path)
- bool saveToFile (const std::string &path) const

Private Member Functions

- Image (const Image &)=delete
- Image & operator= (const Image &)=delete

Private Attributes

- int width
- int height
- int channels
- unsigned char ∗ data

8.8.1 Constructor & Destructor Documentation

8.8.1.1 Image() [1/3]

```
Image::Image (
             const Image & )  [private], [delete]
```

Private constructor to prevent copy construction.

The copy constructor is declared private to prevent unintentional deep copies of image data.

Parameters

other	The Image object to copy

8.8.1.2 Image() [2/3]

```
Image::Image ( )
```

Default constructor for the Image class.

The default constructor initializes the image dimensions and data to zero.

8.8.1.3 Image() [3/3]

```
Image::Image (
             int width,
             int height,
             int channels,
             unsigned char * data )
```

Constructor for the Image class with specified dimensions and data.

This constructor initializes the image with the specified dimensions and data.

Parameters

width	The width of the image
height	The height of the image
channels	The number of color channels in the image
data	The image data as an array of unsigned char

8.8.1.4 ∼Image()

```
Image::∼Image ( )
```

Destructor for the Image class.

The destructor frees the memory allocated for the image data.

8.8.2 Member Function Documentation

8.8.2.1 getChannels()

```
int Image::getChannels ( ) const
```

Get the number of color channels in the image.

This member function returns the number of color channels in the image.

Returns

The number of color channels in the image

8.8.2.2 getData()

```
unsigned char * Image::getData ( ) const
```

Get the image data.

This member function returns a pointer to the image data.

Returns

A pointer to the image data

8.8.2.3 getHeight()

```
int Image::getHeight ( ) const
```

Get the height of the image.

This member function returns the height of the image.

Returns

The height of the image

8.8.2.4 getWidth()

```
int Image::getWidth ( ) const
```

Get the width of the image.

This member function returns the width of the image.

Returns

The width of the image

8.8.2.5 loadFromFile()

```
bool Image::loadFromFile (
             const std::string & path )
```

Loads an image from a file

This member function of the Image class loads an image from the specified file path into the object's data buffer using the stb_image library. It sets the image's width, height, and channels based on the loaded image's properties. If the image cannot be loaded, it prints an error message to standard error.

Parameters

path	A string representing the path to the image file to be loaded.

Returns

: A boolean value indicating the success (true) or failure (false) of loading the image.

8.8.2.6 operator=()

```
Image & Image::operator= (
             const Image & ) [private], [delete]
```

Private assignment operator to prevent assignment.

The assignment operator is declared private to prevent unintentional deep copies of image data.

Parameters

other	The Image object to assign

Returns

The reference to the assigned Image object

8.8.2.7 saveToFile()

```
bool Image::saveToFile (
            const std::string & path ) const
```

Saves the image to a file

This const member function of the Image class attempts to save the current image data to a file at the specified path. It uses the stb_image_write library to write the image in PNG format. Before attempting to save, it checks if the image data buffer is not empty. If there is no data, or if the save operation fails, it prints an error message to standard error.

Parameters

path	A string representing the file path where the image should be saved. The image will be saved in PNG format.

Returns

: A boolean value indicating the success (true) or failure (false) of the save operation.

8.8.2.8 setChannels()

```
void Image::setChannels (
            int channels )
```

Set the number of color channels in the image.

This member function sets the number of color channels in the image.

Parameters

channels	The new number of color channels in the image

8.8.2.9 setHeight()

```
void Image::setHeight (
            int height )
```

Set the height of the image.

This member function sets the height of the image.

Parameters

height	The new height of the image

8.8.2.10 setWidth()

```
void Image::setWidth (
             int width )
```

Set the width of the image.

This member function sets the width of the image.

Parameters

width	The new width of the image

8.8.2.11 updateData()

```
void Image::updateData (
             unsigned char * data )
```

Update the image data.

This member function updates the image data with the specified data.

Parameters

data	The new image data as an array of unsigned char

8.8.3 Member Data Documentation

8.8.3.1 channels

```
int Image::channels  [private]
```

8.8.3.2 data

```
unsigned char* Image::data  [private]
```

8.8.3.3 height

```
int Image::height  [private]
```

8.8.3.4 width

```
int Image::width  [private]
```

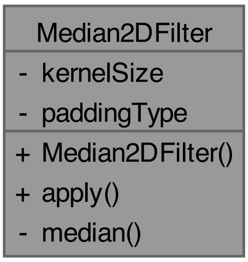
The documentation for this class was generated from the following files:

- Include/Image.h
- Source/Image.cpp

8.9 Median2DFilter Class Reference

```
#include <Median2DFilter.h>
```

Collaboration diagram for Median2DFilter:



Public Member Functions

- Median2DFilter (int kernelSize, PaddingType paddingType=PaddingType::ZeroPadding)
- void apply (Image &image) const

Static Private Member Functions

- static unsigned char median (std::vector< unsigned char > &window)

Private Attributes

- int kernelSize
- PaddingType paddingType

8.9.1 Constructor & Destructor Documentation

8.9.1.1 Median2DFilter()

```
Median2DFilter::Median2DFilter (
            int kernelSize,
            PaddingType paddingType = PaddingType::ZeroPadding )
```

Constructor for the Median2DFilter class.

Initializes a Median2DFilter object with a specified kernel size and padding type. Ensures that the kernel size is odd, which is necessary for symmetrically surrounding each pixel with neighbors. This requirement is crucial for the correct application of the median filter. If an even kernel size is provided, an exception is thrown. The padding type determines the strategy for handling pixels at the borders of the image, ensuring consistent filtering across the entire image.

Parameters

kernelSize	The size of the kernel, must be an odd number for symmetric application.
paddingType	The type of padding to use at the image borders, determining how edge pixels are treated.

Exceptions

std::invalid_argument	if the kernelSize is not an odd number.

8.9.2 Member Function Documentation

8.9.2.1 apply()

```
void Median2DFilter::apply (
             Image & image ) const
```

Applies median filtering to an image.

This method processes the provided Image object, applying median filtering to reduce noise while preserving edges. It operates by sliding a window, defined by the kernel size, across the image and replacing each pixel's value with the median value of its neighborhood. This approach is effective at removing salt-and-pepper noise. The method handles different channels of the image separately, maintaining the color integrity of the original image.

Parameters

image	A reference to the Image object to be filtered. The image is modified in place, receiving the filtered output.

8.9.2.2 median()

```
unsigned char Median2DFilter::median (
             std::vector< unsigned char > & window )  [static], [private]
```

Calculates the median value of a pixel window.

This helper function determines the median value within a given window of pixel values. It employs the quick select algorithm to efficiently find the median, which is particularly effective for non-linear filtering operations like median filtering. The function handles both odd and even-sized windows, returning the middle value for odd-sized windows or the average of the two middle values for even-sized windows. This method ensures that the median filter can be applied consistently across the entire image, including edge pixels.

Parameters

window	A vector of unsigned char representing the intensity values of pixels within the kernel window.

Returns

The median intensity value as an unsigned char.

8.9.3 Member Data Documentation

8.9.3.1 kernelSize

```
int Median2DFilter::kernelSize [private]
```

8.9.3.2 paddingType

```
PaddingType Median2DFilter::paddingType [private]
```

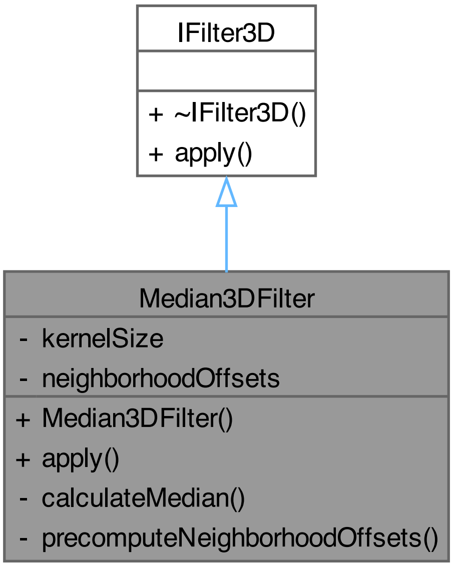
The documentation for this class was generated from the following files:

- Include/Filters/Median2DFilter.h
- Source/Filters/Median2DFilter.cpp

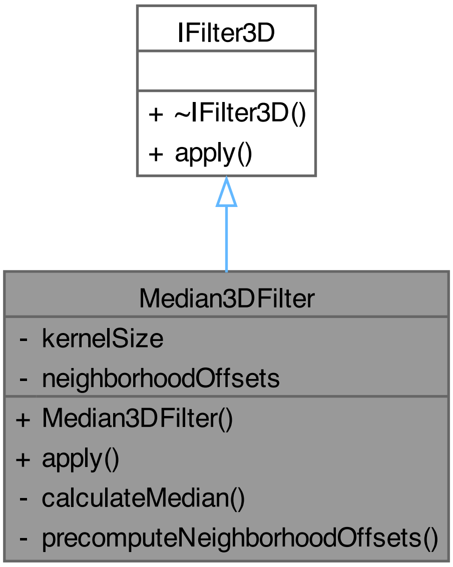
8.10 Median3DFilter Class Reference

```
#include <Median3DFilter.h>
```

Inheritance diagram for Median3DFilter:



Collaboration diagram for Median3DFilter:



Public Member Functions

- Median3DFilter (int kernelSize)
- void apply (Volume &volume) override

Public Member Functions inherited from IFilter3D

- virtual ∼IFilter3D ()=default

Private Member Functions

- unsigned char calculateMedian (std::vector< unsigned char > &neighborhood)
- void precomputeNeighborhoodOffsets (int width, int height, int depth)

Private Attributes

- int kernelSize
- std::vector< std::vector< int > > neighborhoodOffsets

8.10.1 Constructor & Destructor Documentation

8.10.1.1 Median3DFilter()

```
Median3DFilter::Median3DFilter (
            int kernelSize )  [explicit]
```

Constructor for the Median3DFilter class.

Initializes a Median3DFilter object with a specified kernel size for the median filter. The kernel size determines the size of the neighborhood around each voxel from which the median is calculated. The kernel size must be an odd number to ensure there is a central voxel.

Parameters

kernelSize	An integer specifying the size of the kernel. Must be an odd number.

Exceptions

std::invalid_argument	if kernelSize is not an odd number.

8.10.2 Member Function Documentation

8.10.2.1 apply()

```
void Median3DFilter::apply (
            Volume & volume )  [override], [virtual]
```

Applies the median filter to a volume.

This method applies the median filter to a 3D volume represented by a Volume object. The filter iterates through each voxel in the volume and replaces its value with the median value from the neighborhood around that voxel. The neighborhood size is determined by the kernel size specified during object creation. The median filter is a powerful tool for reducing noise in volume data while preserving structural details. This method efficiently computes the median values using precomputed neighborhood offsets to enhance the performance of the filtering operation. Histogram equalization is a critical step in enhancing the contrast of images and improving their visual quality.

Parameters

volume	A reference to a Volume object representing the 3D data to which the median filter will be applied.

Implements IFilter3D.

8.10.2.2 calculateMedian()

```
unsigned char Median3DFilter::calculateMedian (
            std::vector< unsigned char > & neighborhood )  [private]
```

Calculates the median value from a neighborhood of voxel values.

This method finds the median value in a given vector of unsigned chars representing the intensity values of voxels in a neighborhood. If the size of the neighborhood is odd, the median is the middle element. If it is even, the median is the average of the two middle elements. This calculation is critical for the median filtering process, providing the capability to reduce noise while preserving edges in the volume.

Parameters

neighborhood	A reference to a vector of unsigned chars representing voxel values in a neighborhood.

Returns

: The median value as an unsigned char.

8.10.2.3 precomputeNeighborhoodOffsets()

```
void Median3DFilter::precomputeNeighborhoodOffsets (
             int width,
             int height,
             int depth )  [private]
```

Precomputes the offsets for the filter's neighborhood based on the kernel size.

This method calculates the offsets from each voxel within the kernel's neighborhood, taking into account the dimensions of the volume to ensure that the offsets stay within bounds. The offsets are stored in a vector for quick access during the filtering process. This precomputation step optimizes the median filtering operation by avoiding repeated boundary checks for each voxel during the filter application.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.

8.10.3 Member Data Documentation

8.10.3.1 kernelSize

```
int Median3DFilter::kernelSize  [private]
```

8.10.3.2 neighborhoodOffsets

```
std::vector<std::vector<int> > Median3DFilter::neighborhoodOffsets  [private]
```

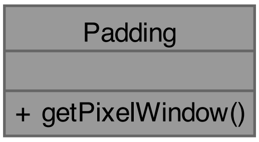
The documentation for this class was generated from the following files:

- Include/Filters/Median3DFilter.h
- Source/Filters/Median3DFilter.cpp

8.11 Padding Class Reference

```
#include <Padding.h>
```

Collaboration diagram for Padding:



Static Public Member Functions

- static std::vector< unsigned char > getPixelWindow (const Image &image, int x, int y, int c, int kernelSize, PaddingType paddingType)

8.11.1 Member Function Documentation

8.11.1.1 getPixelWindow()

```
std::vector< unsigned char > Padding::getPixelWindow (
              const Image & image,
              int x,
              int y,
              int c,
              int kernelSize,
              PaddingType paddingType ) [static]
```

Retrieves a pixel window around a specified location in an image, applying the selected padding strategy.

This function extracts a window of pixel values centered around a specified pixel location (x, y) in the provided image. The size of the window is determined by the kernelSize parameter. The function supports various padding strategies to handle pixels near the image borders effectively. These strategies include zero padding (padding with zeros), edge replication (replicating the edge pixels), and reflect padding (mirroring the pixels near the edge). This functionality is crucial for filter operations that require contextual information about a pixel's neighborhood, ensuring consistent processing across the entire image, including its edges.

Parameters

image	A constant reference to the Image object from which the pixel window will be extracted.
x	The x-coordinate of the central pixel in the window.
y	The y-coordinate of the central pixel in the window.
c	The channel of the image to be processed.
kernelSize	The size of the window to be extracted, which determines how far from the central pixel the window extends.
paddingType	The padding strategy to be applied when the window extends beyond the image boundaries.

Returns

A vector of unsigned char values representing the pixel intensities within the window. The vector size corresponds to the total number of pixels within the window, adjusted for the selected padding strategy as necessary.

Exceptions

std::invalid_argument	if an unsupported padding type is specified.

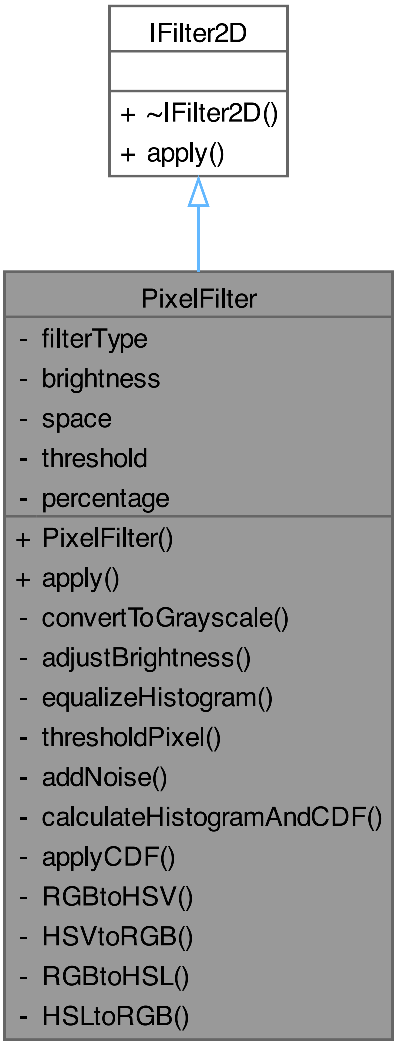
The documentation for this class was generated from the following files:

- Include/Filters/Padding.h
- Source/Filters/Padding.cpp

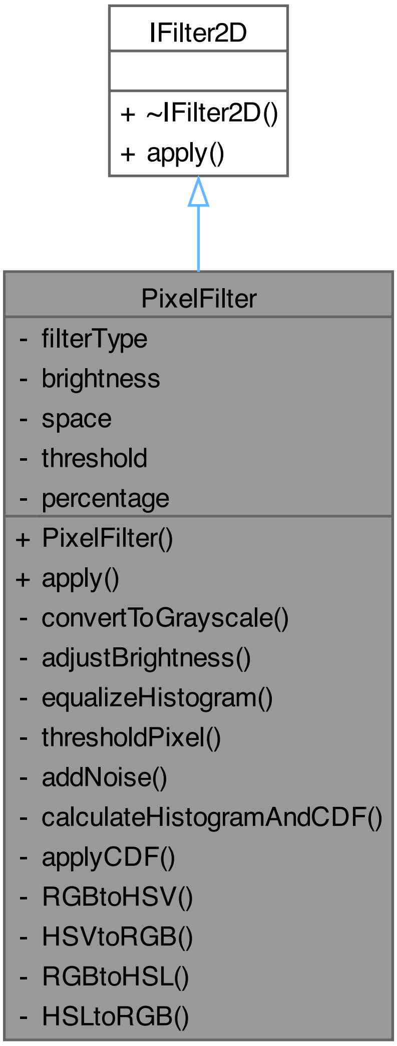
8.12 PixelFilter Class Reference

```
#include <PixelFilter.h>
```

Inheritance diagram for PixelFilter:



Collaboration diagram for PixelFilter:



Public Member Functions

- PixelFilter (const std::string &type, const std::optional< int > &brightness=std::nullopt, const std::string &space="", int threshold=0, double percentage=0)
- void apply (Image &image) override

Public Member Functions inherited from IFilter2D

- virtual ~IFilter2D ()=default

Private Member Functions

- void convertToGrayscale (Image &image)
- void adjustBrightness (Image &image)
- void equalizeHistogram (Image &image)
- void thresholdPixel (Image &image)
- void addNoise (Image &image)
- void calculateHistogramAndCDF (const std::vector< float > &channel, std::vector< int > &histogram, std↩ ::vector< int > &cdf)
- void applyCDF (std::vector< float > &channel, const std::vector< int > &cdf)
- void RGBtoHSV (float r, float g, float b, float &h, float &s, float &v)
- void HSVtoRGB (float h, float s, float v, float &r, float &g, float &b)
- void RGBtoHSL (float r, float g, float b, float &h, float &s, float &l)
- void HSLtoRGB (float h, float s, float l, float &r, float &g, float &b)

Private Attributes

- std::string filterType
- int brightness = 128
- std::string space
- int threshold = 0
- double percentage = 0

8.12.1 Constructor & Destructor Documentation

8.12.1.1 PixelFilter()

```
PixelFilter::PixelFilter (
             const std::string & type,
             const std::optional< int > & brightness = std::nullopt,
             const std::string & space = "",
             int threshold = 0,
             double percentage = 0 )
```

Constructor for the PixelFilter class.

Initializes a PixelFilter object with specified parameters for different filtering operations. The type parameter dictates the filter operation to be applied (e.g., grayscale conversion, brightness adjustment). Optional parameters like brightness adjustment value, color space for histogram equalization, threshold for thresholding, and noise percentage for adding salt-and-pepper noise can also be specified. The constructor validates input parameters to ensure they are within acceptable ranges for the specified filter type.

Parameters

type	The type of filter operation to be applied.
brightness	Optional parameter specifying the brightness adjustment value (applicable if type is "Brightness").
space	Optional parameter specifying the color space for histogram equalization (applicable if type is "Equalisation").
threshold	Optional parameter specifying the threshold value for thresholding operations (applicable if type is "Thresholding").
percentage	Optional parameter specifying the percentage of pixels affected by salt-and-pepper noise (applicable if type is "SaltAndPepperNoise").

Exceptions

std::invalid_argument	if any parameter is outside its expected range based on the filter type.

8.12.2 Member Function Documentation

8.12.2.1 addNoise()

```
void PixelFilter::addNoise (
            Image & image )  [private]
```

Adds salt-and-pepper noise to an image.

This method introduces salt-and-pepper noise to the provided Image object by randomly selecting a specified percentage of pixels and setting them to either the maximum value (255, salt) or the minimum value (0, pepper). This operation is designed to simulate common types of noise found in digital images.

Parameters

image	A reference to an Image object that will have noise added. The image is modified in place.

8.12.2.2 adjustBrightness()

```
void PixelFilter::adjustBrightness (
            Image & image )  [private]
```

Adjusts the brightness of an image.

This method modifies the brightness of each pixel in the provided Image object based on a specified brightness adjustment value. The brightness is adjusted by adding the brightness value to each pixel's intensity, with the result clamped to the [0, 255] range. The operation affects all channels of the image equally.

Parameters

image	A reference to an Image object whose brightness will be adjusted. The image is modified in place.

8.12.2.3 apply()

```
void PixelFilter::apply (
            Image & image )  [override], [virtual]
```

Applies the specified filtering operation to an image.

Based on the filter type specified during object construction, this method applies the corresponding image processing operation to the provided Image object. Operations include grayscale conversion, brightness adjustment, histogram equalization, thresholding, and adding salt-and-pepper noise. The method ensures that operations are applied correctly based on the provided parameters and the image's properties. Unsupported filter types result in an exception.

Parameters

image	A reference to an Image object that will be modified by the filter operation.

Exceptions

std::invalid_argument	if the filter type is unsupported.

Implements IFilter2D.

8.12.2.4 applyCDF()

```
void PixelFilter::applyCDF (
            std::vector< float > & channel,
            const std::vector< int > & cdf )  [private]
```

Applies the cumulative distribution function to a channel of image data.

After calculating the cumulative distribution function (CDF) for an image channel, this method maps each pixel's intensity in the channel to a new value based on the CDF. This mapping enhances the contrast of the channel according to the histogram equalization process. This method is crucial for the final step of applying histogram equalization to an image.

Parameters

channel	A reference to a vector of floats representing the normalized intensity values of a single channel of the image, to be modified in place.
cdf	A vector of ints representing the CDF, used to map the original intensities to their new values.

8.12.2.5 calculateHistogramAndCDF()

```
void PixelFilter::calculateHistogramAndCDF (
            const std::vector< float > & channel,
            std::vector< int > & histogram,
            std::vector< int > & cdf )  [private]
```

Calculates the histogram and cumulative distribution function (CDF) for a given image channel.

This helper function computes the histogram of the provided channel data and then calculates the cumulative distribution function based on the histogram. These calculations are used in histogram equalization processes to adjust the contrast of an image. The CDF is later used to map the original pixel values to their new values for contrast enhancement.

Parameters

channel	A vector of floats representing the normalized intensity values of a single channel of the image.
histogram	A reference to a vector of ints where the histogram data will be stored.
cdf	A reference to a vector of ints where the CDF data will be stored.

8.12.2.6 convertToGrayscale()

```
void PixelFilter::convertToGrayscale (
             Image & image )  [private]
```

Converts an image to grayscale.

This method transforms a color image into grayscale by calculating the luminance of each pixel based on its red, green, and blue components. The grayscale image replaces the original image data. This operation is performed only if the image has more than one color channel.

Parameters

image	A reference to an Image object to be converted to grayscale. The image is modified in place.

8.12.2.7 equalizeHistogram()

```
void PixelFilter::equalizeHistogram (
             Image & image )  [private]
```

Equalizes the histogram of an image.

This method applies histogram equalization to the provided Image object, enhancing the contrast of the image. It can operate on different color spaces, including RGB, HSL, and HSV. For color images, the equalization is applied to the luminance or value channel, preserving color integrity while improving contrast. This method is suitable for images that suffer from poor contrast due to lighting conditions or exposure settings.

Parameters

image	A reference to an Image object whose histogram will be equalized. The image is modified in place.

8.12.2.8 HSLtoRGB()

```
void PixelFilter::HSLtoRGB (
             float h,
             float s,
             float l,
             float & r,
             float & g,
             float & b )  [private]
```

Converts HSL color space to RGB color space.

This function reverses the process of RGB to HSL conversion, translating colors from the HSL (Hue, Saturation, Lightness) space back to RGB. This is crucial for displaying processed images on devices that use RGB color space or for further processing in RGB. Adjustments made in HSL, which can be more intuitive for certain image manipulations, are accurately reflected in the RGB output.

Parameters

h	The hue component in HSL space.

Parameters

s	The saturation component in HSL space.
l	The lightness component in HSL space.
r	Reference to a float to store the calculated red component in RGB space.
g	Reference to a float to store the calculated green component in RGB space.
b	Reference to a float to store the calculated blue component in RGB space.

8.12.2.9 HSVtoRGB()

```
void PixelFilter::HSVtoRGB (
             float h,
             float s,
             float v,
             float & r,
             float & g,
             float & b )  [private]
```

Converts HSV color space to RGB color space.

This function converts the color of a pixel from the HSV color space back to the RGB color space. The conversion allows changes made in the HSV space, such as contrast adjustments or thresholding, to be represented in the conventional RGB color space for display or further processing. This method ensures that operations performed in HSV space can be accurately reflected in the final image.

Parameters

h	The hue component of the color.
s	The saturation component of the color.
v	The value component of the color.
r	Reference to a float to store the red component.
g	Reference to a float to store the green component.
b	Reference to a float to store the blue component.

8.12.2.10 RGBtoHSL()

```
void PixelFilter::RGBtoHSL (
             float r,
             float g,
             float b,
             float & h,
             float & s,
             float & l )  [private]
```

Converts RGB color space to HSL color space.

This function transforms the RGB representation of a color into its HSL counterpart. HSL (Hue, Saturation, Lightness) offers a more intuitive way to describe color adjustments such as changing brightness or saturation. This conversion is particularly useful for image processing tasks that require manipulation of color properties distinct from their brightness, like adjusting colors or performing color-based thresholding.

Parameters

r	The red component of the color.
g	The green component of the color.
b	The blue component of the color.
h	Reference to a float to store the hue component.
s	Reference to a float to store the saturation component.
l	Reference to a float to store the light
l	Reference to a float to store the lightness component.

8.12.2.11 RGBtoHSV()

```
void PixelFilter::RGBtoHSV (
             float r,
             float g,
             float b,
             float & h,
             float & s,
             float & v )  [private]
```

Converts RGB color space to HSV color space.

This function converts the color of a pixel from the RGB color space to the HSV color space. The conversion process takes into account the chroma and the lightness of the color to accurately represent it in HSV space. This conversion is useful for operations that are more effectively performed in HSV space, such as thresholding based on color saturation or value.

Parameters

r	The red component of the color.
g	The green component of the color.
b	The blue component of the color.
h	Reference to a float to store the hue component.
s	Reference to a float to store the saturation component.
v	Reference to a float to store the value component.

8.12.2.12 thresholdPixel()

```
void PixelFilter::thresholdPixel (
             Image & image )  [private]
```

Applies thresholding to an image.

This method applies a thresholding operation to the provided Image object. Pixels with intensity above the threshold are set to the maximum value (255), and those below are set to zero, effectively binarizing the image. The operation can be applied to images in different color spaces, including RGB, HSL, and HSV, based on the specified parameters.

Parameters

image	A reference to an Image object to be thresholded. The image is modified in place.

8.12.3 Member Data Documentation

8.12.3.1 brightness

```
int PixelFilter::brightness = 128  [private]
```

8.12.3.2 filterType

```
std::string PixelFilter::filterType  [private]
```

8.12.3.3 percentage

```
double PixelFilter::percentage = 0  [private]
```

8.12.3.4 space

```
std::string PixelFilter::space  [private]
```

8.12.3.5 threshold

```
int PixelFilter::threshold = 0  [private]
```

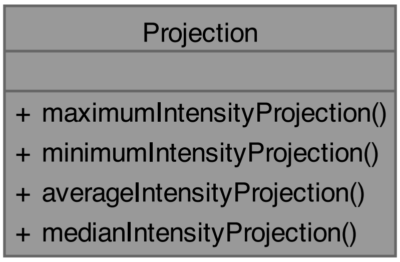
The documentation for this class was generated from the following files:

- Include/Filters/PixelFilter.h
- Source/Filters/PixelFilter.cpp

8.13 Projection Class Reference

```
#include <Projection.h>
```

Collaboration diagram for Projection:


Projection
+ maximumIntensityProjection()
+ minimumIntensityProjection()
+ averageIntensityProjection()
+ medianIntensityProjection()

Static Public Member Functions

- static std::vector< unsigned char > maximumIntensityProjection (int width, int height, int depth, const unsigned char ∗data)
- static std::vector< unsigned char > minimumIntensityProjection (int width, int height, int depth, const unsigned char ∗data)
- static std::vector< unsigned char > averageIntensityProjection (int width, int height, int depth, const unsigned char ∗data)
- static std::vector< unsigned char > medianIntensityProjection (int width, int height, int depth, const unsigned char ∗data)

8.13.1 Member Function Documentation

8.13.1.1 averageIntensityProjection()

```
std::vector< unsigned char > Projection::averageIntensityProjection (
                int width,
                int height,
                int depth,
                const unsigned char ∗ data )  [static]
```

Computes the Average Intensity Projection (AIP) of a 3D volume

This static member function of the Projection class calculates the Average Intensity Projection of a given 3D volume. AIP is a volume rendering method that projects the average intensity value of voxels along a particular direction, in this case, the z-axis, onto a 2D plane. It iterates through each voxel in the volume, accumulates the intensity values for each (x, y) position on the projection plane, and divides the total by the depth of the volume. The result is a 2D image where each pixel represents the average intensity value found at that (x, y) position through the entire depth of the volume. AIP is useful for visualizing the general structure and distribution of densities within the volume.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	A pointer to the volume's raw data.

Returns

: A vector of unsigned char representing the 2D AIP image.

8.13.1.2 maximumIntensityProjection()

```
std::vector< unsigned char > Projection::maximumIntensityProjection (
                int width,
                int height,
                int depth,
                const unsigned char ∗ data )  [static]
```

Computes the Maximum Intensity Projection (MIP) of a 3D volume

This static member function of the Projection class calculates the Maximum Intensity Projection of a given 3D volume. MIP is a volume rendering method that projects the voxel with the highest intensity value along a particular direction, in this case, the z-axis (depth), onto a 2D plane. This function iterates through each voxel in the volume and, for each (x, y) position on the projection plane, it selects the maximum intensity value found along the z-axis. The result is a 2D image (represented as a vector of unsigned char) where each pixel corresponds to the maximum intensity value found at that (x, y) position through the entire depth of the volume. This technique is commonly used in medical imaging to highlight areas of highest density or intensity in volumetric data.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	A pointer to the volume's raw data.

Returns

: A vector of unsigned char representing the 2D MIP image.

8.13.1.3 medianIntensityProjection()

```
std::vector< unsigned char > Projection::medianIntensityProjection (
            int width,
            int height,
            int depth,
            const unsigned char * data )  [static]
```

Computes the Median Intensity Projection (MedIP) of a 3D volume

This static member function of the Projection class calculates the Median Intensity Projection of a given 3D volume. MedIP is a volume rendering technique that projects the median intensity value of voxels along a specific direction, in this case, the z-axis, onto a 2D plane. This function iterates through each voxel in the volume and, for each (x, y) position on the projection plane, it selects the median intensity value found along the z-axis from the depth of the volume. The median is determined using the quickselect algorithm for efficiency. The result is a 2D image where each pixel represents the median intensity value at that (x, y) position throughout the entire depth of the volume. MedIP can provide a balanced visualization that may reduce the impact of outliers in highly variable volumetric data.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	A pointer to the volume's raw data.

Returns

: A vector of unsigned char representing the 2D MedIP image.

8.13.1.4 minimumIntensityProjection()

```
std::vector< unsigned char > Projection::minimumIntensityProjection (
            int width,
            int height,
            int depth,
            const unsigned char * data )  [static]
```

Computes the Minimum Intensity Projection (MinIP) of a 3D volume

This static member function of the Projection class calculates the Minimum Intensity Projection of a given 3D volume. MinIP is a volume rendering technique that projects the voxel with the lowest intensity value along a specific direction, in this case, the z-axis (depth), onto a 2D plane. This function iterates through each voxel in the volume and, for each (x, y) position on the projection plane, it selects the minimum intensity value found along the z-axis. The result is a 2D image (represented as a vector of unsigned char) where each pixel corresponds to the minimum intensity value found at that (x, y) position throughout the entire depth of the volume. This technique can be useful for visualizing low-density areas in volumetric data.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	A pointer to the volume's raw data.

Returns

: A vector of unsigned char representing the 2D MinIP image.

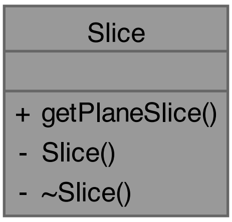
The documentation for this class was generated from the following files:

- Include/Projection.h
- Source/Projection.cpp

8.14 Slice Class Reference

`#include <Slice.h>`

Collaboration diagram for Slice:



Static Public Member Functions

- static std::vector< unsigned char > getPlaneSlice (int width, int height, int depth, const unsigned char ∗data, const std::string &plane, int sliceIndex)

Private Member Functions

- Slice ()=delete
- ∼Slice ()=delete

8.14.1 Constructor & Destructor Documentation

8.14.1.1 Slice()

```
Slice::Slice ( )  [private], [delete]
```

Default constructor for the Slice class.

The default constructor of the Slice class is private and deleted to prevent the creation of Slice objects. The Slice class provides a static member function for extracting slices from 3D volumes and does not require instantiation. By making the constructor private and deleted, the class enforces this design pattern and ensures that no Slice objects can be created.

8.14.1.2 ∼Slice()

```
Slice::∼Slice ( )  [private], [delete]
```

Destructor for the Slice class.

The destructor of the Slice class is private and deleted to prevent the destruction of Slice objects. Since the Slice class does not have any member variables or instance-specific functionality, there is no need to create or destroy Slice objects. By making the destructor private and deleted, the class enforces this design pattern and ensures that no Slice objects can be destroyed, maintaining the static nature of the Slice class.

8.14.2 Member Function Documentation

8.14.2.1 getPlaneSlice()

```
std::vector< unsigned char > Slice::getPlaneSlice (
              int width,
              int height,
              int depth,
              const unsigned char * data,
              const std::string & plane,
              int sliceIndex )  [static]
```

Retrieves a specific slice from a 3D volume

This static member function of the Slice class extracts and returns a specific slice from a 3D volume, given the volume's dimensions, the raw data, the desired plane for slicing ('x-y', 'x-z', 'y-z'), and the slice index. The function first checks the validity of the slice index and the specified plane. Depending on the plane, it computes the starting position of the slice in the volume's data array and copies the relevant data into a vector of unsigned char, which it then returns. If the slice index is out of range for the specified plane or if an invalid plane is specified, the function will print an error message and return an empty vector. This function allows for flexible access to different orientations of slices within the volume, facilitating various analyses and visualizations of 3D data.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	A pointer to the volume's raw data.
plane	A string specifying the plane of the slice ('x-y', 'x-z', 'y-z').
sliceIndex	An integer indicating the index of the slice to retrieve, starting from 1.

Returns

: A vector of unsigned char containing the data of the requested slice. Returns an empty vector if an error occurs.

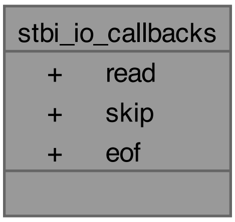
The documentation for this class was generated from the following files:

- Include/Slice.h
- Source/Slice.cpp

8.15 stbi_io_callbacks Struct Reference

```
#include <stb_image.h>
```

Collaboration diagram for stbi_io_callbacks:



Public Attributes

- int(∗ read)(void ∗user, char ∗data, int size)
- void(∗ skip)(void ∗user, int n)
- int(∗ eof)(void ∗user)

8.15.1 Member Data Documentation

8.15.1.1 eof

```
int(∗ stbi_io_callbacks::eof) (void ∗user)
```

8.15.1.2 read

```
int(* stbi_io_callbacks::read) (void *user, char *data, int size)
```

8.15.1.3 skip

```
void(* stbi_io_callbacks::skip) (void *user, int n)
```

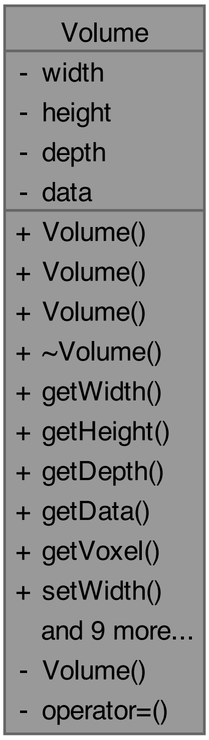
The documentation for this struct was generated from the following file:

- Include/stb_image.h

8.16 Volume Class Reference

```
#include <Volume.h>
```

Collaboration diagram for Volume:



Public Member Functions

- Volume ()

  Default constructor for the Volume class.
- Volume (int width, int height, int depth)

  Constructor for the Volume class.
- Volume (int width, int height, int depth, unsigned char ∗data)

  Constructor for the Volume class.
- ∼Volume ()

  Destructor for the Volume class.
- int getWidth () const

  Get the width of the volume.
- int getHeight () const

  Get the height of the volume.
- int getDepth () const

  Get the depth of the volume.
- unsigned char ∗ getData () const

  Get the volume data.
- unsigned char getVoxel (int x, int y, int z) const

  Get the voxel value at the specified coordinates.
- void setWidth (int width)

  Set the width of the volume.
- void setHeight (int height)

  Set the height of the volume.
- void setDepth (int depth)

  Set the depth of the volume.
- void updateData (const std::vector< unsigned char > &newData)
- bool loadFromFiles (const std::vector< std::string > &paths)
- bool loadFromDirectory (const std::string &directoryPath)
- void save (const std::string &path, const std::string &plane) const
- void save (const std::string &path, const std::string &plane, int sliceIndex) const
- void save (const std::string &path, const std::string &plane, std::string projector) const
- void save (const std::string &path, const std::string &plane, const std::string &projector, int begin, int end) const

Private Member Functions

- Volume (const Volume &)=delete

  Load volume data from a single file.
- Volume & operator= (const Volume &)=delete

  Prevents assignment of volume data.

Private Attributes

- int width
- int height
- int depth
- unsigned char ∗ data

8.16.1 Constructor & Destructor Documentation

8.16.1.1 Volume() [1/4]

```
Volume::Volume (
            const Volume & )  [private], [delete]
```

Load volume data from a single file.

This function loads volume data from a single file and updates the volume's data field.

Parameters

path	The path to the file containing the volume data.

8.16.1.2 Volume() [2/4]

```
Volume::Volume ( )
```

Default constructor for the Volume class.

This constructor initializes a Volume object with default values.

8.16.1.3 Volume() [3/4]

```
Volume::Volume (
            int width,
            int height,
            int depth )
```

Constructor for the Volume class.

This constructor initializes a Volume object with the specified width, height, and depth.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.

8.16.1.4 Volume() [4/4]

```
Volume::Volume (
            int width,
            int height,
            int depth,
            unsigned char * data )
```

Constructor for the Volume class.

This constructor initializes a Volume object with the specified width, height, depth, and data.

Parameters

width	The width of the volume.
height	The height of the volume.
depth	The depth of the volume.
data	The volume data.

8.16.1.5 ∼Volume()

```
Volume::∼Volume ( )
```

Destructor for the Volume class.

This destructor deallocates the memory used by the volume data.

8.16.2 Member Function Documentation

8.16.2.1 getData()

```
unsigned char * Volume::getData ( ) const
```

Get the volume data.

This function returns the volume data.

Returns

The volume data.

8.16.2.2 getDepth()

```
int Volume::getDepth ( ) const
```

Get the depth of the volume.

This function returns the depth of the volume.

Returns

The depth of the volume.

8.16.2.3 getHeight()

```
int Volume::getHeight ( ) const
```

Get the height of the volume.

This function returns the height of the volume.

Returns

The height of the volume.

8.16.2.4 getVoxel()

```
unsigned char Volume::getVoxel (
             int x,
             int y,
             int z ) const
```

Get the voxel value at the specified coordinates.

This function returns the voxel value at the specified coordinates.

Parameters

x	The x-coordinate of the voxel.
y	The y-coordinate of the voxel.
z	The z-coordinate of the voxel.

Returns

The voxel value at the specified coordinates.

8.16.2.5 getWidth()

```
int Volume::getWidth ( ) const
```

Get the width of the volume.

This function returns the width of the volume.

Returns

The width of the volume.

8.16.2.6 loadFromDirectory()

```
bool Volume::loadFromDirectory (
            const std::string & directoryPath )
```

Loads a 3D volume from image files located in a specified directory

This member function of the Volume class loads a series of image files from a given directory to construct a 3D volume. It iterates through the directory, collecting paths of all regular files. These paths are then sorted to ensure that the images are loaded in the correct order, which is crucial for correctly assembling the 3D volume. The function leverages the loadFromFiles member function to load the images into the volume. If the directory cannot be read or contains no image files, the function prints an error message and returns false.

Parameters

directoryPath	A string representing the path to the directory containing the image files to be loaded.

Returns

: A boolean value indicating the success (true) or failure (false) of loading the volume from the directory.

8.16.2.7 loadFromFiles()

```
bool Volume::loadFromFiles (
            const std::vector< std::string > & paths )
```

Loads a 3D volume from multiple image files

This member function of the Volume class loads a series of images from the specified file paths to construct a 3D volume. The images are assumed to have the same dimensions and number of channels. The function initializes the volume's depth based on the number of paths provided, checks if at least one path is provided, and then loads each image in sequence. The images are concatenated into a single data buffer representing the 3D volume. If any image fails to load, or if memory allocation for the entire volume data fails, it prints an error message to standard error and cleans up any allocated resources.

Parameters

paths	A vector of strings representing the paths to the image files that comprise the volume.

Returns

: A boolean value indicating the success (true) or failure (false) of loading the volume.

8.16.2.8 operator=()

```
Volume & Volume::operator= (
            const Volume & )  [private], [delete]
```

Prevents assignment of volume data.

This function prevents the assignment of volume data by deleting the assignment operator.

Parameters

volume	The volume to assign to.

Returns

The assigned volume.

8.16.2.9 save() [1/4]

```
void Volume::save (
            const std::string & path,
            const std::string & plane ) const
```

Saves all slices along a specified plane to files

This member function of the Volume class saves all slices along the specified plane to files in the specified directory. The function first checks if the plane is valid, if the output directory exists, and if the plane is valid. If any of these checks fail, it prints an error message and returns. If the output directory does not exist, the function attempts to create it. It then iterates through all slices along the specified plane, extracts each slice from the volume's data, saves the result to a file in the specified directory, and uses the stb_image_write library to write the slice data to a PNG file.

Parameters

path	A string representing the path to the directory where the slices will be saved.
plane	A string representing the plane along which the slices will be extracted. Valid planes are 'x-y', 'x-z', and 'y-z'.

Returns

: None

8.16.2.10 save() [2/4]

```
void Volume::save (
            const std::string & path,
            const std::string & plane,
            const std::string & projector,
            int begin,
            int end ) const
```

Saves a range-based projection to a file

This member function of the Volume class computes a range-based projection along the specified plane and saves the result to a file. The function first checks if the plane and projector are valid, and if the specified range is within the bounds of the volume. If any of these checks fail, it prints an error message and returns. If the output directory does not exist, the function attempts to create it. It then extracts the specified range of slices from the volume's data, computes the projection using the specified projector, and saves the result to a file in the specified directory. The function uses the stb_image_write library to write the projection data to a PNG file.

Parameters

path	A string representing the path to the directory where the projection will be saved.
plane	A string representing the plane along which the projection will be computed. Valid planes are 'x-y', 'x-z', and 'y-z'.
projector	A string representing the type of projection to be computed. Valid projectors are 'MIP', 'MinIP', 'AIP', and 'MedIP'.
begin	An integer representing the starting index of the range of slices to be projected.
end	An integer representing the ending index of the range of slices to be projected.

Returns

: None

8.16.2.11 save() [3/4]

```
void Volume::save (
            const std::string & path,
            const std::string & plane,
            int sliceIndex ) const
```

Saves a specific slice to a file

This member function of the Volume class saves a specific slice along the specified plane to a file. The function first checks if the plane is valid, if the output directory exists, and if the slice index is within the bounds of the volume. If any of these checks fail, it prints an error message and returns. If the output directory does not exist, the function attempts to create it. It then extracts the specified slice from the volume's data, saves the result to a file in the specified directory, and uses the stb_image_write library to write the slice data to a PNG file.

Parameters

path	A string representing the path to the directory where the slice will be saved.
plane	A string representing the plane along which the slice will be extracted. Valid planes are 'x-y', 'x-z', and 'y-z'.
sliceIndex	An integer representing the index of the slice to be saved.

Returns

: None

8.16.2.12 save() [4/4]

```
void Volume::save (
            const std::string & path,
            const std::string & plane,
            std::string projector ) const
```

Saves a specific projection to a file

This member function of the Volume class computes a specific projection along the specified plane and saves the result to a file. The function first checks if the plane and projector are valid, and if the output directory exists. If any of these checks fail, it prints an error message and returns. If the output directory does not exist, the function attempts to create it. It then computes the specified projection using the Projection class and saves the result to a file in the specified directory. The function uses the stb_image_write library to write the projection data to a PNG file.

Parameters

path	A string representing the path to the directory where the projection will be saved.
plane	A string representing the plane along which the projection will be computed. Valid planes are 'x-y', 'x-z', and 'y-z'.
projector	A string representing the type of projection to be computed. Valid projectors are 'MIP', 'MinIP', 'AIP', and 'MedIP'.

Returns

: None

8.16.2.13 setDepth()

```
void Volume::setDepth (
             int depth )
```

Set the depth of the volume.

This function sets the depth of the volume.

Parameters

depth	The depth of the volume.

8.16.2.14 setHeight()

```
void Volume::setHeight (
             int height )
```

Set the height of the volume.

This function sets the height of the volume.

Parameters

height	The height of the volume.

8.16.2.15 setWidth()

```
void Volume::setWidth (
             int width )
```

Set the width of the volume.

This function sets the width of the volume.

Parameters

width	The width of the volume.

8.16.2.16 updateData()

```
void Volume::updateData (
            const std::vector< unsigned char > & newData )
```

Updates the volume data with new data

This member function of the Volume class updates the volume's data array with new data provided via the newData parameter. Before proceeding with the update, it checks if the size of the new data matches the expected size of the volume (calculated as width ∗ height ∗ depth). If the sizes do not match, it prints an error message and aborts the update. If the volume's data pointer is null (indicating that the volume has not been initialized with data), it allocates new memory to store the newData. Otherwise, it reuses the existing memory block and copies the new←Data into the volume's data array. This function ensures that the volume's data is kept up-to-date with any changes or transformations applied externally.

Parameters

newData	A constant reference to a vector of unsigned char, representing the new data to be copied into the volume's data array.

Returns

: None

8.16.3 Member Data Documentation

8.16.3.1 data

```
unsigned char* Volume::data [private]
```

8.16.3.2 depth

```
int Volume::depth [private]
```

8.16.3.3 height

```
int Volume::height [private]
```

8.16.3.4 width

```
int Volume::width  [private]
```

The documentation for this class was generated from the following files:

- Include/Volume.h
- Source/Volume.cpp

Chapter 9

File Documentation

9.1 Build/CMakeFiles/3.28.3/CompilerIdCXX/CMakeCXXCompilerId.cpp File Reference

Macros

- #define __has_include(x) 0
- #define COMPILER_ID ""
- #define STRINGIFY_HELPER(X) #X
- #define STRINGIFY(X) STRINGIFY_HELPER(X)
- #define PLATFORM_ID
- #define ARCHITECTURE_ID
- #define DEC(n)
- #define HEX(n)
- #define CXX_STD __cplusplus

Functions

- int main (int argc, char ∗argv[ ])

Variables

- char const ∗ info_compiler = "INFO" ":" "compiler[" COMPILER_ID "]"
- char const ∗ info_platform = "INFO" ":" "platform[" PLATFORM_ID "]"
- char const ∗ info_arch = "INFO" ":" "arch[" ARCHITECTURE_ID "]"
- const char ∗ info_language_standard_default
- const char ∗ info_language_extensions_default

9.1.1 Macro Definition Documentation

9.1.1.1 __has_include

```
#define __has_include(
              x ) 0
```

9.1.1.2 ARCHITECTURE_ID

```
#define ARCHITECTURE_ID
```

9.1.1.3 COMPILER_ID

```
#define COMPILER_ID ""
```

9.1.1.4 CXX_STD

```
#define CXX_STD __cplusplus
```

9.1.1.5 DEC

```
#define DEC(
              n )
```

Value:

```
  ('0' + (((n) / 10000000)%10)), \
  ('0' + (((n) / 1000000)%10)),  \
  ('0' + (((n) / 100000)%10)),   \
  ('0' + (((n) / 10000)%10)),    \
  ('0' + (((n) / 1000)%10)),     \
  ('0' + (((n) / 100)%10)),      \
  ('0' + (((n) / 10)%10)),       \
  ('0' +  ((n) % 10))
```

9.1.1.6 HEX

```
#define HEX(
              n )
```

Value:

```
  ('0' + ((n)»28 & 0xF)), \
  ('0' + ((n)»24 & 0xF)), \
  ('0' + ((n)»20 & 0xF)), \
  ('0' + ((n)»16 & 0xF)), \
  ('0' + ((n)»12 & 0xF)), \
  ('0' + ((n)»8  & 0xF)), \
  ('0' + ((n)»4  & 0xF)), \
  ('0' + ((n)     & 0xF))
```

9.1.1.7 PLATFORM_ID

```
#define PLATFORM_ID
```

9.1.1.8 STRINGIFY

```
#define STRINGIFY(
              X ) STRINGIFY_HELPER(X)
```

9.1.1.9 STRINGIFY_HELPER

```
#define STRINGIFY_HELPER(
              X ) #X
```

9.1.2 Function Documentation

9.1.2.1 main()

```
int main (
             int argc,
             char * argv[] )
```

9.1.3 Variable Documentation

9.1.3.1 info_arch

```
char const* info_arch = "INFO" ":" "arch[" ARCHITECTURE_ID "]"
```

9.1.3.2 info_compiler

```
char const* info_compiler = "INFO" ":" "compiler[" COMPILER_ID "]"
```

9.1.3.3 info_language_extensions_default

```
const char* info_language_extensions_default
```

Initial value:

```
= "INFO" ":" "extensions_default["




  "OFF"

"]"
```

9.1.3.4 info_language_standard_default

```
const char* info_language_standard_default
```

Initial value:

```
= "INFO" ":" "standard_default["











  "98"

"]"
```

9.1.3.5 info_platform

```
char const* info_platform = "INFO" ":" "platform[" PLATFORM_ID "]"
```

9.2 Build/CMakeFiles/advanced_programming_group_radix_sort.dir/↩ Source/main.cpp.o.d File Reference

9.3 Build/Source/CMakeFiles/core_lib.dir/main.cpp.o.d File Reference

9.4 Build/Tests/CMakeFiles/runTests.dir/main.cpp.o.d File Reference

9.5 Build/Source/CMakeFiles/core_lib.dir/Algorithm.cpp.o.d File Reference

9.6 Build/Source/CMakeFiles/core_lib.dir/Filters/Box2DFilter.cpp.o.d File Reference

9.7 Build/Source/CMakeFiles/core_lib.dir/Filters/EdgeFilter.cpp.o.d File Reference

9.8 Build/Source/CMakeFiles/core_lib.dir/Filters/Gaussian2DFilter.cpp.↩ o.d File Reference

9.9 Build/Source/CMakeFiles/core_lib.dir/Filters/Gaussian3DFilter.cpp.↩ o.d File Reference

9.10 Build/Source/CMakeFiles/core_lib.dir/Filters/Median2DFilter.cpp.o.d File Reference

9.11 Build/Source/CMakeFiles/core_lib.dir/Filters/Median3DFilter.cpp.o.d File Reference

9.12 Build/Source/CMakeFiles/core_lib.dir/Filters/Padding.cpp.o.d File Reference

9.13 Build/Source/CMakeFiles/core_lib.dir/Filters/PixelFilter.cpp.o.d File Reference

9.14 Build/Source/CMakeFiles/core_lib.dir/Image.cpp.o.d File Reference

9.15 Build/Source/CMakeFiles/core_lib.dir/Projection.cpp.o.d File Reference

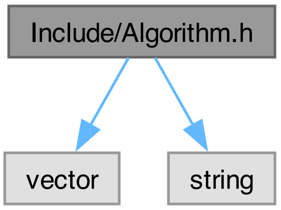
9.16 Build/Source/CMakeFiles/core_lib.dir/Slice.cpp.o.d File Reference

9.17 Build/Source/CMakeFiles/core_lib.dir/Volume.cpp.o.d File Reference

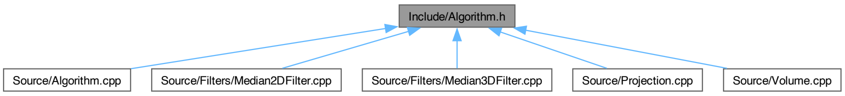
9.18 Include/Algorithm.h File Reference

```
#include <vector>
#include <string>
```

Include dependency graph for Algorithm.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Algorithm

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H

9.18.1 Detailed Description

Implements a radix sort algorithm for efficient data sorting.

The RadixSortAlgorithm class provides an implementation of the radix sort algorithm, which is a non-comparative integer sorting algorithm. It sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A radix sort can be applied to data that can be sorted lexicographically, be it integers or strings. This implementation supports both least significant digit (LSD) and most significant digit (MSD) radix sorts. It is part of the data sorting and manipulation toolkit developed by the Advanced Programming Group.

Date

Created on March 19, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.18.2 Macro Definition Documentation

9.18.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H
```

9.19 Algorithm.h

Go to the documentation of this file.

```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H

#include <vector>
#include <string>

class Algorithm {
public:
    static unsigned char quickSelect(std::vector<unsigned char> &arr, int left, int right, int k);

    static void quickSort(std::vector<std::string> &arr, int low, int high);

    static int partition(std::vector<unsigned char> &arr, int left, int right, int pivotIndex);

    static int partition(std::vector<std::string> &arr, int low, int high);
};


#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_ALGORITHM_H
```

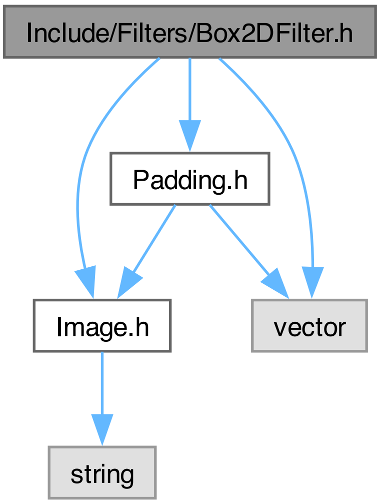
9.20 Include/Filters/Box2DFilter.h File Reference

Implements a 2D box filter for image processing.

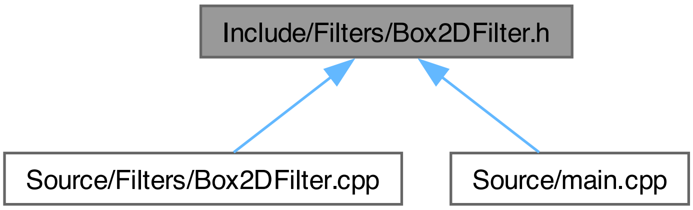
```
#include "Image.h"
#include "Padding.h"
```

```
#include <vector>
```

Include dependency graph for Box2DFilter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Box2DFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H

9.20.1 Detailed Description

Implements a 2D box filter for image processing.

The Box2DFilter class provides functionality for applying a 2D box filtering operation on images. It supports various padding strategies to handle edge cases. The primary purpose of this filter is to perform spatial averaging across an image, which can be useful for blurring or smoothing. This implementation allows for customization of the kernel size and padding type, making it versatile for different image processing needs. This class is a contribution of the Advanced Programming Group, aimed at enhancing tools available for image manipulation and processing tasks.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.20.2 Macro Definition Documentation

9.20.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H
```

9.21 Box2DFilter.h

Go to the documentation of this file.

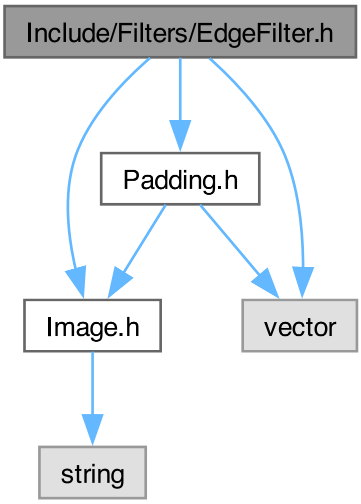
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H

#include "Image.h"
#include "Padding.h"

#include <vector>

class Box2DFilter {
private:
    int kernelSize; // Size of the kernel
    PaddingType paddingType; // Padding strategy

public:
    Box2DFilter(int kernelSize, PaddingType paddingType = PaddingType::ZeroPadding);

    void apply(Image &image) const;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_BOX2DFILTER_H
```

9.22 Include/Filters/EdgeFilter.h File Reference

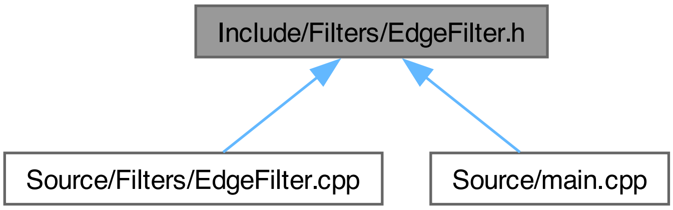
Implements edge detection filters for image processing.

```
#include "Image.h"
#include "Padding.h"
#include <vector>
```

Include dependency graph for EdgeFilter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class EdgeFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H

Enumerations

- enum class FilterType { Sobel , Prewitt , Scharr , Roberts }

9.22.1 Detailed Description

Implements edge detection filters for image processing.

The EdgeFilter class enables edge detection in images through the application of various edge detection algorithms, including Sobel, Prewitt, Scharr, and Roberts filters. It allows for customization of the filter type and padding strategy, accommodating different requirements for edge detection tasks. This class facilitates the extraction of edges, which is a crucial step in many image processing applications such as feature detection, image segmentation, and object recognition. Developed by the Advanced Programming Group, this tool enhances the capabilities for image analysis and processing.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.22.2 Macro Definition Documentation

9.22.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H
```

9.22.3 Enumeration Type Documentation

9.22.3.1 FilterType

```
enum class FilterType  [strong]
```

Enumerator

Sobel	
Prewitt	
Scharr	
Roberts	

9.23 EdgeFilter.h

Go to the documentation of this file.

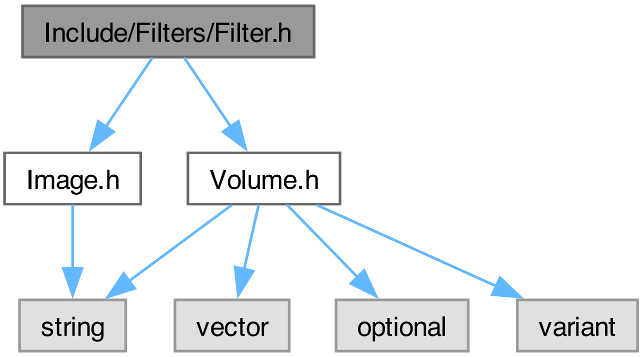
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H

#include "Image.h"
#include "Padding.h"

#include <vector>

enum class FilterType {
    Sobel,
    Prewitt,
    Scharr,
    Roberts
};

class EdgeFilter {
private:
    FilterType filterType; // Add filter type attribute
    PaddingType paddingType; // Add padding type attribute

    void applySobel(Image &image) const;

    void applyPrewitt(Image &image) const;

    void applyScharr(Image &image) const;

    void applyRoberts(Image &image) const;

    bool isGrayscale(const Image &image) const;

public:
    EdgeFilter(FilterType type, PaddingType paddingType = PaddingType::ZeroPadding);

    void apply(Image &image);
};

#endif // ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_EDGEFILTER_H
```

9.24 Include/Filters/Filter.h File Reference

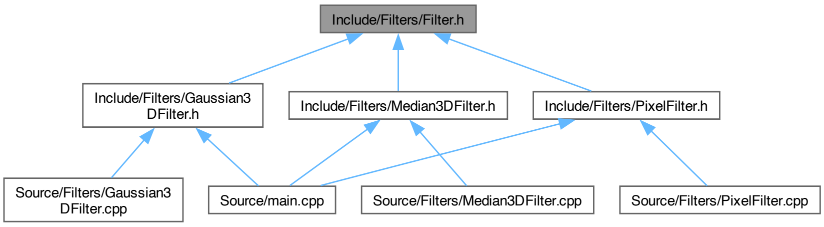
Defines interfaces for 2D and 3D filtering operations on images and volume data.

```
#include "Image.h"
#include "Volume.h"
```

Include dependency graph for Filter.h:



This graph shows which files directly or indirectly include this file:


Include/Filters/Filter.h
Include/Filters/Gaussian3DFilter.h
Include/Filters/Median3DFilter.h
Include/Filters/PixelFilter.h
Source/Filters/Gaussian3DFilter.cpp
Source/main.cpp
Source/Filters/Median3DFilter.cpp
Source/Filters/PixelFilter.cpp

Classes

- class IFilter3D
- class IFilter2D

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H

9.24.1 Detailed Description

Defines interfaces for 2D and 3D filtering operations on images and volume data.

This file contains the IFilter2D and IFilter3D interfaces, which establish the foundational structure for implementing filters on 2D images and 3D volume data, respectively. These interfaces are designed to enforce a uniform approach to applying various filtering techniques across different data dimensions. They are integral to the Advanced Programming Group's efforts in standardizing and enhancing the tools available for image and volume data manipulation and processing.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.24.2 Macro Definition Documentation

9.24.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H
```

9.25 Filter.h

Go to the documentation of this file.

```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H

#include "Image.h"
#include "Volume.h"

// Interface for 3D filters
class IFilter3D {
public:
    virtual ~IFilter3D() = default;

    virtual void apply(Volume &volume) = 0;
};

// Interface for 2D filters
class IFilter2D {
public:
    virtual ~IFilter2D() = default;

    virtual void apply(Image &image) = 0;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_FILTER_H
```

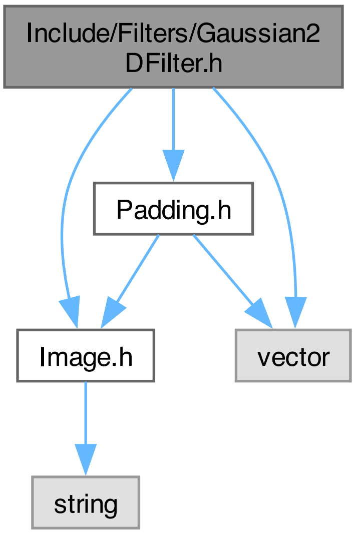
9.26 Include/Filters/Gaussian2DFilter.h File Reference

Implements a Gaussian 2D filter for image blurring.

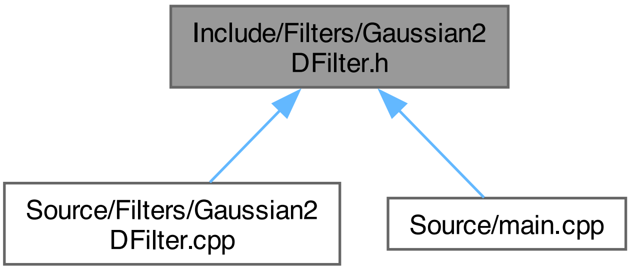
```
#include "Image.h"
#include "Padding.h"
```

```
#include <vector>
```

Include dependency graph for Gaussian2DFilter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Gaussian2DFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H

9.26.1 Detailed Description

Implements a Gaussian 2D filter for image blurring.

The Gaussian2DFilter class encapsulates the functionality required to apply a Gaussian blur to 2D images. This process involves creating a Gaussian kernel based on the specified sigma and kernel size, and applying this kernel to an image with the chosen padding type. Gaussian blurring is widely used in image processing for noise reduction, detail smoothing, and preparing images for higher-level processing tasks. The design allows for easy adjustment of blur intensity and compatibility with various image padding strategies. Created as part of the Advanced Programming Group's project initiatives, this tool aids in sophisticated image manipulation and enhancement techniques.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.26.2 Macro Definition Documentation

9.26.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H
```

9.27 Gaussian2DFilter.h

Go to the documentation of this file.

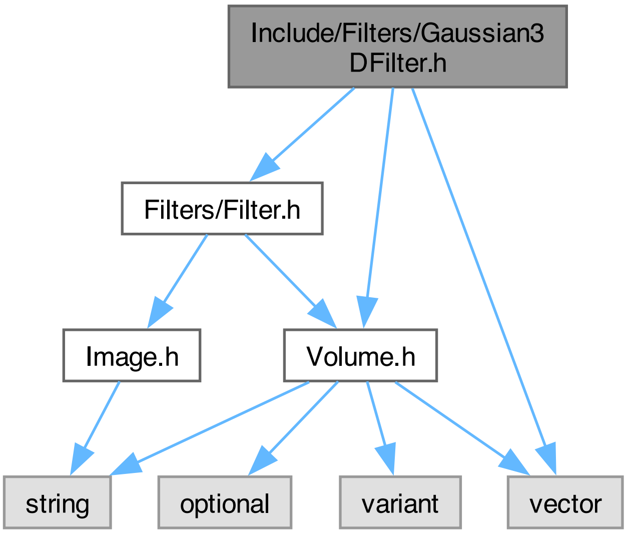
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H

#include "Image.h"
#include "Padding.h"

#include <vector>

class Gaussian2DFilter {
private:
    std::vector <std::vector<double» kernel; // Gaussian kernel
    double sigma; // standard deviation
    int kernelSize; // size of the kernel
    PaddingType paddingType; // padding type

    void generateKernel();

public:
    Gaussian2DFilter(int kernelSize, double sigma = 1.0, PaddingType paddingType =
PaddingType::ZeroPadding);

    std::vector<std::vector<double» getKernel() const;

    void apply(Image &image) const;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN2DFILTER_H
```

9.28 Include/Filters/Gaussian3DFilter.h File Reference

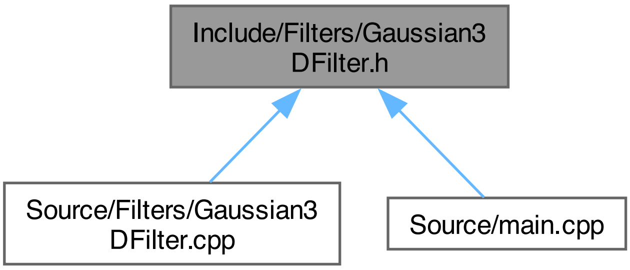
Implements a 3D Gaussian filter for smoothing volume data.

```
#include "Filters/Filter.h"
#include "Volume.h"
#include <vector>
```

Include dependency graph for Gaussian3DFilter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Gaussian3DFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H

9.28.1 Detailed Description

Implements a 3D Gaussian filter for smoothing volume data.

The Gaussian3DFilter class applies a Gaussian smoothing operation to 3D volume data. It performs the smoothing separately along the X, Y, and Z axes using a 1D Gaussian kernel. This class is part of the tools developed by the Advanced Programming Group to facilitate volume data manipulation and processing.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.28.2 Macro Definition Documentation

9.28.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H
```

9.29 Gaussian3DFilter.h

Go to the documentation of this file.

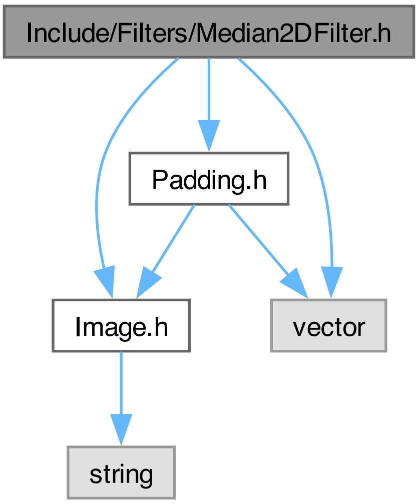
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H

#include "Filters/Filter.h"
#include "Volume.h"

#include <vector>

class Gaussian3DFilter : public IFilter3D {
private:
    double sigma; // Standard deviation of the Gaussian
    int kernelSize; // Size of the kernel

    std::vector<double> computeGaussian1DKernel() const;

    void applyGaussian1DFilter_X(std::vector<unsigned char> &data, int width, int height, int depth);

    void applyGaussian1DFilter_Y(std::vector<unsigned char> &data, int width, int height, int depth);

    void applyGaussian1DFilter_Z(std::vector<unsigned char> &data, int width, int height, int depth);

public:
    Gaussian3DFilter(double sigma, int kernelSize);

    void apply(Volume &volume) override;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_GAUSSIAN3DFILTER_H
```

9.30 Include/Filters/Median2DFilter.h File Reference

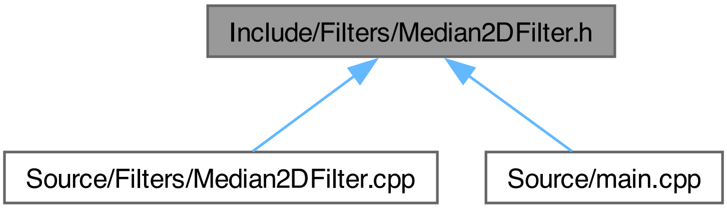
Implements a Median 2D filter for noise reduction in images.

```
#include "Image.h"
#include "Padding.h"
#include <vector>
```

Include dependency graph for Median2DFilter.h:



This graph shows which files directly or indirectly include this file:


Include/Filters/Median2DFilter.h
Source/Filters/Median2DFilter.cpp
Source/main.cpp

Classes

- class Median2DFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H

9.30.1 Detailed Description

Implements a Median 2D filter for noise reduction in images.

The Median2DFilter class applies a median filtering operation to 2D images to reduce noise while preserving edges. This technique is particularly useful in pre-processing for other image analysis tasks, as it helps to reduce salt-and-pepper noise without blurring sharp edges in the image. The filter supports customizable kernel sizes and padding types, allowing for flexibility in handling different types of images and noise levels. Part of the tools developed by the Advanced Programming Group, this class enhances the suite of image processing capabilities with a focus on maintaining image quality during noise reduction.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.30.2 Macro Definition Documentation

9.30.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H
```

9.31 Median2DFilter.h

Go to the documentation of this file.

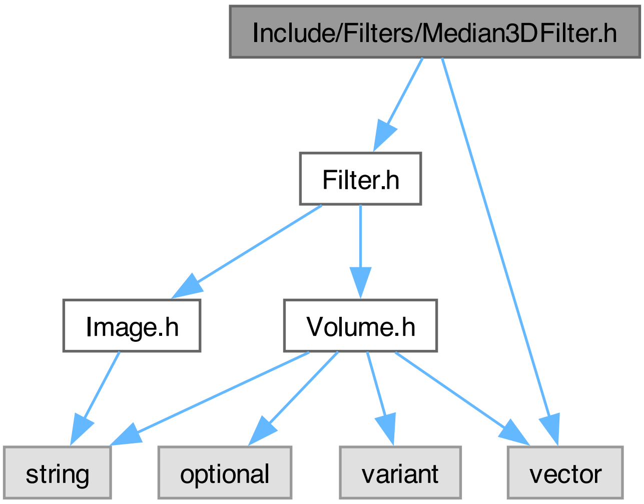
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H

#include "Image.h"
#include "Padding.h"

#include <vector>

class Median2DFilter {
private:
    int kernelSize; // Kernel size
    PaddingType paddingType; // Padding type

    static unsigned char median(std::vector<unsigned char> &window);

public:
    Median2DFilter(int kernelSize, PaddingType paddingType = PaddingType::ZeroPadding);

    void apply(Image &image) const;

};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN2DFILTER_H
```

9.32 Include/Filters/Median3DFilter.h File Reference

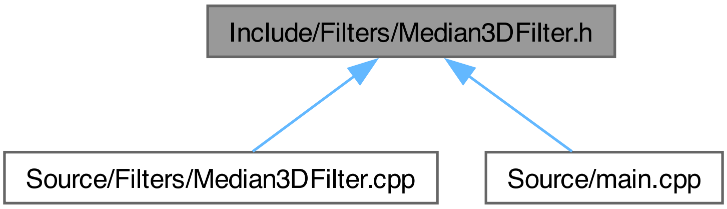
Implements a 3D Median filter for volume data processing.

```
#include "Filter.h"
#include <vector>
```

Include dependency graph for Median3DFilter.h:



This graph shows which files directly or indirectly include this file:


Include/Filters/Median3DFilter.h
Source/Filters/Median3DFilter.cpp
Source/main.cpp

Classes

- class Median3DFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H

9.32.1 Detailed Description

Implements a 3D Median filter for volume data processing.

The Median3DFilter class applies a median filtering operation to 3D volume data. It is designed to reduce noise within the volume data while preserving edges by replacing each voxel's value with the median value within a specified neighborhood around that voxel. The class supports customizable kernel sizes and efficiently computes the median values using precomputed neighborhood offsets. This class is a part of the volume data manipulation toolkit developed by the Advanced Programming Group.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.32.2 Macro Definition Documentation

9.32.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H
```

9.33 Median3DFilter.h

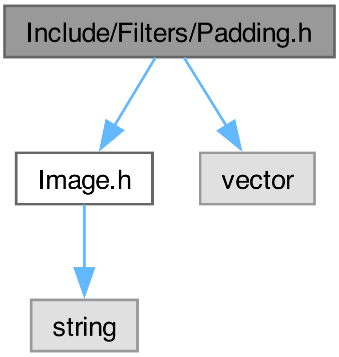
Go to the documentation of this file.

```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H

#include "Filter.h"

#include <vector>

class Median3DFilter : public IFilter3D {
private:
    int kernelSize; // The size of the kernel.
    std::vector<std::vector<int» neighborhoodOffsets; // The offsets of the neighborhood.

    unsigned char calculateMedian(std::vector<unsigned char> &neighborhood);

    void precomputeNeighborhoodOffsets(int width, int height, int depth);

public:
    explicit Median3DFilter(int kernelSize);

    void apply(Volume &volume) override;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_MEDIAN3DFILTER_H
```

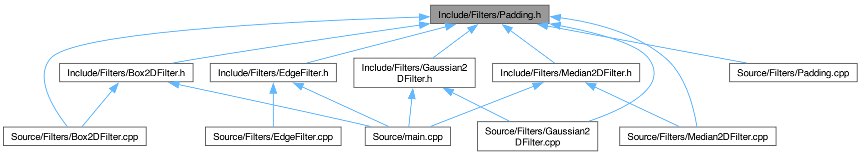
9.34 Include/Filters/Padding.h File Reference

```
#include "Image.h"
#include <vector>
```

Include dependency graph for Padding.h:



This graph shows which files directly or indirectly include this file:


Include/Filters/Padding.h
Include/Filters/Box2DFilter.h
Include/Filters/EdgeFilter.h
Include/Filters/Gaussian2 DFilter.h
Include/Filters/Median2DFilter.h
Source/Filters/Padding.cpp
Source/Filters/Box2DFilter.cpp
Source/Filters/EdgeFilter.cpp
Source/main.cpp
Source/Filters/Gaussian2 DFilter.cpp
Source/Filters/Median2DFilter.cpp

Classes

- class Padding

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDING_H

Enumerations

- enum class PaddingType { ZeroPadding , EdgeReplication , ReflectPadding }

9.34.1 Macro Definition Documentation

9.34.1.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDING_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDING_H
```

9.34.2 Enumeration Type Documentation

9.34.2.1 PaddingType

```
enum class PaddingType  [strong]
```

Enumerator

ZeroPadding	
EdgeReplication	
ReflectPadding	

9.35 Padding.h

Go to the documentation of this file.

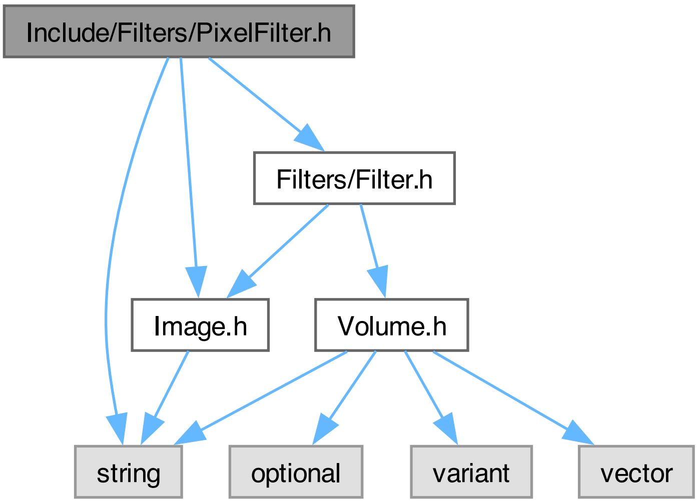
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDINGTYPE_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDING_H

#include "Image.h"

#include <vector>

enum class PaddingType {
    ZeroPadding, // Zero padding
    EdgeReplication, // Edge replication
    ReflectPadding // Reflect padding
};

class Padding {
public:
    static std::vector<unsigned char>
    getPixelWindow(const Image &image, int x, int y, int c, int kernelSize, PaddingType paddingType);
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PADDINGTYPE_H
```

9.36 Include/Filters/PixelFilter.h File Reference

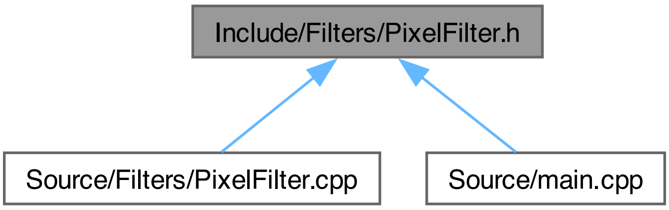
Defines the PixelFilter class for pixel-level image processing operations.

```
#include <Image.h>
#include <Filters/Filter.h>
#include <string>
```

Include dependency graph for PixelFilter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PixelFilter

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H

9.36.1 Detailed Description

Defines the PixelFilter class for pixel-level image processing operations.

PixelFilter, implementing the IFilter2D interface, offers a variety of pixel-level manipulations for 2D images, including grayscale conversion, brightness adjustment, histogram equalization, thresholding, and noise addition. It supports operations in different color spaces (HSL, HSV) and allows for configurable processing parameters such as brightness level, threshold for pixelation, and noise percentage. This class facilitates detailed image editing and enhancement tasks, making it a versatile tool in the Advanced Programming Group's image processing toolkit. Whether for pre-processing steps or standalone adjustments, PixelFilter provides robust solutions for refining image aesthetics and quality.

Date

Created on March 20, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.36.2 Macro Definition Documentation

9.36.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H
```

9.37 PixelFilter.h

Go to the documentation of this file.

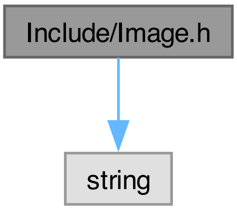
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H

#include <Image.h>
#include <Filters/Filter.h>

#include <string>

class PixelFilter : public IFilter2D {
private:
    std::string filterType; // "grayscale", "brightness", "histogram", "threshold", "noise"
    int brightness = 128; // Default brightness
    std::string space; // "HSL" or "HSV"
    int threshold = 0; // Threshold value
    double percentage = 0; // Percentage of noise

    void convertToGrayscale(Image &image);

    void adjustBrightness(Image &image);

    void equalizeHistogram(Image &image);

    void thresholdPixel(Image &image);

    void addNoise(Image &image);

    void calculateHistogramAndCDF(const std::vector<float> &channel, std::vector<int> &histogram,
std::vector<int> &cdf);

    void applyCDF(std::vector<float> &channel, const std::vector<int> &cdf);

    void RGBtoHSV(float r, float g, float b, float &h, float &s, float &v);

    void HSVtoRGB(float h, float s, float v, float &r, float &g, float &b);

    void RGBtoHSL(float r, float g, float b, float &h, float &s, float &l);

    void HSLtoRGB(float h, float s, float l, float &r, float &g, float &b);

public:
    PixelFilter(const std::string &type, const std::optional<int> &brightness = std::nullopt,
                const std::string &space = "", int threshold = 0, double percentage = 0);

    void apply(Image &image) override;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PIXELFILTER_H
```

9.38 Include/Image.h File Reference

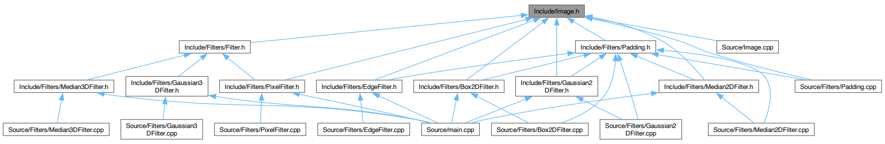
Provides an Image class for basic image processing operations.

```
#include <string>
```

Include dependency graph for Image.h:



This graph shows which files directly or indirectly include this file:


Include/Image.h
Include/Filters/Filter.h
Include/Filters/Padding.h
Source/Image.cpp
Include/Filters/Median3DFilter.h
Include/Filters/Gaussian3 DFilter.h
Include/Filters/PixelFilter.h
Include/Filters/EdgeFilter.h
Include/Filters/Box2DFilter.h
Include/Filters/Gaussian2 DFilter.h
Include/Filters/Median2DFilter.h
Source/Filters/Padding.cpp
Source/Filters/Median3DFilter.cpp
Source/Filters/Gaussian3 DFilter.cpp
Source/Filters/PixelFilter.cpp
Source/Filters/EdgeFilter.cpp
Source/main.cpp
Source/Filters/Box2DFilter.cpp
Source/Filters/Gaussian2 DFilter.cpp
Source/Filters/Median2DFilter.cpp

Classes

- class Image

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H

9.38.1 Detailed Description

Provides an Image class for basic image processing operations.

The Image class supports basic image processing functionalities including loading, saving, and converting images to grayscale. It is designed with a focus on handling image data efficiently while providing a simple interface for image manipulation. This class handles images as arrays of unsigned char, representing pixel data, and provides methods to load images from files, save images to files, and convert color images to grayscale. The class also explicitly bans copy construction and assignment to avoid unintentional deep copies of image data. It is a part of the utility toolkit developed by the Advanced Programming Group.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.38.2 Macro Definition Documentation

9.38.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H
```

9.39 Image.h

Go to the documentation of this file.

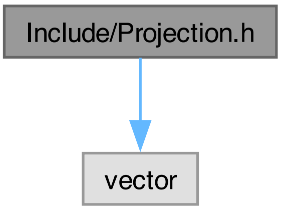
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H

#include <string>

class Image {
private:
    int width, height, channels; // Size of the image
    unsigned char *data; // Image data

    Image(const Image &) = delete;

    Image &operator=(const Image &) = delete;

public:
    Image();

    Image(int width, int height, int channels, unsigned char *data);

    ~Image();

    int getWidth() const;

    int getHeight() const;

    int getChannels() const;

    unsigned char *getData() const;

    void updateData(unsigned char *data);

    void setWidth(int width);

    void setHeight(int height);

    void setChannels(int channels);

    bool loadFromFile(const std::string &path);

    bool saveToFile(const std::string &path) const;

};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_IMAGE_H
```

9.40 Include/Projection.h File Reference

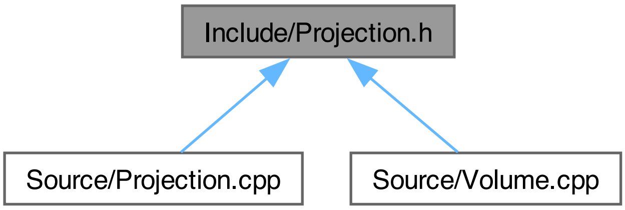
Provides projection techniques for 3D data visualization.

```
#include <vector>
```

Include dependency graph for Projection.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Projection

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H

9.40.1 Detailed Description

Provides projection techniques for 3D data visualization.

The Projection class encompasses methods to generate various types of projections from 3D data, facilitating the visualization of volumetric data in a 2D format. It supports maximum, minimum, average, and median intensity projections, each offering a different perspective on the 3D data's structure and composition. These projection methods are crucial for analyzing and interpreting volumetric datasets in fields such as medical imaging, scientific visualization, and computer graphics. Implementing efficient algorithms for these projections, the class aims to deliver high-performance tools as part of the visualization toolkit developed by the Advanced Programming Group.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.40.2 Macro Definition Documentation

9.40.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H
```

9.41 Projection.h

Go to the documentation of this file.

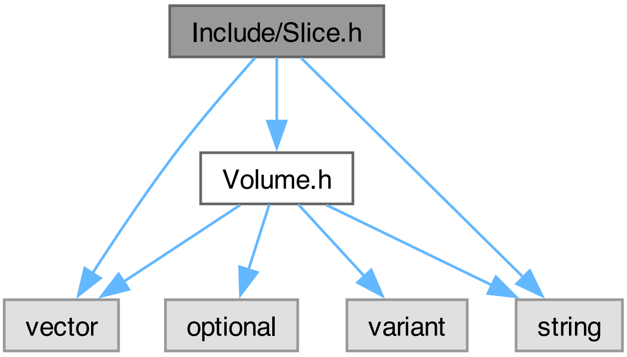
```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H

#include <vector>

class Projection {
public:
    static std::vector<unsigned char>
    maximumIntensityProjection(int width, int height, int depth, const unsigned char *data);

    static std::vector<unsigned char>
    minimumIntensityProjection(int width, int height, int depth, const unsigned char *data);

    static std::vector<unsigned char>
    averageIntensityProjection(int width, int height, int depth, const unsigned char *data);

    static std::vector<unsigned char>
    medianIntensityProjection(int width, int height, int depth, const unsigned char *data);
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_PROJECTION_H
```

9.42 Include/Slice.h File Reference

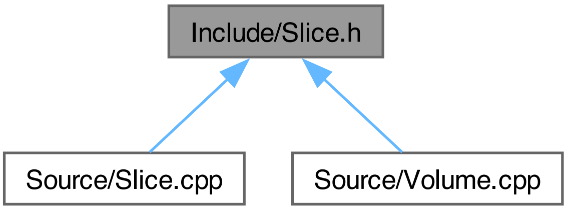
Facilitates extraction of 2D slices from 3D volume data.

```
#include "Volume.h"
#include <vector>
#include <string>
```

Include dependency graph for Slice.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Slice

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H

9.42.1 Detailed Description

Facilitates extraction of 2D slices from 3D volume data.

The Slice class is specifically designed to extract 2D slices from 3D volumetric data along specified planes (axial, coronal, or sagittal). This functionality is critical in medical imaging, scientific research, and 3D data analysis, where examining specific cross-sections of a dataset can provide insightful information about its internal structure. By offering a method to obtain plane slices at given indices, the class enables detailed examination and manipulation of 3D volume data. The Slice class is an integral part of the data processing toolkit developed by the Advanced Programming Group, focusing on high efficiency and precision in handling volumetric data.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.42.2 Macro Definition Documentation

9.42.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H
```

9.43 Slice.h

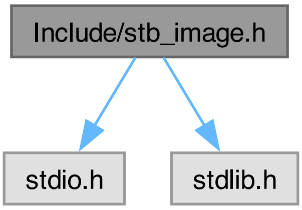
Go to the documentation of this file.

```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H

#include "Volume.h"

#include <vector>
#include <string>

class Slice {
public:
    static std::vector<unsigned char>
    getPlaneSlice(int width, int height, int depth, const unsigned char *data, const std::string &plane,
                  int sliceIndex);

private:
    Slice() = delete;

    ~Slice() = delete;
};

#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_SLICE_H
```

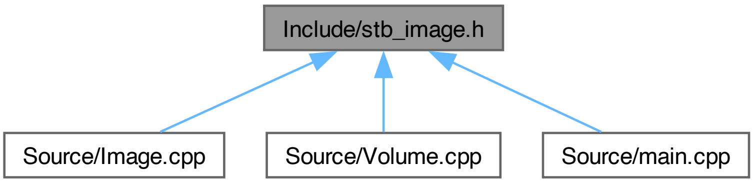
9.44 Include/stb_image.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for stb_image.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct stbi_io_callbacks

Macros

- #define STBI_VERSION 1
- #define STBIDEF extern

Typedefs

- typedef unsigned char stbi_uc
- typedef unsigned short stbi_us

Enumerations

- enum {
STBI_default = 0 , STBI_grey = 1 , STBI_grey_alpha = 2 , STBI_rgb = 3 ,
STBI_rgb_alpha = 4 }

Functions

- STBIDEF stbi_uc ∗ stbi_load_from_memory (stbi_uc const ∗buffer, int len, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_uc ∗ stbi_load_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_uc ∗ stbi_load (char const ∗filename, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_uc ∗ stbi_load_from_file (FILE ∗f, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_uc ∗ stbi_load_gif_from_memory (stbi_uc const ∗buffer, int len, int ∗∗delays, int ∗x, int ∗y, int ∗z, int ∗comp, int req_comp)
- STBIDEF stbi_us ∗ stbi_load_16_from_memory (stbi_uc const ∗buffer, int len, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_us ∗ stbi_load_16_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF stbi_us ∗ stbi_load_16 (char const ∗filename, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)

- STBIDEF stbi_us ∗ stbi_load_from_file_16 (FILE ∗f, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF float ∗ stbi_loadf_from_memory (stbi_uc const ∗buffer, int len, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF float ∗ stbi_loadf_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF float ∗ stbi_loadf (char const ∗filename, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF float ∗ stbi_loadf_from_file (FILE ∗f, int ∗x, int ∗y, int ∗channels_in_file, int desired_channels)
- STBIDEF void stbi_hdr_to_ldr_gamma (float gamma)
- STBIDEF void stbi_hdr_to_ldr_scale (float scale)
- STBIDEF void stbi_ldr_to_hdr_gamma (float gamma)
- STBIDEF void stbi_ldr_to_hdr_scale (float scale)
- STBIDEF int stbi_is_hdr_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user)
- STBIDEF int stbi_is_hdr_from_memory (stbi_uc const ∗buffer, int len)
- STBIDEF int stbi_is_hdr (char const ∗filename)
- STBIDEF int stbi_is_hdr_from_file (FILE ∗f)
- STBIDEF const char ∗ stbi_failure_reason (void)
- STBIDEF void stbi_image_free (void ∗retval_from_stbi_load)
- STBIDEF int stbi_info_from_memory (stbi_uc const ∗buffer, int len, int ∗x, int ∗y, int ∗comp)
- STBIDEF int stbi_info_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user, int ∗x, int ∗y, int ∗comp)
- STBIDEF int stbi_is_16_bit_from_memory (stbi_uc const ∗buffer, int len)
- STBIDEF int stbi_is_16_bit_from_callbacks (stbi_io_callbacks const ∗clbk, void ∗user)
- STBIDEF int stbi_info (char const ∗filename, int ∗x, int ∗y, int ∗comp)
- STBIDEF int stbi_info_from_file (FILE ∗f, int ∗x, int ∗y, int ∗comp)
- STBIDEF int stbi_is_16_bit (char const ∗filename)
- STBIDEF int stbi_is_16_bit_from_file (FILE ∗f)
- STBIDEF void stbi_set_unpremultiply_on_load (int flag_true_if_should_unpremultiply)
- STBIDEF void stbi_convert_iphone_png_to_rgb (int flag_true_if_should_convert)
- STBIDEF void stbi_set_flip_vertically_on_load (int flag_true_if_should_flip)
- STBIDEF void stbi_set_unpremultiply_on_load_thread (int flag_true_if_should_unpremultiply)
- STBIDEF void stbi_convert_iphone_png_to_rgb_thread (int flag_true_if_should_convert)
- STBIDEF void stbi_set_flip_vertically_on_load_thread (int flag_true_if_should_flip)
- STBIDEF char ∗ stbi_zlib_decode_malloc_guesssize (const char ∗buffer, int len, int initial_size, int ∗outlen)
- STBIDEF char ∗ stbi_zlib_decode_malloc_guesssize_headerflag (const char ∗buffer, int len, int initial_size, int ∗outlen, int parse_header)
- STBIDEF char ∗ stbi_zlib_decode_malloc (const char ∗buffer, int len, int ∗outlen)
- STBIDEF int stbi_zlib_decode_buffer (char ∗obuffer, int olen, const char ∗ibuffer, int ilen)
- STBIDEF char ∗ stbi_zlib_decode_noheader_malloc (const char ∗buffer, int len, int ∗outlen)
- STBIDEF int stbi_zlib_decode_noheader_buffer (char ∗obuffer, int olen, const char ∗ibuffer, int ilen)

9.44.1 Macro Definition Documentation

9.44.1.1 STBI_VERSION

```
#define STBI_VERSION 1
```

9.44.1.2 STBIDEF

```
#define STBIDEF extern
```

9.44.2 Typedef Documentation

9.44.2.1 stbi_uc

```
typedef unsigned char stbi_uc
```

9.44.2.2 stbi_us

```
typedef unsigned short stbi_us
```

9.44.3 Enumeration Type Documentation

9.44.3.1 anonymous enum

```
anonymous enum
```

Enumerator

STBI_default	
STBI_grey	
STBI_grey_alpha	
STBI_rgb	
STBI_rgb_alpha	

9.44.4 Function Documentation

9.44.4.1 stbi_convert_iphone_png_to_rgb()

```
STBIDEF void stbi_convert_iphone_png_to_rgb (
             int flag_true_if_should_convert )
```

9.44.4.2 stbi_convert_iphone_png_to_rgb_thread()

```
STBIDEF void stbi_convert_iphone_png_to_rgb_thread (
             int flag_true_if_should_convert )
```

9.44.4.3 stbi_failure_reason()

```
STBIDEF const char * stbi_failure_reason (
             void )
```

9.44.4.4 stbi_hdr_to_ldr_gamma()

```
STBIDEF void stbi_hdr_to_ldr_gamma (
             float gamma )
```

9.44.4.5 stbi_hdr_to_ldr_scale()

```
STBIDEF void stbi_hdr_to_ldr_scale (
        float scale )
```

9.44.4.6 stbi_image_free()

```
STBIDEF void stbi_image_free (
        void * retval_from_stbi_load )
```

9.44.4.7 stbi_info()

```
STBIDEF int stbi_info (
        char const * filename,
        int * x,
        int * y,
        int * comp )
```

9.44.4.8 stbi_info_from_callbacks()

```
STBIDEF int stbi_info_from_callbacks (
        stbi_io_callbacks const * clbk,
        void * user,
        int * x,
        int * y,
        int * comp )
```

9.44.4.9 stbi_info_from_file()

```
STBIDEF int stbi_info_from_file (
        FILE * f,
        int * x,
        int * y,
        int * comp )
```

9.44.4.10 stbi_info_from_memory()

```
STBIDEF int stbi_info_from_memory (
        stbi_uc const * buffer,
        int len,
        int * x,
        int * y,
        int * comp )
```

9.44.4.11 stbi_is_16_bit()

```
STBIDEF int stbi_is_16_bit (
        char const * filename )
```

9.44.4.12 stbi_is_16_bit_from_callbacks()

```
STBIDEF int stbi_is_16_bit_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user )
```

9.44.4.13 stbi_is_16_bit_from_file()

```
STBIDEF int stbi_is_16_bit_from_file (
            FILE * f )
```

9.44.4.14 stbi_is_16_bit_from_memory()

```
STBIDEF int stbi_is_16_bit_from_memory (
            stbi_uc const * buffer,
            int len )
```

9.44.4.15 stbi_is_hdr()

```
STBIDEF int stbi_is_hdr (
            char const * filename )
```

9.44.4.16 stbi_is_hdr_from_callbacks()

```
STBIDEF int stbi_is_hdr_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user )
```

9.44.4.17 stbi_is_hdr_from_file()

```
STBIDEF int stbi_is_hdr_from_file (
            FILE * f )
```

9.44.4.18 stbi_is_hdr_from_memory()

```
STBIDEF int stbi_is_hdr_from_memory (
            stbi_uc const * buffer,
            int len )
```

9.44.4.19 stbi_ldr_to_hdr_gamma()

```
STBIDEF void stbi_ldr_to_hdr_gamma (
            float gamma )
```

9.44.4.20 stbi_ldr_to_hdr_scale()

```
STBIDEF void stbi_ldr_to_hdr_scale (
              float scale )
```

9.44.4.21 stbi_load()

```
STBIDEF stbi_uc * stbi_load (
              char const * filename,
              int * x,
              int * y,
              int * channels_in_file,
              int desired_channels )
```

9.44.4.22 stbi_load_16()

```
STBIDEF stbi_us * stbi_load_16 (
              char const * filename,
              int * x,
              int * y,
              int * channels_in_file,
              int desired_channels )
```

9.44.4.23 stbi_load_16_from_callbacks()

```
STBIDEF stbi_us * stbi_load_16_from_callbacks (
              stbi_io_callbacks const * clbk,
              void * user,
              int * x,
              int * y,
              int * channels_in_file,
              int desired_channels )
```

9.44.4.24 stbi_load_16_from_memory()

```
STBIDEF stbi_us * stbi_load_16_from_memory (
              stbi_uc const * buffer,
              int len,
              int * x,
              int * y,
              int * channels_in_file,
              int desired_channels )
```

9.44.4.25 stbi_load_from_callbacks()

```
STBIDEF stbi_uc * stbi_load_from_callbacks (
              stbi_io_callbacks const * clbk,
              void * user,
              int * x,
              int * y,
              int * channels_in_file,
              int desired_channels )
```

9.44.4.26 stbi_load_from_file()

```
STBIDEF stbi_uc * stbi_load_from_file (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.27 stbi_load_from_file_16()

```
STBIDEF stbi_us * stbi_load_from_file_16 (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.28 stbi_load_from_memory()

```
STBIDEF stbi_uc * stbi_load_from_memory (
            stbi_uc const * buffer,
            int len,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.29 stbi_load_gif_from_memory()

```
STBIDEF stbi_uc * stbi_load_gif_from_memory (
            stbi_uc const * buffer,
            int len,
            int ** delays,
            int * x,
            int * y,
            int * z,
            int * comp,
            int req_comp )
```

9.44.4.30 stbi_loadf()

```
STBIDEF float * stbi_loadf (
            char const * filename,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.31 stbi_loadf_from_callbacks()

```
STBIDEF float * stbi_loadf_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.32 stbi_loadf_from_file()

```
STBIDEF float * stbi_loadf_from_file (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.33 stbi_loadf_from_memory()

```
STBIDEF float * stbi_loadf_from_memory (
            stbi_uc const * buffer,
            int len,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

9.44.4.34 stbi_set_flip_vertically_on_load()

```
STBIDEF void stbi_set_flip_vertically_on_load (
            int flag_true_if_should_flip )
```

9.44.4.35 stbi_set_flip_vertically_on_load_thread()

```
STBIDEF void stbi_set_flip_vertically_on_load_thread (
            int flag_true_if_should_flip )
```

9.44.4.36 stbi_set_unpremultiply_on_load()

```
STBIDEF void stbi_set_unpremultiply_on_load (
            int flag_true_if_should_unpremultiply )
```

9.44.4.37 stbi_set_unpremultiply_on_load_thread()

```
STBIDEF void stbi_set_unpremultiply_on_load_thread (
            int flag_true_if_should_unpremultiply )
```

9.44.4.38 stbi_zlib_decode_buffer()

```
STBIDEF int stbi_zlib_decode_buffer (
            char * obuffer,
            int olen,
            const char * ibuffer,
            int ilen )
```

9.44.4.39 stbi_zlib_decode_malloc()

```
STBIDEF char * stbi_zlib_decode_malloc (
            const char * buffer,
            int len,
            int * outlen )
```

9.44.4.40 stbi_zlib_decode_malloc_guesssize()

```
STBIDEF char * stbi_zlib_decode_malloc_guesssize (
            const char * buffer,
            int len,
            int initial_size,
            int * outlen )
```

9.44.4.41 stbi_zlib_decode_malloc_guesssize_headerflag()

```
STBIDEF char * stbi_zlib_decode_malloc_guesssize_headerflag (
            const char * buffer,
            int len,
            int initial_size,
            int * outlen,
            int parse_header )
```

9.44.4.42 stbi_zlib_decode_noheader_buffer()

```
STBIDEF int stbi_zlib_decode_noheader_buffer (
            char * obuffer,
            int olen,
            const char * ibuffer,
            int ilen )
```

9.44.4.43 stbi_zlib_decode_noheader_malloc()

```
STBIDEF char * stbi_zlib_decode_noheader_malloc (
            const char * buffer,
            int len,
            int * outlen )
```

9.45 stb_image.h

Go to the documentation of this file.

```
/* stb_image - v2.28 - public domain image loader - http://nothings.org/stb
                                  no warranty implied; use at your own risk

   Do this:
      #define STB_IMAGE_IMPLEMENTATION
   before you include this file in *one* C or C++ file to create the implementation.

   // i.e. it should look like this:
   #include ...
   #include ...
   #include ...
   #define STB_IMAGE_IMPLEMENTATION
   #include "stb_image.h"

   You can #define STBI_ASSERT(x) before the #include to avoid using assert.h.
   And #define STBI_MALLOC, STBI_REALLOC, and STBI_FREE to avoid using malloc,realloc,free


   QUICK NOTES:
      Primarily of interest to game developers and other people who can
          avoid problematic images and only need the trivial interface

      JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
      PNG 1/2/4/8/16-bit-per-channel

      TGA (not sure what subset, if a subset)
      BMP non-1bpp, non-RLE
      PSD (composited view only, no extra channels, 8/16 bit-per-channel)

      GIF (*comp always reports as 4-channel)
      HDR (radiance rgbE format)
      PIC (Softimage PIC)
      PNM (PPM and PGM binary only)

      Animated GIF still needs a proper API, but here's one way to do it:
          http://gist.github.com/urraka/685d9a6340b26b830d49

      - decode from memory or through FILE (define STBI_NO_STDIO to remove code)
      - decode from arbitrary I/O callbacks
      - SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)

   Full documentation under "DOCUMENTATION" below.


LICENSE

  See end of file for license information.

RECENT REVISION HISTORY:

      2.28  (2023-01-29) many error fixes, security errors, just tons of stuff
      2.27  (2021-07-11) document stbi_info better, 16-bit PNM support, bug fixes
      2.26  (2020-07-13) many minor fixes
      2.25  (2020-02-02) fix warnings
      2.24  (2020-02-02) fix warnings; thread-local failure_reason and flip_vertically
      2.23  (2019-08-11) fix clang static analysis warning
      2.22  (2019-03-04) gif fixes, fix warnings
      2.21  (2019-02-25) fix typo in comment
      2.20  (2019-02-07) support utf8 filenames in Windows; fix warnings and platform ifdefs
      2.19  (2018-02-11) fix warning
      2.18  (2018-01-30) fix warnings
      2.17  (2018-01-29) bugfix, 1-bit BMP, 16-bitness query, fix warnings
      2.16  (2017-07-23) all functions have 16-bit variants; optimizations; bugfixes
      2.15  (2017-03-18) fix png-1,2,4; all Imagenet JPGs; no runtime SSE detection on GCC
      2.14  (2017-03-03) remove deprecated STBI_JPEG_OLD; fixes for Imagenet JPGs
      2.13  (2016-12-04) experimental 16-bit API, only for PNG so far; fixes
      2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
      2.11  (2016-04-02) 16-bit PNGS; enable SSE2 in non-gcc x64
                         RGB-format JPEG; remove white matting in PSD;
                         allocate large structures on the stack;
                         correct channel count for PNG & BMP
      2.10  (2016-01-22) avoid warning introduced in 2.09
      2.09  (2016-01-16) 16-bit TGA; comments in PNM files; STBI_REALLOC_SIZED

   See end of file for full revision history.


 ============================    Contributors    =========================

 Image formats                          Extensions, features
    Sean Barrett (jpeg, png, bmp)          Jetro Lauha (stbi_info)
    Nicolas Schulz (hdr, psd)              Martin "SpartanJ" Golini (stbi_info)
    Jonathan Dummer (tga)                  James "moose2000" Brown (iPhone PNG)
    Jean-Marc Lienher (gif)                Ben "Disch" Wenger (io callbacks)
    Tom Seddon (pic)                       Omar Cornut (1/2/4-bit PNG)
    Thatcher Ulrich (psd)                  Nicolas Guillemot (vertical flip)
    Ken Miller (pgm, ppm)                  Richard Mitton (16-bit PSD)
    github:urraka (animated gif)           Junggon Kim (PNM comments)
    Christopher Forseth (animated gif)     Daniel Gibson (16-bit TGA)
                                           socks-the-fox (16-bit PNG)
                                           Jeremy Sawicki (handle all ImageNet JPGs)
 Optimizations & bugfixes                  Mikhail Morozov (1-bit BMP)
    Fabian "ryg" Giesen                    Anael Seghezzi (is-16-bit query)
    Arseny Kapoulkine                      Simon Breuss (16-bit PNM)
    John-Mark Allen
    Carmelo J Fdez-Aguera

 Bug & warning fixes
    Marc LeBlanc            David Woo          Guillaume George     Martins Mozeiko
    Christpher Lloyd        Jerry Jansson      Joseph Thomson       Blazej Dariusz Roszkowski
    Phil Jordan                                Dave Moore           Roy Eltham
    Hayaki Saito            Nathan Reed        Won Chun
    Luke Graham             Johan Duparc       Nick Verigakis       the Horde3D community
    Thomas Ruf              Ronny Chevalier                         github:rlyeh
    Janez Zemva             John Bartholomew   Michal Cichon        github:romigrou
    Jonathan Blow           Ken Hamada         Tero Hanninen        github:svdijk
    Eugene Golushkov        Laurent Gomila     Cort Stratton        github:snagar
    Aruelien Pocheville     Sergio Gonzalez    Thibault Reuille     github:Zelex
    Cass Everitt            Ryamond Barbiero                        github:grim210
    Paul Du Bois            Engin Manap        Aldo Culquicondor    github:sammyhw
    Philipp Wiesemann       Dale Weiler        Oriol Ferrer Mesia   github:phprus
    Josh Tobin              Neil Bickford      Matthew Gregan       github:poppolopoppo
    Julian Raschke          Gregory Mullen     Christian Floisand   github:darealshinji
    Baldur Karlsson         Kevin Schmidt      JR Smith             github:Michaelangel007
                            Brad Weinberger    Matvey Cherevko      github:mosra
    Luca Sas                Alexander Veselov  Zack Middleton       [reserved]
    Ryan C. Gordon          [reserved]                              [reserved]
                     DO NOT ADD YOUR NAME HERE

                     Jacko Dirks

  To add your name to the credits, pick a random blank space in the middle and fill it.
  80% of merge conflicts on stb PRs are due to people adding their name at the end
  of the credits.
*/

#ifndef STBI_INCLUDE_STB_IMAGE_H
#define STBI_INCLUDE_STB_IMAGE_H

// DOCUMENTATION
//
// Limitations:
//    - no 12-bit-per-channel JPEG
//    - no JPEGs with arithmetic coding
//    - GIF always returns *comp=4
//
// Basic usage (see HDR discussion below for HDR usage):
//    int x,y,n;
//    unsigned char *data = stbi_load(filename, &x, &y, &n, 0);
//    // ... process data if not NULL ...
//    // ... x = width, y = height, n = # 8-bit components per pixel ...
//    // ... replace '0' with '1'..'4' to force that many components per pixel
//    // ... but 'n' will always be the number that it would have been if you said 0
//    stbi_image_free(data);
//
// Standard parameters:
//    int *x                 -- outputs image width in pixels
//    int *y                 -- outputs image height in pixels
//    int *channels_in_file  -- outputs # of image components in image file
//    int desired_channels   -- if non-zero, # of image components requested in result
//
// The return value from an image loader is an 'unsigned char *' which points
// to the pixel data, or NULL on an allocation failure or if the image is
// corrupt or invalid. The pixel data consists of *y scanlines of *x pixels,
// with each pixel consisting of N interleaved 8-bit components; the first
// pixel pointed to is top-left-most in the image. There is no padding between
// image scanlines or between pixels, regardless of format. The number of
// components N is 'desired_channels' if desired_channels is non-zero, or
// *channels_in_file otherwise. If desired_channels is non-zero,
// *channels_in_file has the number of components that _would_ have been
// output otherwise. E.g. if you set desired_channels to 4, you will always
// get RGBA output, but you can check *channels_in_file to see if it's trivially
// opaque because e.g. there were only 3 channels in the source image.
//
// An output image with N components has the following components interleaved
// in this order in each pixel:
//
//     N=#comp     components
//       1           grey
//       2           grey, alpha
//       3           red, green, blue
//       4           red, green, blue, alpha
//
// If image loading fails for any reason, the return value will be NULL,
// and *x, *y, *channels_in_file will be unchanged. The function
// stbi_failure_reason() can be queried for an extremely brief, end-user
// unfriendly explanation of why the load failed. Define STBI_NO_FAILURE_STRINGS
// to avoid compiling these strings at all, and STBI_FAILURE_USERMSG to get slightly
// more user-friendly ones.
//
// Paletted PNG, BMP, GIF, and PIC images are automatically depalettized.
//
// To query the width, height and component count of an image without having to
// decode the full file, you can use the stbi_info family of functions:
//
//   int x,y,n,ok;
//   ok = stbi_info(filename, &x, &y, &n);
//   // returns ok=1 and sets x, y, n if image is a supported format,
//   // 0 otherwise.
//
// Note that stb_image pervasively uses ints in its public API for sizes,
// including sizes of memory buffers. This is now part of the API and thus
// hard to change without causing breakage. As a result, the various image
// loaders all have certain limits on image size; these differ somewhat
// by format but generally boil down to either just under 2GB or just under
// 1GB. When the decoded image would be larger than this, stb_image decoding
// will fail.
//
// Additionally, stb_image will reject image files that have any of their
// dimensions set to a larger value than the configurable STBI_MAX_DIMENSIONS,
// which defaults to 2**24 = 16777216 pixels. Due to the above memory limit,
// the only way to have an image with such dimensions load correctly
// is for it to have a rather extreme aspect ratio. Either way, the
// assumption here is that such larger images are likely to be malformed
// or malicious. If you do need to load an image with individual dimensions
// larger than that, and it still fits in the overall size limit, you can
// #define STBI_MAX_DIMENSIONS on your own to be something larger.
//
// ===========================================================================
//
// UNICODE:
//
//   If compiling for Windows and you wish to use Unicode filenames, compile
//   with
//       #define STBI_WINDOWS_UTF8
//   and pass utf8-encoded filenames. Call stbi_convert_wchar_to_utf8 to convert
//   Windows wchar_t filenames to utf8.
//
// ===========================================================================
//
// Philosophy
//
// stb libraries are designed with the following priorities:
//
//    1. easy to use
//    2. easy to maintain
//    3. good performance
//
// Sometimes I let "good performance" creep up in priority over "easy to maintain",
// and for best performance I may provide less-easy-to-use APIs that give higher
// performance, in addition to the easy-to-use ones. Nevertheless, it's important
// to keep in mind that from the standpoint of you, a client of this library,
// all you care about is #1 and #3, and stb libraries DO NOT emphasize #3 above all.
//
// Some secondary priorities arise directly from the first two, some of which
// provide more explicit reasons why performance can't be emphasized.
//
//    - Portable ("ease of use")
//    - Small source code footprint ("easy to maintain")
//    - No dependencies ("ease of use")
//
// ===========================================================================
//
// I/O callbacks
//
// I/O callbacks allow you to read from arbitrary sources, like packaged
// files or some other source. Data read from callbacks are processed
// through a small internal buffer (currently 128 bytes) to try to reduce
// overhead.
//
// The three functions you must define are "read" (reads some bytes of data),
// "skip" (skips some bytes of data), "eof" (reports if the stream is at the end).
//
// ===========================================================================
//
// SIMD support
```

```
//
// The JPEG decoder will try to automatically use SIMD kernels on x86 when
// supported by the compiler. For ARM Neon support, you must explicitly
// request it.
//
// (The old do-it-yourself SIMD API is no longer supported in the current
// code.)
//
// On x86, SSE2 will automatically be used when available based on a run-time
// test; if not, the generic C versions are used as a fall-back. On ARM targets,
// the typical path is to have separate builds for NEON and non-NEON devices
// (at least this is true for iOS and Android). Therefore, the NEON support is
// toggled by a build flag: define STBI_NEON to get NEON loops.
//
// If for some reason you do not want to use any of SIMD code, or if
// you have issues compiling it, you can disable it entirely by
// defining STBI_NO_SIMD.
//
// ==========================================================================
//
// HDR image support   (disable by defining STBI_NO_HDR)
//
// stb_image supports loading HDR images in general, and currently the Radiance
// .HDR file format specifically. You can still load any file through the existing
// interface; if you attempt to load an HDR file, it will be automatically remapped
// to LDR, assuming gamma 2.2 and an arbitrary scale factor defaulting to 1;
// both of these constants can be reconfigured through this interface:
//
//     stbi_hdr_to_ldr_gamma(2.2f);
//     stbi_hdr_to_ldr_scale(1.0f);
//
// (note, do not use _inverse_ constants; stbi_image will invert them
// appropriately).
//
// Additionally, there is a new, parallel interface for loading files as
// (linear) floats to preserve the full dynamic range:
//
//    float *data = stbi_loadf(filename, &x, &y, &n, 0);
//
// If you load LDR images through this interface, those images will
// be promoted to floating point values, run through the inverse of
// constants corresponding to the above:
//
//     stbi_ldr_to_hdr_scale(1.0f);
//     stbi_ldr_to_hdr_gamma(2.2f);
//
// Finally, given a filename (or an open file or memory block--see header
// file for details) containing image data, you can query for the "most
// appropriate" interface to use (that is, whether the image is HDR or
// not), using:
//
//     stbi_is_hdr(char *filename);
//
// ==========================================================================
//
// iPhone PNG support:
//
// We optionally support converting iPhone-formatted PNGs (which store
// premultiplied BGRA) back to RGB, even though they're internally encoded
// differently. To enable this conversion, call
// stbi_convert_iphone_png_to_rgb(1).
//
// Call stbi_set_unpremultiply_on_load(1) as well to force a divide per
// pixel to remove any premultiplied alpha *only* if the image file explicitly
// says there's premultiplied data (currently only happens in iPhone images,
// and only if iPhone convert-to-rgb processing is on).
//
// ==========================================================================
//
// ADDITIONAL CONFIGURATION
//
//  - You can suppress implementation of any of the decoders to reduce
//    your code footprint by #defining one or more of the following
//    symbols before creating the implementation.
//
//        STBI_NO_JPEG
//        STBI_NO_PNG
//        STBI_NO_BMP
//        STBI_NO_PSD
//        STBI_NO_TGA
//        STBI_NO_GIF
//        STBI_NO_HDR
//        STBI_NO_PIC
//        STBI_NO_PNM   (.ppm and .pgm)
//
//  - You can request *only* certain decoders and suppress all other ones
//    (this will be more forward-compatible, as addition of new decoders
//    doesn’t require you to disable them explicitly):
//
//        STBI_ONLY_JPEG
//        STBI_ONLY_PNG
//        STBI_ONLY_BMP
//        STBI_ONLY_PSD
//        STBI_ONLY_TGA
//        STBI_ONLY_GIF
//        STBI_ONLY_HDR
//        STBI_ONLY_PIC
//        STBI_ONLY_PNM   (.ppm and .pgm)
//
//   - If you use STBI_NO_PNG (or _ONLY_ without PNG), and you still
//     want the zlib decoder to be available, #define STBI_SUPPORT_ZLIB
//
//  - If you define STBI_MAX_DIMENSIONS, stb_image will reject images greater
//    than that size (in either width or height) without further processing.
//    This is to let programs in the wild set an upper bound to prevent
//    denial-of-service attacks on untrusted data, as one could generate a
//    valid image of gigantic dimensions and force stb_image to allocate a
//    huge block of memory and spend disproportionate time decoding it. By
//    default this is set to (1 « 24), which is 16777216, but that’s still
//    very big.

#ifndef STBI_NO_STDIO
#include <stdio.h>
#endif // STBI_NO_STDIO

#define STBI_VERSION 1

enum
{
   STBI_default = 0, // only used for desired_channels

   STBI_grey       = 1,
   STBI_grey_alpha = 2,
   STBI_rgb        = 3,
   STBI_rgb_alpha  = 4
};

#include <stdlib.h>
typedef unsigned char stbi_uc;
typedef unsigned short stbi_us;

#ifdef __cplusplus
extern "C" {
#endif

#ifndef STBIDEF
#ifdef STB_IMAGE_STATIC
#define STBIDEF static
#else
#define STBIDEF extern
#endif
#endif

//
// PRIMARY API - works on images of any type
//

//
// load image by filename, open file, or memory buffer
//

typedef struct
{
   int      (*read)  (void *user,char *data,int size);   // fill ’data’ with ’size’ bytes.  return number of bytes actually read
   void     (*skip)  (void *user,int n);                 // skip the next ’n’ bytes, or ’unget’ the last -n bytes if negative
   int      (*eof)   (void *user);                       // returns nonzero if we are at end of file/data
} stbi_io_callbacks;

//
// 8-bits-per-channel interface
//

STBIDEF stbi_uc *stbi_load_from_memory   (stbi_uc           const *buffer, int len   , int *x, int *y, int *channels_in_file, int desired_channels);
STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk  , void *user, int *x, int *y, int *channels_in_file, int desired_channels);

#ifndef STBI_NO_STDIO
STBIDEF stbi_uc *stbi_load            (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
STBIDEF stbi_uc *stbi_load_from_file  (FILE *f, int *x, int *y, int *channels_in_file, int
desired_channels);
// for stbi_load_from_file, file pointer is left pointing immediately after image
#endif

#ifndef STBI_NO_GIF
STBIDEF stbi_uc *stbi_load_gif_from_memory(stbi_uc const *buffer, int len, int **delays, int *x, int *y, int *z, int *comp, int req_comp);
#endif

#ifdef STBI_WINDOWS_UTF8
STBIDEF int stbi_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input);
#endif

//
// 16-bits-per-channel interface
//

STBIDEF stbi_us *stbi_load_16_from_memory   (stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels);
STBIDEF stbi_us *stbi_load_16_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels);

#ifndef STBI_NO_STDIO
STBIDEF stbi_us *stbi_load_16          (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
STBIDEF stbi_us *stbi_load_from_file_16(FILE *f, int *x, int *y, int *channels_in_file, int desired_channels);
#endif

//
// float-per-channel interface
//
#ifndef STBI_NO_LINEAR
   STBIDEF float *stbi_loadf_from_memory     (stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels);
   STBIDEF float *stbi_loadf_from_callbacks  (stbi_io_callbacks const *clbk, void *user, int *x, int *y,  int *channels_in_file, int desired_channels);

   #ifndef STBI_NO_STDIO
   STBIDEF float *stbi_loadf            (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
   STBIDEF float *stbi_loadf_from_file  (FILE *f, int *x, int *y, int *channels_in_file, int desired_channels);
   #endif
#endif

#ifndef STBI_NO_HDR
   STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma);
   STBIDEF void   stbi_hdr_to_ldr_scale(float scale);
#endif // STBI_NO_HDR

#ifndef STBI_NO_LINEAR
   STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma);
   STBIDEF void   stbi_ldr_to_hdr_scale(float scale);
#endif // STBI_NO_LINEAR

// stbi_is_hdr is always defined, but always returns false if STBI_NO_HDR
STBIDEF int    stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user);
STBIDEF int    stbi_is_hdr_from_memory(stbi_uc const *buffer, int len);
#ifndef STBI_NO_STDIO
STBIDEF int      stbi_is_hdr          (char const *filename);
STBIDEF int      stbi_is_hdr_from_file(FILE *f);
#endif // STBI_NO_STDIO


// get a VERY brief reason for failure
// on most compilers (and ALL modern mainstream compilers) this is threadsafe
STBIDEF const char *stbi_failure_reason  (void);

// free the loaded image -- this is just free()
STBIDEF void     stbi_image_free      (void *retval_from_stbi_load);

// get image dimensions & components without fully decoding
STBIDEF int      stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp);
STBIDEF int      stbi_info_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *comp);
STBIDEF int      stbi_is_16_bit_from_memory(stbi_uc const *buffer, int len);
STBIDEF int      stbi_is_16_bit_from_callbacks(stbi_io_callbacks const *clbk, void *user);

#ifndef STBI_NO_STDIO
STBIDEF int      stbi_info               (char const *filename,     int *x, int *y, int *comp);
STBIDEF int      stbi_info_from_file     (FILE *f,                  int *x, int *y, int *comp);
STBIDEF int      stbi_is_16_bit          (char const *filename);
STBIDEF int      stbi_is_16_bit_from_file(FILE *f);
#endif


```

```

// for image formats that explicitly notate that they have premultiplied alpha,
// we just return the colors as stored in the file. set this flag to force
// unpremultiplication. results are undefined if the unpremultiply overflow.
STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply);

// indicate whether we should process iphone images back to canonical format,
// or just pass them through "as-is"
STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert);

// flip the image vertically, so the first pixel in the output array is the bottom left
STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip);

// as above, but only applies to images loaded on the thread that calls the function
// this function is only available if your compiler supports thread-local variables;
// calling it will fail to link if your compiler doesn’t
STBIDEF void stbi_set_unpremultiply_on_load_thread(int flag_true_if_should_unpremultiply);
STBIDEF void stbi_convert_iphone_png_to_rgb_thread(int flag_true_if_should_convert);
STBIDEF void stbi_set_flip_vertically_on_load_thread(int flag_true_if_should_flip);

// ZLIB client - used by PNG, available for other purposes

STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int *outlen);
STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int initial_size, int *outlen, int parse_header);
STBIDEF char *stbi_zlib_decode_malloc(const char *buffer, int len, int *outlen);
STBIDEF int   stbi_zlib_decode_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);

STBIDEF char *stbi_zlib_decode_noheader_malloc(const char *buffer, int len, int *outlen);
STBIDEF int   stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);


#ifdef __cplusplus
}
#endif

//
//
#endif // STBI_INCLUDE_STB_IMAGE_H

#ifdef STB_IMAGE_IMPLEMENTATION

#if defined(STBI_ONLY_JPEG) || defined(STBI_ONLY_PNG) || defined(STBI_ONLY_BMP) \
  || defined(STBI_ONLY_TGA) || defined(STBI_ONLY_GIF) || defined(STBI_ONLY_PSD) \
  || defined(STBI_ONLY_HDR) || defined(STBI_ONLY_PIC) || defined(STBI_ONLY_PNM) \
  || defined(STBI_ONLY_ZLIB)
   #ifndef STBI_ONLY_JPEG
   #define STBI_NO_JPEG
   #endif
   #ifndef STBI_ONLY_PNG
   #define STBI_NO_PNG
   #endif
   #ifndef STBI_ONLY_BMP
   #define STBI_NO_BMP
   #endif
   #ifndef STBI_ONLY_PSD
   #define STBI_NO_PSD
   #endif
   #ifndef STBI_ONLY_TGA
   #define STBI_NO_TGA
   #endif
   #ifndef STBI_ONLY_GIF
   #define STBI_NO_GIF
   #endif
   #ifndef STBI_ONLY_HDR
   #define STBI_NO_HDR
   #endif
   #ifndef STBI_ONLY_PIC
   #define STBI_NO_PIC
   #endif
   #ifndef STBI_ONLY_PNM
   #define STBI_NO_PNM
   #endif
#endif

#if defined(STBI_NO_PNG) && !defined(STBI_SUPPORT_ZLIB) && !defined(STBI_NO_ZLIB)
#define STBI_NO_ZLIB
#endif


#include <stdarg.h>
#include <stddef.h> // ptrdiff_t on osx
#include <stdlib.h>
#include <string.h>
#include <limits.h>
```

```

#if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR)
#include <math.h>  // ldexp, pow
#endif

#ifndef STBI_NO_STDIO
#include <stdio.h>
#endif

#ifndef STBI_ASSERT
#include <assert.h>
#define STBI_ASSERT(x) assert(x)
#endif

#ifdef __cplusplus
#define STBI_EXTERN extern "C"
#else
#define STBI_EXTERN extern
#endif


#ifndef _MSC_VER
   #ifdef __cplusplus
   #define stbi_inline inline
   #else
   #define stbi_inline
   #endif
#else
   #define stbi_inline __forceinline
#endif

#ifndef STBI_NO_THREAD_LOCALS
   #if defined(__cplusplus) &&  __cplusplus >= 201103L
      #define STBI_THREAD_LOCAL       thread_local
   #elif defined(__GNUC__) && __GNUC__ < 5
      #define STBI_THREAD_LOCAL       __thread
   #elif defined(_MSC_VER)
      #define STBI_THREAD_LOCAL       __declspec(thread)
   #elif defined (__STDC_VERSION__) && __STDC_VERSION__ >= 201112L && !defined(__STDC_NO_THREADS__)
      #define STBI_THREAD_LOCAL       _Thread_local
   #endif

   #ifndef STBI_THREAD_LOCAL
      #if defined(__GNUC__)
        #define STBI_THREAD_LOCAL       __thread
      #endif
   #endif
#endif

#if defined(_MSC_VER) || defined(__SYMBIAN32__)
typedef unsigned short stbi__uint16;
typedef   signed short stbi__int16;
typedef unsigned int   stbi__uint32;
typedef   signed int   stbi__int32;
#else
#include <stdint.h>
typedef uint16_t stbi__uint16;
typedef int16_t  stbi__int16;
typedef uint32_t stbi__uint32;
typedef int32_t  stbi__int32;
#endif

// should produce compiler error if size is wrong
typedef unsigned char validate_uint32[sizeof(stbi__uint32)==4 ? 1 : -1];

#ifdef _MSC_VER
#define STBI_NOTUSED(v)  (void)(v)
#else
#define STBI_NOTUSED(v)  (void)sizeof(v)
#endif

#ifdef _MSC_VER
#define STBI_HAS_LROTL
#endif

#ifdef STBI_HAS_LROTL
   #define stbi_lrot(x,y)  _lrotl(x,y)
#else
   #define stbi_lrot(x,y)  (((x) « (y)) | ((x) » (-(y) & 31)))
#endif

#if defined(STBI_MALLOC) && defined(STBI_FREE) && (defined(STBI_REALLOC) || defined(STBI_REALLOC_SIZED))
// ok
#elif !defined(STBI_MALLOC) && !defined(STBI_FREE) && !defined(STBI_REALLOC) && !defined(STBI_REALLOC_SIZED)
// ok
```

```
#else
#error "Must define all or none of STBI_MALLOC, STBI_FREE, and STBI_REALLOC (or STBI_REALLOC_SIZED)."
#endif

#ifndef STBI_MALLOC
#define STBI_MALLOC(sz)           malloc(sz)
#define STBI_REALLOC(p,newsz)     realloc(p,newsz)
#define STBI_FREE(p)              free(p)
#endif

#ifndef STBI_REALLOC_SIZED
#define STBI_REALLOC_SIZED(p,oldsz,newsz) STBI_REALLOC(p,newsz)
#endif

// x86/x64 detection
#if defined(__x86_64__) || defined(_M_X64)
#define STBI__X64_TARGET
#elif defined(__i386) || defined(_M_IX86)
#define STBI__X86_TARGET
#endif

#if defined(__GNUC__) && defined(STBI__X86_TARGET) && !defined(__SSE2__) && !defined(STBI_NO_SIMD)
// gcc doesn't support sse2 intrinsics unless you compile with -msse2,
// which in turn means it gets to use SSE2 everywhere. This is unfortunate,
// but previous attempts to provide the SSE2 functions with runtime
// detection caused numerous issues. The way architecture extensions are
// exposed in GCC/Clang is, sadly, not really suited for one-file libs.
// New behavior: if compiled with -msse2, we use SSE2 without any
// detection; if not, we don't use it at all.
#define STBI_NO_SIMD
#endif

#if defined(__MINGW32__) && defined(STBI__X86_TARGET) && !defined(STBI_MINGW_ENABLE_SSE2) && !defined(STBI_NO_SIMD)
// Note that __MINGW32__ doesn't actually mean 32-bit, so we have to avoid STBI__X64_TARGET
//
// 32-bit MinGW wants ESP to be 16-byte aligned, but this is not in the
// Windows ABI and VC++ as well as Windows DLLs don't maintain that invariant.
// As a result, enabling SSE2 on 32-bit MinGW is dangerous when not
// simultaneously enabling "-mstackrealign".
//
// See https://github.com/nothings/stb/issues/81 for more information.
//
// So default to no SSE2 on 32-bit MinGW. If you've read this far and added
// -mstackrealign to your build settings, feel free to #define STBI_MINGW_ENABLE_SSE2.
#define STBI_NO_SIMD
#endif

#if !defined(STBI_NO_SIMD) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET))
#define STBI_SSE2
#include <emmintrin.h>

#ifdef _MSC_VER

#if _MSC_VER >= 1400  // not VC6
#include <intrin.h> // __cpuid
static int stbi__cpuid3(void)
{
   int info[4];
   __cpuid(info,1);
   return info[3];
}
#else
static int stbi__cpuid3(void)
{
   int res;
   __asm {
      mov  eax,1
      cpuid
      mov  res,edx
   }
   return res;
}
#endif

#define STBI_SIMD_ALIGN(type, name) __declspec(align(16)) type name

#if !defined(STBI_NO_JPEG) && defined(STBI_SSE2)
static int stbi__sse2_available(void)
{
   int info3 = stbi__cpuid3();
   return ((info3 » 26) & 1) != 0;
}
#endif

#else // assume GCC-style if not VC++
#define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
```

```

#if !defined(STBI_NO_JPEG) && defined(STBI_SSE2)
static int stbi__sse2_available(void)
{
   // If we’re even attempting to compile this on GCC/Clang, that means
   // -msse2 is on, which means the compiler is allowed to use SSE2
   // instructions at will, and so are we.
   return 1;
}
#endif

#endif
#endif

// ARM NEON
#if defined(STBI_NO_SIMD) && defined(STBI_NEON)
#undef STBI_NEON
#endif

#ifdef STBI_NEON
#include <arm_neon.h>
#ifdef _MSC_VER
#define STBI_SIMD_ALIGN(type, name) __declspec(align(16)) type name
#else
#define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
#endif
#endif

#ifndef STBI_SIMD_ALIGN
#define STBI_SIMD_ALIGN(type, name) type name
#endif

#ifndef STBI_MAX_DIMENSIONS
#define STBI_MAX_DIMENSIONS (1 « 24)
#endif

//
//  stbi__context struct and start_xxx functions

// stbi__context structure is our basic context used by all images, so it
// contains all the IO context, plus some basic image information
typedef struct
{
   stbi__uint32 img_x, img_y;
   int img_n, img_out_n;

   stbi_io_callbacks io;
   void *io_user_data;

   int read_from_callbacks;
   int buflen;
   stbi_uc buffer_start[128];
   int callback_already_read;

   stbi_uc *img_buffer, *img_buffer_end;
   stbi_uc *img_buffer_original, *img_buffer_original_end;
} stbi__context;


static void stbi__refill_buffer(stbi__context *s);

// initialize a memory-decode context
static void stbi__start_mem(stbi__context *s, stbi_uc const *buffer, int len)
{
   s->io.read = NULL;
   s->read_from_callbacks = 0;
   s->callback_already_read = 0;
   s->img_buffer = s->img_buffer_original = (stbi_uc *) buffer;
   s->img_buffer_end = s->img_buffer_original_end = (stbi_uc *) buffer+len;
}

// initialize a callback-based context
static void stbi__start_callbacks(stbi__context *s, stbi_io_callbacks *c, void *user)
{
   s->io = *c;
   s->io_user_data = user;
   s->buflen = sizeof(s->buffer_start);
   s->read_from_callbacks = 1;
   s->callback_already_read = 0;
   s->img_buffer = s->img_buffer_original = s->buffer_start;
   stbi__refill_buffer(s);
   s->img_buffer_original_end = s->img_buffer_end;
}

#ifndef STBI_NO_STDIO

static int stbi__stdio_read(void *user, char *data, int size)
```

```
{
   return (int) fread(data,1,size,(FILE*) user);
}

static void stbi__stdio_skip(void *user, int n)
{
   int ch;
   fseek((FILE*) user, n, SEEK_CUR);
   ch = fgetc((FILE*) user);  /* have to read a byte to reset feof()'s flag */
   if (ch != EOF) {
      ungetc(ch, (FILE *) user);  /* push byte back onto stream if valid. */
   }
}

static int stbi__stdio_eof(void *user)
{
   return feof((FILE*) user) || ferror((FILE *) user);
}

static stbi_io_callbacks stbi__stdio_callbacks =
{
   stbi__stdio_read,
   stbi__stdio_skip,
   stbi__stdio_eof,
};

static void stbi__start_file(stbi__context *s, FILE *f)
{
   stbi__start_callbacks(s, &stbi__stdio_callbacks, (void *) f);
}

//static void stop_file(stbi__context *s) { }

#endif // !STBI_NO_STDIO

static void stbi__rewind(stbi__context *s)
{
   // conceptually rewind SHOULD rewind to the beginning of the stream,
   // but we just rewind to the beginning of the initial buffer, because
   // we only use it after doing 'test', which only ever looks at at most 92 bytes
   s->img_buffer = s->img_buffer_original;
   s->img_buffer_end = s->img_buffer_original_end;
}

enum
{
   STBI_ORDER_RGB,
   STBI_ORDER_BGR
};

typedef struct
{
   int bits_per_channel;
   int num_channels;
   int channel_order;
} stbi__result_info;

#ifndef STBI_NO_JPEG
static int      stbi__jpeg_test(stbi__context *s);
static void    *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri);
static int      stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PNG
static int      stbi__png_test(stbi__context *s);
static void    *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri);
static int      stbi__png_info(stbi__context *s, int *x, int *y, int *comp);
static int      stbi__png_is16(stbi__context *s);
#endif

#ifndef STBI_NO_BMP
static int      stbi__bmp_test(stbi__context *s);
static void    *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri);
static int      stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_TGA
static int      stbi__tga_test(stbi__context *s);
static void    *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri);
static int      stbi__tga_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PSD
```

```
static int      stbi__psd_test(stbi__context *s);
static void    *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri, int bpc);
static int      stbi__psd_info(stbi__context *s, int *x, int *y, int *comp);
static int      stbi__psd_is16(stbi__context *s);
#endif

#ifndef STBI_NO_HDR
static int      stbi__hdr_test(stbi__context *s);
static float   *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri);
static int      stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PIC
static int      stbi__pic_test(stbi__context *s);
static void    *stbi__pic_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri);
static int      stbi__pic_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_GIF
static int      stbi__gif_test(stbi__context *s);
static void    *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri);
static void    *stbi__load_gif_main(stbi__context *s, int **delays, int *x, int *y, int *z, int *comp, int req_comp);
static int      stbi__gif_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PNM
static int      stbi__pnm_test(stbi__context *s);
static void    *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri);
static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp);
static int      stbi__pnm_is16(stbi__context *s);
#endif

static
#ifdef STBI_THREAD_LOCAL
STBI_THREAD_LOCAL
#endif
const char *stbi__g_failure_reason;

STBIDEF const char *stbi_failure_reason(void)
{
   return stbi__g_failure_reason;
}

#ifndef STBI_NO_FAILURE_STRINGS
static int stbi__err(const char *str)
{
   stbi__g_failure_reason = str;
   return 0;
}
#endif

static void *stbi__malloc(size_t size)
{
    return STBI_MALLOC(size);
}

// stb_image uses ints pervasively, including for offset calculations.
// therefore the largest decoded image size we can support with the
// current code, even on 64-bit targets, is INT_MAX. this is not a
// significant limitation for the intended use case.
//
// we do, however, need to make sure our size calculations don’t
// overflow. hence a few helper functions for size calculations that
// multiply integers together, making sure that they’re non-negative
// and no overflow occurs.

// return 1 if the sum is valid, 0 on overflow.
// negative terms are considered invalid.
static int stbi__addsizes_valid(int a, int b)
{
   if (b < 0) return 0;
   // now 0 <= b <= INT_MAX, hence also
   // 0 <= INT_MAX - b <= INTMAX.
   // And "a + b <= INT_MAX" (which might overflow) is the
   // same as a <= INT_MAX - b (no overflow)
   return a <= INT_MAX - b;
}

// returns 1 if the product is valid, 0 on overflow.
// negative factors are considered invalid.
static int stbi__mul2sizes_valid(int a, int b)
```

```
{
   if (a < 0 || b < 0) return 0;
   if (b == 0) return 1; // mul-by-0 is always safe
   // portable way to check for no overflows in a*b
   return a <= INT_MAX/b;
}

#if !defined(STBI_NO_JPEG) || !defined(STBI_NO_PNG) || !defined(STBI_NO_TGA) || !defined(STBI_NO_HDR)
// returns 1 if "a*b + add" has no negative terms/factors and doesn't overflow
static int stbi__mad2sizes_valid(int a, int b, int add)
{
   return stbi__mul2sizes_valid(a, b) && stbi__addsizes_valid(a*b, add);
}
#endif

// returns 1 if "a*b*c + add" has no negative terms/factors and doesn't overflow
static int stbi__mad3sizes_valid(int a, int b, int c, int add)
{
   return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
      stbi__addsizes_valid(a*b*c, add);
}

// returns 1 if "a*b*c*d + add" has no negative terms/factors and doesn't overflow
#if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR) || !defined(STBI_NO_PNM)
static int stbi__mad4sizes_valid(int a, int b, int c, int d, int add)
{
   return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
      stbi__mul2sizes_valid(a*b*c, d) && stbi__addsizes_valid(a*b*c*d, add);
}
#endif

#if !defined(STBI_NO_JPEG) || !defined(STBI_NO_PNG) || !defined(STBI_NO_TGA) || !defined(STBI_NO_HDR)
// mallocs with size overflow checking
static void *stbi__malloc_mad2(int a, int b, int add)
{
   if (!stbi__mad2sizes_valid(a, b, add)) return NULL;
   return stbi__malloc(a*b + add);
}
#endif

static void *stbi__malloc_mad3(int a, int b, int c, int add)
{
   if (!stbi__mad3sizes_valid(a, b, c, add)) return NULL;
   return stbi__malloc(a*b*c + add);
}

#if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR) || !defined(STBI_NO_PNM)
static void *stbi__malloc_mad4(int a, int b, int c, int d, int add)
{
   if (!stbi__mad4sizes_valid(a, b, c, d, add)) return NULL;
   return stbi__malloc(a*b*c*d + add);
}
#endif

// returns 1 if the sum of two signed ints is valid (between -2^31 and 2^31-1 inclusive), 0 on overflow.
static int stbi__addints_valid(int a, int b)
{
   if ((a >= 0) != (b >= 0)) return 1; // a and b have different signs, so no overflow
   if (a < 0 && b < 0) return a >= INT_MIN - b; // same as a + b >= INT_MIN; INT_MIN - b cannot overflow since b < 0.
   return a <= INT_MAX - b;
}

// returns 1 if the product of two signed shorts is valid, 0 on overflow.
static int stbi__mul2shorts_valid(short a, short b)
{
   if (b == 0 || b == -1) return 1; // multiplication by 0 is always 0; check for -1 so SHRT_MIN/b doesn't overflow
   if ((a >= 0) == (b >= 0)) return a <= SHRT_MAX/b; // product is positive, so similar to mul2sizes_valid
   if (b < 0) return a <= SHRT_MIN / b; // same as a * b >= SHRT_MIN
   return a >= SHRT_MIN / b;
}

// stbi__err - error
// stbi__errpf - error returning pointer to float
// stbi__errpuc - error returning pointer to unsigned char

#ifdef STBI_NO_FAILURE_STRINGS
   #define stbi__err(x,y)  0
#elif defined(STBI_FAILURE_USERMSG)
   #define stbi__err(x,y)  stbi__err(y)
#else
   #define stbi__err(x,y)  stbi__err(x)
#endif

```

```
#define stbi__errpf(x,y)   ((float *)(size_t) (stbi__err(x,y)?NULL:NULL))
#define stbi__errpuc(x,y)  ((unsigned char *)(size_t) (stbi__err(x,y)?NULL:NULL))

STBIDEF void stbi_image_free(void *retval_from_stbi_load)
{
   STBI_FREE(retval_from_stbi_load);
}

#ifndef STBI_NO_LINEAR
static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp);
#endif

#ifndef STBI_NO_HDR
static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp);
#endif

static int stbi__vertically_flip_on_load_global = 0;

STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip)
{
   stbi__vertically_flip_on_load_global = flag_true_if_should_flip;
}

#ifndef STBI_THREAD_LOCAL
#define stbi__vertically_flip_on_load   stbi__vertically_flip_on_load_global
#else
static STBI_THREAD_LOCAL int stbi__vertically_flip_on_load_local, stbi__vertically_flip_on_load_set;

STBIDEF void stbi_set_flip_vertically_on_load_thread(int flag_true_if_should_flip)
{
   stbi__vertically_flip_on_load_local = flag_true_if_should_flip;
   stbi__vertically_flip_on_load_set = 1;
}

#define stbi__vertically_flip_on_load   (stbi__vertically_flip_on_load_set       \
                                         ? stbi__vertically_flip_on_load_local  \
                                         : stbi__vertically_flip_on_load_global)
#endif // STBI_THREAD_LOCAL

static void *stbi__load_main(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri, int bpc)
{
   memset(ri, 0, sizeof(*ri)); // make sure it's initialized if we add new fields
   ri->bits_per_channel = 8; // default is 8 so most paths don't have to be changed
   ri->channel_order = STBI_ORDER_RGB; // all current input & output are this, but this is here so we can add BGR order
   ri->num_channels = 0;

   // test the formats with a very explicit header first (at least a FOURCC
   // or distinctive magic number first)
   #ifndef STBI_NO_PNG
   if (stbi__png_test(s))  return stbi__png_load(s,x,y,comp,req_comp, ri);
   #endif
   #ifndef STBI_NO_BMP
   if (stbi__bmp_test(s))  return stbi__bmp_load(s,x,y,comp,req_comp, ri);
   #endif
   #ifndef STBI_NO_GIF
   if (stbi__gif_test(s))  return stbi__gif_load(s,x,y,comp,req_comp, ri);
   #endif
   #ifndef STBI_NO_PSD
   if (stbi__psd_test(s))  return stbi__psd_load(s,x,y,comp,req_comp, ri, bpc);
   #else
   STBI_NOTUSED(bpc);
   #endif
   #ifndef STBI_NO_PIC
   if (stbi__pic_test(s))  return stbi__pic_load(s,x,y,comp,req_comp, ri);
   #endif

   // then the formats that can end up attempting to load with just 1 or 2
   // bytes matching expectations; these are prone to false positives, so
   // try them later
   #ifndef STBI_NO_JPEG
   if (stbi__jpeg_test(s)) return stbi__jpeg_load(s,x,y,comp,req_comp, ri);
   #endif
   #ifndef STBI_NO_PNM
   if (stbi__pnm_test(s))  return stbi__pnm_load(s,x,y,comp,req_comp, ri);
   #endif

   #ifndef STBI_NO_HDR
   if (stbi__hdr_test(s)) {
      float *hdr = stbi__hdr_load(s, x,y,comp,req_comp, ri);
      return stbi__hdr_to_ldr(hdr, *x, *y, req_comp ? req_comp : *comp);
   }
   #endif

   #ifndef STBI_NO_TGA
   // test tga last because it's a crappy test!
   if (stbi__tga_test(s))
      return stbi__tga_load(s,x,y,comp,req_comp, ri);
   #endif

   return stbi__errpuc("unknown image type", "Image not of any known type, or corrupt");
}

static stbi_uc *stbi__convert_16_to_8(stbi__uint16 *orig, int w, int h, int channels)
{
   int i;
   int img_len = w * h * channels;
   stbi_uc *reduced;

   reduced = (stbi_uc *) stbi__malloc(img_len);
   if (reduced == NULL) return stbi__errpuc("outofmem", "Out of memory");

   for (i = 0; i < img_len; ++i)
      reduced[i] = (stbi_uc)((orig[i] » 8) & 0xFF); // top half of each byte is sufficient approx of 16->8 bit scaling

   STBI_FREE(orig);
   return reduced;
}

static stbi__uint16 *stbi__convert_8_to_16(stbi_uc *orig, int w, int h, int channels)
{
   int i;
   int img_len = w * h * channels;
   stbi__uint16 *enlarged;

   enlarged = (stbi__uint16 *) stbi__malloc(img_len*2);
   if (enlarged == NULL) return (stbi__uint16 *) stbi__errpuc("outofmem", "Out of memory");

   for (i = 0; i < img_len; ++i)
      enlarged[i] = (stbi__uint16)((orig[i] « 8) + orig[i]); // replicate to high and low byte, maps 0->0, 255->0xffff

   STBI_FREE(orig);
   return enlarged;
}

static void stbi__vertical_flip(void *image, int w, int h, int bytes_per_pixel)
{
   int row;
   size_t bytes_per_row = (size_t)w * bytes_per_pixel;
   stbi_uc temp[2048];
   stbi_uc *bytes = (stbi_uc *)image;

   for (row = 0; row < (h»1); row++) {
      stbi_uc *row0 = bytes + row*bytes_per_row;
      stbi_uc *row1 = bytes + (h - row - 1)*bytes_per_row;
      // swap row0 with row1
      size_t bytes_left = bytes_per_row;
      while (bytes_left) {
         size_t bytes_copy = (bytes_left < sizeof(temp)) ? bytes_left : sizeof(temp);
         memcpy(temp, row0, bytes_copy);
         memcpy(row0, row1, bytes_copy);
         memcpy(row1, temp, bytes_copy);
         row0 += bytes_copy;
         row1 += bytes_copy;
         bytes_left -= bytes_copy;
      }
   }
}

#ifndef STBI_NO_GIF
static void stbi__vertical_flip_slices(void *image, int w, int h, int z, int bytes_per_pixel)
{
   int slice;
   int slice_size = w * h * bytes_per_pixel;

   stbi_uc *bytes = (stbi_uc *)image;
   for (slice = 0; slice < z; ++slice) {
      stbi__vertical_flip(bytes, w, h, bytes_per_pixel);
      bytes += slice_size;
   }
}
#endif

static unsigned char *stbi__load_and_postprocess_8bit(stbi__context *s, int *x, int *y, int *comp, int req_comp)
{
   stbi__result_info ri;
   void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 8);

   if (result == NULL)
      return NULL;

   // it is the responsibility of the loaders to make sure we get either 8 or 16 bit.
   STBI_ASSERT(ri.bits_per_channel == 8 || ri.bits_per_channel == 16);

   if (ri.bits_per_channel != 8) {
      result = stbi__convert_16_to_8((stbi__uint16 *) result, *x, *y, req_comp == 0 ? *comp : req_comp);
      ri.bits_per_channel = 8;
   }

   // @TODO: move stbi__convert_format to here

   if (stbi__vertically_flip_on_load) {
      int channels = req_comp ? req_comp : *comp;
      stbi__vertical_flip(result, *x, *y, channels * sizeof(stbi_uc));
   }

   return (unsigned char *) result;
}

static stbi__uint16 *stbi__load_and_postprocess_16bit(stbi__context *s, int *x, int *y, int *comp, int req_comp)
{
   stbi__result_info ri;
   void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 16);

   if (result == NULL)
      return NULL;

   // it is the responsibility of the loaders to make sure we get either 8 or 16 bit.
   STBI_ASSERT(ri.bits_per_channel == 8 || ri.bits_per_channel == 16);

   if (ri.bits_per_channel != 16) {
      result = stbi__convert_8_to_16((stbi_uc *) result, *x, *y, req_comp == 0 ? *comp : req_comp);
      ri.bits_per_channel = 16;
   }

   // @TODO: move stbi__convert_format16 to here
   // @TODO: special case RGB-to-Y (and RGBA-to-YA) for 8-bit-to-16-bit case to keep more precision

   if (stbi__vertically_flip_on_load) {
      int channels = req_comp ? req_comp : *comp;
      stbi__vertical_flip(result, *x, *y, channels * sizeof(stbi__uint16));
   }

   return (stbi__uint16 *) result;
}

#if !defined(STBI_NO_HDR) && !defined(STBI_NO_LINEAR)
static void stbi__float_postprocess(float *result, int *x, int *y, int *comp, int req_comp)
{
   if (stbi__vertically_flip_on_load && result != NULL) {
      int channels = req_comp ? req_comp : *comp;
      stbi__vertical_flip(result, *x, *y, channels * sizeof(float));
   }
}
#endif

#ifndef STBI_NO_STDIO

#if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
STBI_EXTERN __declspec(dllimport) int __stdcall MultiByteToWideChar(unsigned int cp, unsigned long flags, const char *str, int cbmb, wchar_t *widestr, int cchwide);
STBI_EXTERN __declspec(dllimport) int __stdcall WideCharToMultiByte(unsigned int cp, unsigned long flags, const wchar_t *widestr, int cchwide, char *str, int cbmb, const char *defchar, int *used_default);
#endif

#if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
STBIDEF int stbi_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input)
{
    return WideCharToMultiByte(65001 /* UTF8 */, 0, input, -1, buffer, (int) bufferlen, NULL, NULL);
}
#endif

static FILE *stbi__fopen(char const *filename, char const *mode)
{
   FILE *f;
#if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
   wchar_t wMode[64];
   wchar_t wFilename[1024];
    if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, filename, -1, wFilename, sizeof(wFilename)/sizeof(*wFilename)))
      return 0;

    if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, mode, -1, wMode, sizeof(wMode)/sizeof(*wMode)))
      return 0;
```

```

#if defined(_MSC_VER) && _MSC_VER >= 1400
    if (0 != _wfopen_s(&f, wFilename, wMode))
        f = 0;
#else
   f = _wfopen(wFilename, wMode);
#endif

#elif defined(_MSC_VER) && _MSC_VER >= 1400
   if (0 != fopen_s(&f, filename, mode))
      f=0;
#else
   f = fopen(filename, mode);
#endif
   return f;
}


STBIDEF stbi_uc *stbi_load(char const *filename, int *x, int *y, int *comp, int req_comp)
{
   FILE *f = stbi__fopen(filename, "rb");
   unsigned char *result;
   if (!f) return stbi__errpuc("can't fopen", "Unable to open file");
   result = stbi_load_from_file(f,x,y,comp,req_comp);
   fclose(f);
   return result;
}

STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
{
   unsigned char *result;
   stbi__context s;
   stbi__start_file(&s,f);
   result = stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
   if (result) {
      // need to 'unget' all the characters in the IO buffer
      fseek(f, - (int) (s.img_buffer_end - s.img_buffer), SEEK_CUR);
   }
   return result;
}

STBIDEF stbi__uint16 *stbi_load_from_file_16(FILE *f, int *x, int *y, int *comp, int req_comp)
{
   stbi__uint16 *result;
   stbi__context s;
   stbi__start_file(&s,f);
   result = stbi__load_and_postprocess_16bit(&s,x,y,comp,req_comp);
   if (result) {
      // need to 'unget' all the characters in the IO buffer
      fseek(f, - (int) (s.img_buffer_end - s.img_buffer), SEEK_CUR);
   }
   return result;
}

STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *comp, int req_comp)
{
   FILE *f = stbi__fopen(filename, "rb");
   stbi__uint16 *result;
   if (!f) return (stbi_us *) stbi__errpuc("can't fopen", "Unable to open file");
   result = stbi_load_from_file_16(f,x,y,comp,req_comp);
   fclose(f);
   return result;
}


#endif

STBIDEF stbi_us *stbi_load_16_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels)
{
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__load_and_postprocess_16bit(&s,x,y,channels_in_file,desired_channels);
}

STBIDEF stbi_us *stbi_load_16_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels)
{
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
   return stbi__load_and_postprocess_16bit(&s,x,y,channels_in_file,desired_channels);
}

STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int req_comp)
{
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
}

STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *comp, int req_comp)
{
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
   return stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
}

#ifndef STBI_NO_GIF
STBIDEF stbi_uc *stbi_load_gif_from_memory(stbi_uc const *buffer, int len, int **delays, int *x, int *y, int *z, int *comp, int req_comp)
{
   unsigned char *result;
   stbi__context s;
   stbi__start_mem(&s,buffer,len);

   result = (unsigned char*) stbi__load_gif_main(&s, delays, x, y, z, comp, req_comp);
   if (stbi__vertically_flip_on_load) {
      stbi__vertical_flip_slices( result, *x, *y, *z, *comp );
   }

   return result;
}
#endif

#ifndef STBI_NO_LINEAR
static float *stbi__loadf_main(stbi__context *s, int *x, int *y, int *comp, int req_comp)
{
   unsigned char *data;
   #ifndef STBI_NO_HDR
   if (stbi__hdr_test(s)) {
      stbi__result_info ri;
      float *hdr_data = stbi__hdr_load(s,x,y,comp,req_comp, &ri);
      if (hdr_data)
         stbi__float_postprocess(hdr_data,x,y,comp,req_comp);
      return hdr_data;
   }
   #endif
   data = stbi__load_and_postprocess_8bit(s, x, y, comp, req_comp);
   if (data)
      return stbi__ldr_to_hdr(data, *x, *y, req_comp ? req_comp : *comp);
   return stbi__errpf("unknown image type", "Image not of any known type, or corrupt");
}

STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int req_comp)
{
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__loadf_main(&s,x,y,comp,req_comp);
}

STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *comp, int req_comp)
{
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
   return stbi__loadf_main(&s,x,y,comp,req_comp);
}

#ifndef STBI_NO_STDIO
STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *comp, int req_comp)
{
   float *result;
   FILE *f = stbi__fopen(filename, "rb");
   if (!f) return stbi__errpf("can't fopen", "Unable to open file");
   result = stbi_loadf_from_file(f,x,y,comp,req_comp);
   fclose(f);
   return result;
}

STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
{
   stbi__context s;
   stbi__start_file(&s,f);
   return stbi__loadf_main(&s,x,y,comp,req_comp);
}
#endif // !STBI_NO_STDIO

#endif // !STBI_NO_LINEAR

// these is-hdr-or-not is defined independent of whether STBI_NO_LINEAR is
// defined, for API simplicity; if STBI_NO_LINEAR is defined, it always
// reports false!

STBIDEF int stbi_is_hdr_from_memory(stbi_uc const *buffer, int len)
{
   #ifndef STBI_NO_HDR
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__hdr_test(&s);
   #else
   STBI_NOTUSED(buffer);
   STBI_NOTUSED(len);
   return 0;
   #endif
}

#ifndef STBI_NO_STDIO
STBIDEF int      stbi_is_hdr          (char const *filename)
{
   FILE *f = stbi__fopen(filename, "rb");
   int result=0;
   if (f) {
      result = stbi_is_hdr_from_file(f);
      fclose(f);
   }
   return result;
}

STBIDEF int stbi_is_hdr_from_file(FILE *f)
{
   #ifndef STBI_NO_HDR
   long pos = ftell(f);
   int res;
   stbi__context s;
   stbi__start_file(&s,f);
   res = stbi__hdr_test(&s);
   fseek(f, pos, SEEK_SET);
   return res;
   #else
   STBI_NOTUSED(f);
   return 0;
   #endif
}
#endif // !STBI_NO_STDIO

STBIDEF int      stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user)
{
   #ifndef STBI_NO_HDR
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
   return stbi__hdr_test(&s);
   #else
   STBI_NOTUSED(clbk);
   STBI_NOTUSED(user);
   return 0;
   #endif
}

#ifndef STBI_NO_LINEAR
static float stbi__l2h_gamma=2.2f, stbi__l2h_scale=1.0f;

STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma) { stbi__l2h_gamma = gamma; }
STBIDEF void   stbi_ldr_to_hdr_scale(float scale) { stbi__l2h_scale = scale; }
#endif

static float stbi__h2l_gamma_i=1.0f/2.2f, stbi__h2l_scale_i=1.0f;

STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma) { stbi__h2l_gamma_i = 1/gamma; }
STBIDEF void   stbi_hdr_to_ldr_scale(float scale) { stbi__h2l_scale_i = 1/scale; }


//
// Common code used by all image loaders
//

enum
{
   STBI__SCAN_load=0,
   STBI__SCAN_type,
   STBI__SCAN_header
};

static void stbi__refill_buffer(stbi__context *s)
{
   int n = (s->io.read)(s->io_user_data,(char*)s->buffer_start,s->buflen);
   s->callback_already_read += (int) (s->img_buffer - s->img_buffer_original);
   if (n == 0) {
      // at end of file, treat same as if from memory, but need to handle case
      // where s->img_buffer isn't pointing to safe memory, e.g. 0-byte file
      s->read_from_callbacks = 0;
      s->img_buffer = s->buffer_start;
      s->img_buffer_end = s->buffer_start+1;
      *s->img_buffer = 0;
   } else {
      s->img_buffer = s->buffer_start;
      s->img_buffer_end = s->buffer_start + n;
   }
}

stbi_inline static stbi_uc stbi__get8(stbi__context *s)
{
   if (s->img_buffer < s->img_buffer_end)
      return *s->img_buffer++;
   if (s->read_from_callbacks) {
      stbi__refill_buffer(s);
      return *s->img_buffer++;
   }
   return 0;
}

#if defined(STBI_NO_JPEG) && defined(STBI_NO_HDR) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
// nothing
#else
stbi_inline static int stbi__at_eof(stbi__context *s)
{
   if (s->io.read) {
      if (!(s->io.eof)(s->io_user_data)) return 0;
      // if feof() is true, check if buffer = end
      // special case: we've only got the special 0 character at the end
      if (s->read_from_callbacks == 0) return 1;
   }

   return s->img_buffer >= s->img_buffer_end;
}
#endif

#if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF) && defined(STBI_NO_PIC)
// nothing
#else
static void stbi__skip(stbi__context *s, int n)
{
   if (n == 0) return;  // already there!
   if (n < 0) {
      s->img_buffer = s->img_buffer_end;
      return;
   }
   if (s->io.read) {
      int blen = (int) (s->img_buffer_end - s->img_buffer);
      if (blen < n) {
         s->img_buffer = s->img_buffer_end;
         (s->io.skip)(s->io_user_data, n - blen);
         return;
      }
   }
   s->img_buffer += n;
}
#endif

#if defined(STBI_NO_PNG) && defined(STBI_NO_TGA) && defined(STBI_NO_HDR) && defined(STBI_NO_PNM)
// nothing
#else
static int stbi__getn(stbi__context *s, stbi_uc *buffer, int n)
{
   if (s->io.read) {
      int blen = (int) (s->img_buffer_end - s->img_buffer);
      if (blen < n) {
         int res, count;

         memcpy(buffer, s->img_buffer, blen);

         count = (s->io.read)(s->io_user_data, (char*) buffer + blen, n - blen);
         res = (count == (n-blen));
         s->img_buffer = s->img_buffer_end;
         return res;
      }
   }

   if (s->img_buffer+n <= s->img_buffer_end) {
      memcpy(buffer, s->img_buffer, n);
      s->img_buffer += n;
      return 1;
   } else
      return 0;
}
```

```
#endif

#if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_PSD) && defined(STBI_NO_PIC)
// nothing
#else
static int stbi__get16be(stbi__context *s)
{
   int z = stbi__get8(s);
   return (z « 8) + stbi__get8(s);
}
#endif

#if defined(STBI_NO_PNG) && defined(STBI_NO_PSD) && defined(STBI_NO_PIC)
// nothing
#else
static stbi__uint32 stbi__get32be(stbi__context *s)
{
   stbi__uint32 z = stbi__get16be(s);
   return (z « 16) + stbi__get16be(s);
}
#endif

#if defined(STBI_NO_BMP) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF)
// nothing
#else
static int stbi__get16le(stbi__context *s)
{
   int z = stbi__get8(s);
   return z + (stbi__get8(s) « 8);
}
#endif

#ifndef STBI_NO_BMP
static stbi__uint32 stbi__get32le(stbi__context *s)
{
   stbi__uint32 z = stbi__get16le(s);
   z += (stbi__uint32)stbi__get16le(s) « 16;
   return z;
}
#endif

#define STBI__BYTECAST(x)  ((stbi_uc) ((x) & 255))  // truncate int to byte without warnings

#if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
// nothing
#else
//
//  generic converter from built-in img_n to req_comp
//    individual types do this automatically as much as possible (e.g. jpeg
//    does all cases internally since it needs to colorspace convert anyway,
//    and it never has alpha, so very few cases ). png can automatically
//    interleave an alpha=255 channel, but falls back to this for other cases
//
//  assume data buffer is malloced, so malloc a new one and free that one
//  only failure mode is malloc failing

static stbi_uc stbi__compute_y(int r, int g, int b)
{
   return (stbi_uc) (((r*77) + (g*150) +  (29*b)) » 8);
}
#endif

#if defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
// nothing
#else
static unsigned char *stbi__convert_format(unsigned char *data, int img_n, int req_comp, unsigned int x, unsigned int y)
{
   int i,j;
   unsigned char *good;

   if (req_comp == img_n) return data;
   STBI_ASSERT(req_comp >= 1 && req_comp <= 4);

   good = (unsigned char *) stbi__malloc_mad3(req_comp, x, y, 0);
   if (good == NULL) {
      STBI_FREE(data);
      return stbi__errpuc("outofmem", "Out of memory");
   }

   for (j=0; j < (int) y; ++j) {
      unsigned char *src  = data + j * x * img_n   ;
      unsigned char *dest = good + j * x * req_comp;

      #define STBI__COMBO(a,b)  ((a)*8+(b))
      #define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
      // convert source image with img_n components to one with req_comp components;
      // avoid switch per pixel, so use switch per scanline and massive macros
      switch (STBI__COMBO(img_n, req_comp)) {
         STBI__CASE(1,2) { dest[0]=src[0]; dest[1]=255;                                     } break;
         STBI__CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0];                                  } break;
         STBI__CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=255;                     } break;
         STBI__CASE(2,1) { dest[0]=src[0];                                                  } break;
         STBI__CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0];                                  } break;
         STBI__CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=src[1];                  } break;
         STBI__CASE(3,4) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];dest[3]=255;        } break;
         STBI__CASE(3,1) { dest[0]=stbi__compute_y(src[0],src[1],src[2]);                   } break;
         STBI__CASE(3,2) { dest[0]=stbi__compute_y(src[0],src[1],src[2]); dest[1] = 255;    } break;
         STBI__CASE(4,1) { dest[0]=stbi__compute_y(src[0],src[1],src[2]);                   } break;
         STBI__CASE(4,2) { dest[0]=stbi__compute_y(src[0],src[1],src[2]); dest[1] = src[3]; } break;
         STBI__CASE(4,3) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];                    } break;
         default: STBI_ASSERT(0); STBI_FREE(data); STBI_FREE(good); return stbi__errpuc("unsupported", "Unsupported format conversion");
      }
      #undef STBI__CASE
   }

   STBI_FREE(data);
   return good;
}
#endif

#if defined(STBI_NO_PNG) && defined(STBI_NO_PSD)
// nothing
#else
static stbi__uint16 stbi__compute_y_16(int r, int g, int b)
{
   return (stbi__uint16) (((r*77) + (g*150) +  (29*b)) >> 8);
}
#endif

#if defined(STBI_NO_PNG) && defined(STBI_NO_PSD)
// nothing
#else
static stbi__uint16 *stbi__convert_format16(stbi__uint16 *data, int img_n, int req_comp, unsigned int x, unsigned int y)
{
   int i,j;
   stbi__uint16 *good;

   if (req_comp == img_n) return data;
   STBI_ASSERT(req_comp >= 1 && req_comp <= 4);

   good = (stbi__uint16 *) stbi__malloc(req_comp * x * y * 2);
   if (good == NULL) {
      STBI_FREE(data);
      return (stbi__uint16 *) stbi__errpuc("outofmem", "Out of memory");
   }

   for (j=0; j < (int) y; ++j) {
      stbi__uint16 *src  = data + j * x * img_n   ;
      stbi__uint16 *dest = good + j * x * req_comp;

      #define STBI__COMBO(a,b)  ((a)*8+(b))
      #define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
      // convert source image with img_n components to one with req_comp components;
      // avoid switch per pixel, so use switch per scanline and massive macros
      switch (STBI__COMBO(img_n, req_comp)) {
         STBI__CASE(1,2) { dest[0]=src[0]; dest[1]=0xffff;                                     } break;
         STBI__CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0];                                     } break;
         STBI__CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=0xffff;                     } break;
         STBI__CASE(2,1) { dest[0]=src[0];                                                     } break;
         STBI__CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0];                                     } break;
         STBI__CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=src[1];                     } break;
         STBI__CASE(3,4) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];dest[3]=0xffff;        } break;
         STBI__CASE(3,1) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]);                   } break;
         STBI__CASE(3,2) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]); dest[1] = 0xffff; } break;
         STBI__CASE(4,1) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]);                   } break;
         STBI__CASE(4,2) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]); dest[1] = src[3]; } break;
         STBI__CASE(4,3) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];                       } break;
         default: STBI_ASSERT(0); STBI_FREE(data); STBI_FREE(good); return (stbi__uint16*) stbi__errpuc("unsupported", "Unsupported format conversion");
      }
      #undef STBI__CASE
   }

   STBI_FREE(data);
   return good;
}
#endif

#ifndef STBI_NO_LINEAR
static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp)
{
   int i,k,n;
   float *output;
   if (!data) return NULL;
   output = (float *) stbi__malloc_mad4(x, y, comp, sizeof(float), 0);
   if (output == NULL) { STBI_FREE(data); return stbi__errpf("outofmem", "Out of memory"); }
   // compute number of non-alpha components
   if (comp & 1) n = comp; else n = comp-1;
   for (i=0; i < x*y; ++i) {
      for (k=0; k < n; ++k) {
         output[i*comp + k] = (float) (pow(data[i*comp+k]/255.0f, stbi__l2h_gamma) * stbi__l2h_scale);
      }
   }
   if (n < comp) {
      for (i=0; i < x*y; ++i) {
         output[i*comp + n] = data[i*comp + n]/255.0f;
      }
   }
   STBI_FREE(data);
   return output;
}
#endif

#ifndef STBI_NO_HDR
#define stbi__float2int(x)   ((int) (x))
static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp)
{
   int i,k,n;
   stbi_uc *output;
   if (!data) return NULL;
   output = (stbi_uc *) stbi__malloc_mad3(x, y, comp, 0);
   if (output == NULL) { STBI_FREE(data); return stbi__errpuc("outofmem", "Out of memory"); }
   // compute number of non-alpha components
   if (comp & 1) n = comp; else n = comp-1;
   for (i=0; i < x*y; ++i) {
      for (k=0; k < n; ++k) {
         float z = (float) pow(data[i*comp+k]*stbi__h2l_scale_i, stbi__h2l_gamma_i) * 255 + 0.5f;
         if (z < 0) z = 0;
         if (z > 255) z = 255;
         output[i*comp + k] = (stbi_uc) stbi__float2int(z);
      }
      if (k < comp) {
         float z = data[i*comp+k] * 255 + 0.5f;
         if (z < 0) z = 0;
         if (z > 255) z = 255;
         output[i*comp + k] = (stbi_uc) stbi__float2int(z);
      }
   }
   STBI_FREE(data);
   return output;
}
#endif

//
//  "baseline" JPEG/JFIF decoder
//
//    simple implementation
//      - doesn’t support delayed output of y-dimension
//      - simple interface (only one output format: 8-bit interleaved RGB)
//      - doesn’t try to recover corrupt jpegs
//      - doesn’t allow partial loading, loading multiple at once
//      - still fast on x86 (copying globals into locals doesn’t help x86)
//      - allocates lots of intermediate memory (full size of all components)
//        - non-interleaved case requires this anyway
//        - allows good upsampling (see next)
//    high-quality
//      - upsampled channels are bilinearly interpolated, even across blocks
//      - quality integer IDCT derived from IJG’s ’slow’
//    performance
//      - fast huffman; reasonable integer IDCT
//      - some SIMD kernels for common paths on targets with SSE2/NEON
//      - uses a lot of intermediate memory, could cache poorly

#ifndef STBI_NO_JPEG
```

```

// huffman decoding acceleration
#define FAST_BITS   9  // larger handles more cases; smaller stomps less cache

typedef struct
{
   stbi_uc  fast[1 « FAST_BITS];
   // weirdly, repacking this into AoS is a 10% speed loss, instead of a win
   stbi__uint16 code[256];
   stbi_uc  values[256];
   stbi_uc  size[257];
   unsigned int maxcode[18];
   int    delta[17];   // old ’firstsymbol’ - old ’firstcode’
} stbi__huffman;

typedef struct
{
   stbi__context *s;
   stbi__huffman huff_dc[4];
   stbi__huffman huff_ac[4];
   stbi__uint16 dequant[4][64];
   stbi__int16 fast_ac[4][1 « FAST_BITS];

// sizes for components, interleaved MCUs
   int img_h_max, img_v_max;
   int img_mcu_x, img_mcu_y;
   int img_mcu_w, img_mcu_h;

// definition of jpeg image component
   struct
   {
      int id;
      int h,v;
      int tq;
      int hd,ha;
      int dc_pred;

      int x,y,w2,h2;
      stbi_uc *data;
      void *raw_data, *raw_coeff;
      stbi_uc *linebuf;
      short   *coeff;   // progressive only
      int      coeff_w, coeff_h; // number of 8x8 coefficient blocks
   } img_comp[4];

   stbi__uint32   code_buffer; // jpeg entropy-coded buffer
   int            code_bits;   // number of valid bits
   unsigned char  marker;      // marker seen while filling entropy buffer
   int            nomore;      // flag if we saw a marker so must stop

   int            progressive;
   int            spec_start;
   int            spec_end;
   int            succ_high;
   int            succ_low;
   int            eob_run;
   int            jfif;
   int            app14_color_transform; // Adobe APP14 tag
   int            rgb;

   int scan_n, order[4];
   int restart_interval, todo;

// kernels
   void (*idct_block_kernel)(stbi_uc *out, int out_stride, short data[64]);
   void (*YCbCr_to_RGB_kernel)(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc *pcr, int count, int step);
   stbi_uc *(*resample_row_hv_2_kernel)(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs);
} stbi__jpeg;

static int stbi__build_huffman(stbi__huffman *h, int *count)
{
   int i,j,k=0;
   unsigned int code;
   // build size list for each symbol (from JPEG spec)
   for (i=0; i < 16; ++i) {
      for (j=0; j < count[i]; ++j) {
         h->size[k++] = (stbi_uc) (i+1);
         if(k >= 257) return stbi__err("bad size list","Corrupt JPEG");
      }
   }
   h->size[k] = 0;

   // compute actual symbols (from jpeg spec)
   code = 0;
   k = 0;
   for(j=1; j <= 16; ++j) {
      // compute delta to add to code to compute symbol id
      h->delta[j] = k - code;
      if (h->size[k] == j) {
         while (h->size[k] == j)
            h->code[k++] = (stbi__uint16) (code++);
         if (code-1 >= (1u « j)) return stbi__err("bad code lengths","Corrupt JPEG");
      }
      // compute largest code + 1 for this size, preshifted as needed later
      h->maxcode[j] = code « (16-j);
      code «= 1;
   }
   h->maxcode[j] = 0xffffffff;

   // build non-spec acceleration table; 255 is flag for not-accelerated
   memset(h->fast, 255, 1 « FAST_BITS);
   for (i=0; i < k; ++i) {
      int s = h->size[i];
      if (s <= FAST_BITS) {
         int c = h->code[i] « (FAST_BITS-s);
         int m = 1 « (FAST_BITS-s);
         for (j=0; j < m; ++j) {
            h->fast[c+j] = (stbi_uc) i;
         }
      }
   }
   return 1;
}

// build a table that decodes both magnitude and value of small ACs in
// one go.
static void stbi__build_fast_ac(stbi__int16 *fast_ac, stbi__huffman *h)
{
   int i;
   for (i=0; i < (1 « FAST_BITS); ++i) {
      stbi_uc fast = h->fast[i];
      fast_ac[i] = 0;
      if (fast < 255) {
         int rs = h->values[fast];
         int run = (rs » 4) & 15;
         int magbits = rs & 15;
         int len = h->size[fast];

         if (magbits && len + magbits <= FAST_BITS) {
            // magnitude code followed by receive_extend code
            int k = ((i « len) & ((1 « FAST_BITS) - 1)) » (FAST_BITS - magbits);
            int m = 1 « (magbits - 1);
            if (k < m) k += (~0U « magbits) + 1;
            // if the result is small enough, we can fit it in fast_ac table
            if (k >= -128 && k <= 127)
               fast_ac[i] = (stbi__int16) ((k * 256) + (run * 16) + (len + magbits));
         }
      }
   }
}

static void stbi__grow_buffer_unsafe(stbi__jpeg *j)
{
   do {
      unsigned int b = j->nomore ? 0 : stbi__get8(j->s);
      if (b == 0xff) {
         int c = stbi__get8(j->s);
         while (c == 0xff) c = stbi__get8(j->s); // consume fill bytes
         if (c != 0) {
            j->marker = (unsigned char) c;
            j->nomore = 1;
            return;
         }
      }
      j->code_buffer |= b « (24 - j->code_bits);
      j->code_bits += 8;
   } while (j->code_bits <= 24);
}

// (1 « n) - 1
static const stbi__uint32
stbi__bmask[17]={0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535};

// decode a jpeg huffman value from the bitstream
stbi_inline static int stbi__jpeg_huff_decode(stbi__jpeg *j, stbi__huffman *h)
{
   unsigned int temp;
   int c,k;

   if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);

   // look at the top FAST_BITS and determine what symbol ID it is,
   // if the code is <= FAST_BITS
   c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS)-1);
   k = h->fast[c];
   if (k < 255) {
      int s = h->size[k];
      if (s > j->code_bits)
         return -1;
      j->code_buffer «= s;
      j->code_bits -= s;
      return h->values[k];
   }

   // naive test is to shift the code_buffer down so k bits are
   // valid, then test against maxcode. To speed this up, we've
   // preshifted maxcode left so that it has (16-k) 0s at the
   // end; in other words, regardless of the number of bits, it
   // wants to be compared against something shifted to have 16;
   // that way we don't need to shift inside the loop.
   temp = j->code_buffer » 16;
   for (k=FAST_BITS+1 ; ; ++k)
      if (temp < h->maxcode[k])
         break;
   if (k == 17) {
      // error! code not found
      j->code_bits -= 16;
      return -1;
   }

   if (k > j->code_bits)
      return -1;

   // convert the huffman code to the symbol id
   c = ((j->code_buffer » (32 - k)) & stbi__bmask[k]) + h->delta[k];
   if(c < 0 || c >= 256) // symbol id out of bounds!
       return -1;
   STBI_ASSERT((((j->code_buffer) » (32 - h->size[c])) & stbi__bmask[h->size[c]]) == h->code[c]);

   // convert the id to a symbol
   j->code_bits -= k;
   j->code_buffer «= k;
   return h->values[c];
}

// bias[n] = (-1«n) + 1
static const int stbi__jbias[16] = {0,-1,-3,-7,-15,-31,-63,-127,-255,-511,-1023,-2047,-4095,-8191,-16383,-32767};

// combined JPEG 'receive' and JPEG 'extend', since baseline
// always extends everything it receives.
stbi_inline static int stbi__extend_receive(stbi__jpeg *j, int n)
{
   unsigned int k;
   int sgn;
   if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
   if (j->code_bits < n) return 0; // ran out of bits from stream, return 0s intead of continuing

   sgn = j->code_buffer » 31; // sign bit always in MSB; 0 if MSB clear (positive), 1 if MSB set (negative)
   k = stbi_lrot(j->code_buffer, n);
   j->code_buffer = k & ~stbi__bmask[n];
   k &= stbi__bmask[n];
   j->code_bits -= n;
   return k + (stbi__jbias[n] & (sgn - 1));
}

// get some unsigned bits
stbi_inline static int stbi__jpeg_get_bits(stbi__jpeg *j, int n)
{
   unsigned int k;
   if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
   if (j->code_bits < n) return 0; // ran out of bits from stream, return 0s intead of continuing
   k = stbi_lrot(j->code_buffer, n);
   j->code_buffer = k & ~stbi__bmask[n];
   k &= stbi__bmask[n];
   j->code_bits -= n;
   return k;
}

stbi_inline static int stbi__jpeg_get_bit(stbi__jpeg *j)
{
   unsigned int k;
   if (j->code_bits < 1) stbi__grow_buffer_unsafe(j);
   if (j->code_bits < 1) return 0; // ran out of bits from stream, return 0s intead of continuing
   k = j->code_buffer;
   j->code_buffer «= 1;
   --j->code_bits;
   return k & 0x80000000;
```

```
}

// given a value that’s at position X in the zigzag stream,
// where does it appear in the 8x8 matrix coded as row-major?
static const stbi_uc stbi__jpeg_dezigzag[64+15] =
{
    0,  1,  8, 16,  9,  2,  3, 10,
   17, 24, 32, 25, 18, 11,  4,  5,
   12, 19, 26, 33, 40, 48, 41, 34,
   27, 20, 13,  6,  7, 14, 21, 28,
   35, 42, 49, 56, 57, 50, 43, 36,
   29, 22, 15, 23, 30, 37, 44, 51,
   58, 59, 52, 45, 38, 31, 39, 46,
   53, 60, 61, 54, 47, 55, 62, 63,
   // let corrupt input sample past end
   63, 63, 63, 63, 63, 63, 63, 63,
   63, 63, 63, 63, 63, 63, 63
};

// decode one 64-entry block--
static int stbi__jpeg_decode_block(stbi__jpeg *j, short data[64], stbi__huffman *hdc, stbi__huffman *hac, stbi__int16 *fac, int b, stbi__uint16 *dequant)
{
   int diff,dc,k;
   int t;

   if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
   t = stbi__jpeg_huff_decode(j, hdc);
   if (t < 0 || t > 15) return stbi__err("bad huffman code","Corrupt JPEG");

   // 0 all the ac values now so we can do it 32-bits at a time
   memset(data,0,64*sizeof(data[0]));

   diff = t ? stbi__extend_receive(j, t) : 0;
   if (!stbi__addints_valid(j->img_comp[b].dc_pred, diff)) return stbi__err("bad delta","Corrupt JPEG");
   dc = j->img_comp[b].dc_pred + diff;
   j->img_comp[b].dc_pred = dc;
   if (!stbi__mul2shorts_valid(dc, dequant[0])) return stbi__err("can’t merge dc and ac", "Corrupt JPEG");
   data[0] = (short) (dc * dequant[0]);

   // decode AC components, see JPEG spec
   k = 1;
   do {
      unsigned int zig;
      int c,r,s;
      if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
      c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS)-1);
      r = fac[c];
      if (r) { // fast-AC path
         k += (r » 4) & 15; // run
         s = r & 15; // combined length
         if (s > j->code_bits) return stbi__err("bad huffman code", "Combined length longer than code bits available");
         j->code_buffer «= s;
         j->code_bits -= s;
         // decode into unzigzag’d location
         zig = stbi__jpeg_dezigzag[k++];
         data[zig] = (short) ((r » 8) * dequant[zig]);
      } else {
         int rs = stbi__jpeg_huff_decode(j, hac);
         if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
         s = rs & 15;
         r = rs » 4;
         if (s == 0) {
            if (rs != 0xf0) break; // end block
            k += 16;
         } else {
            k += r;
            // decode into unzigzag’d location
            zig = stbi__jpeg_dezigzag[k++];
            data[zig] = (short) (stbi__extend_receive(j,s) * dequant[zig]);
         }
      }
   } while (k < 64);
   return 1;
}

static int stbi__jpeg_decode_block_prog_dc(stbi__jpeg *j, short data[64], stbi__huffman *hdc, int b)
{
   int diff,dc;
   int t;
   if (j->spec_end != 0) return stbi__err("can’t merge dc and ac", "Corrupt JPEG");

   if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);

   if (j->succ_high == 0) {
      // first scan for DC coefficient, must be first
      memset(data,0,64*sizeof(data[0])); // 0 all the ac values now
      t = stbi__jpeg_huff_decode(j, hdc);
      if (t < 0 || t > 15) return stbi__err("can’t merge dc and ac", "Corrupt JPEG");
      diff = t ? stbi__extend_receive(j, t) : 0;

      if (!stbi__addints_valid(j->img_comp[b].dc_pred, diff)) return stbi__err("bad delta", "Corrupt JPEG");
      dc = j->img_comp[b].dc_pred + diff;
      j->img_comp[b].dc_pred = dc;
      if (!stbi__mul2shorts_valid(dc, 1 « j->succ_low)) return stbi__err("can’t merge dc and ac", "Corrupt JPEG");
      data[0] = (short) (dc * (1 « j->succ_low));
   } else {
      // refinement scan for DC coefficient
      if (stbi__jpeg_get_bit(j))
         data[0] += (short) (1 « j->succ_low);
   }
   return 1;
}

// @OPTIMIZE: store non-zigzagged during the decode passes,
// and only de-zigzag when dequantizing
static int stbi__jpeg_decode_block_prog_ac(stbi__jpeg *j, short data[64], stbi__huffman *hac, stbi__int16 *fac)
{
   int k;
   if (j->spec_start == 0) return stbi__err("can’t merge dc and ac", "Corrupt JPEG");

   if (j->succ_high == 0) {
      int shift = j->succ_low;

      if (j->eob_run) {
         --j->eob_run;
         return 1;
      }

      k = j->spec_start;
      do {
         unsigned int zig;
         int c,r,s;
         if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
         c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS)-1);
         r = fac[c];
         if (r) { // fast-AC path
            k += (r » 4) & 15; // run
            s = r & 15; // combined length
            if (s > j->code_bits) return stbi__err("bad huffman code", "Combined length longer than code bits available");
            j->code_buffer «= s;
            j->code_bits -= s;
            zig = stbi__jpeg_dezigzag[k++];
            data[zig] = (short) ((r » 8) * (1 « shift));
         } else {
            int rs = stbi__jpeg_huff_decode(j, hac);
            if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
            s = rs & 15;
            r = rs » 4;
            if (s == 0) {
               if (r < 15) {
                  j->eob_run = (1 « r);
                  if (r)
                     j->eob_run += stbi__jpeg_get_bits(j, r);
                  --j->eob_run;
                  break;
               }
               k += 16;
            } else {
               k += r;
               zig = stbi__jpeg_dezigzag[k++];
               data[zig] = (short) (stbi__extend_receive(j,s) * (1 « shift));
            }
         }
      } while (k <= j->spec_end);
   } else {
      // refinement scan for these AC coefficients

      short bit = (short) (1 « j->succ_low);

      if (j->eob_run) {
         --j->eob_run;
         for (k = j->spec_start; k <= j->spec_end; ++k) {
            short *p = &data[stbi__jpeg_dezigzag[k]];
            if (*p != 0)
               if (stbi__jpeg_get_bit(j))
                  if ((*p & bit)==0) {
                     if (*p > 0)
                        *p += bit;
                     else
                        *p -= bit;
                  }
         }
      } else {
         k = j->spec_start;
         do {
            int r,s;
            int rs = stbi__jpeg_huff_decode(j, hac); // @OPTIMIZE see if we can use the fast path
here, advance-by-r is so slow, eh
            if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
            s = rs & 15;
            r = rs » 4;
            if (s == 0) {
               if (r < 15) {
                  j->eob_run = (1 « r) - 1;
                  if (r)
                     j->eob_run += stbi__jpeg_get_bits(j, r);
                  r = 64; // force end of block
               } else {
                  // r=15 s=0 should write 16 0s, so we just do
                  // a run of 15 0s and then write s (which is 0),
                  // so we don't have to do anything special here
               }
            } else {
               if (s != 1) return stbi__err("bad huffman code", "Corrupt JPEG");
               // sign bit
               if (stbi__jpeg_get_bit(j))
                  s = bit;
               else
                  s = -bit;
            }

            // advance by r
            while (k <= j->spec_end) {
               short *p = &data[stbi__jpeg_dezigzag[k++]];
               if (*p != 0) {
                  if (stbi__jpeg_get_bit(j))
                     if ((*p & bit)==0) {
                        if (*p > 0)
                           *p += bit;
                        else
                           *p -= bit;
                     }
               } else {
                  if (r == 0) {
                     *p = (short) s;
                     break;
                  }
                  --r;
               }
            }
         } while (k <= j->spec_end);
      }
   }
   return 1;
}

// take a -128..127 value and stbi__clamp it and convert to 0..255
stbi_inline static stbi_uc stbi__clamp(int x)
{
   // trick to use a single test to catch both cases
   if ((unsigned int) x > 255) {
      if (x < 0) return 0;
      if (x > 255) return 255;
   }
   return (stbi_uc) x;
}

#define stbi__f2f(x)  ((int) (((x) * 4096 + 0.5)))
#define stbi__fsh(x)  ((x) * 4096)

// derived from jidctint -- DCT_ISLOW
#define STBI__IDCT_1D(s0,s1,s2,s3,s4,s5,s6,s7) \
   int t0,t1,t2,t3,p1,p2,p3,p4,p5,x0,x1,x2,x3; \
   p2 = s2;                                    \
   p3 = s6;                                    \
   p1 = (p2+p3) * stbi__f2f(0.5411961f);       \
   t2 = p1 + p3*stbi__f2f(-1.847759065f);      \
   t3 = p1 + p2*stbi__f2f( 0.765366865f);      \
   p2 = s0;                                    \
   p3 = s4;                                    \
   t0 = stbi__fsh(p2+p3);                      \
   t1 = stbi__fsh(p2-p3);                      \
   x0 = t0+t3;                                 \
   x3 = t0-t3;                                 \
   x1 = t1+t2;                                 \
   x2 = t1-t2;                                 \
   t0 = s7;                                    \
   t1 = s5;                                    \
   t2 = s3;                                    \
   t3 = s1;                                    \
   p3 = t0+t2;                                 \
   p4 = t1+t3;                                 \
   p1 = t0+t3;                                 \
   p2 = t1+t2;                                 \
   p5 = (p3+p4)*stbi__f2f( 1.175875602f);      \
   t0 = t0*stbi__f2f( 0.298631336f);           \
   t1 = t1*stbi__f2f( 2.053119869f);           \
   t2 = t2*stbi__f2f( 3.072711026f);           \
   t3 = t3*stbi__f2f( 1.501321110f);           \
   p1 = p5 + p1*stbi__f2f(-0.899976223f);      \
   p2 = p5 + p2*stbi__f2f(-2.562915447f);      \
   p3 = p3*stbi__f2f(-1.961570560f);           \
   p4 = p4*stbi__f2f(-0.390180644f);           \
   t3 += p1+p4;                                \
   t2 += p2+p3;                                \
   t1 += p2+p4;                                \
   t0 += p1+p3;

static void stbi__idct_block(stbi_uc *out, int out_stride, short data[64])
{
   int i,val[64],*v=val;
   stbi_uc *o;
   short *d = data;

   // columns
   for (i=0; i < 8; ++i,++d, ++v) {
      // if all zeroes, shortcut -- this avoids dequantizing 0s and IDCTing
      if (d[ 8]==0 && d[16]==0 && d[24]==0 && d[32]==0
           && d[40]==0 && d[48]==0 && d[56]==0) {
         //    no shortcut                 0     seconds
         //    (1|2|3|4|5|6|7)==0          0     seconds
         //    all separate               -0.047 seconds
         //    1 && 2|3 && 4|5 && 6|7:    -0.047 seconds
         int dcterm = d[0]*4;
         v[0] = v[8] = v[16] = v[24] = v[32] = v[40] = v[48] = v[56] = dcterm;
      } else {
         STBI__IDCT_1D(d[ 0],d[ 8],d[16],d[24],d[32],d[40],d[48],d[56])
         // constants scaled things up by 1«12; let’s bring them back
         // down, but keep 2 extra bits of precision
         x0 += 512; x1 += 512; x2 += 512; x3 += 512;
         v[ 0] = (x0+t3) » 10;
         v[56] = (x0-t3) » 10;
         v[ 8] = (x1+t2) » 10;
         v[48] = (x1-t2) » 10;
         v[16] = (x2+t1) » 10;
         v[40] = (x2-t1) » 10;
         v[24] = (x3+t0) » 10;
         v[32] = (x3-t0) » 10;
      }
   }

   for (i=0, v=val, o=out; i < 8; ++i,v+=8,o+=out_stride) {
      // no fast case since the first 1D IDCT spread components out
      STBI__IDCT_1D(v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7])
      // constants scaled things up by 1«12, plus we had 1«2 from first
      // loop, plus horizontal and vertical each scale by sqrt(8) so together
      // we’ve got an extra 1«3, so 1«17 total we need to remove.
      // so we want to round that, which means adding 0.5 * 1«17,
      // aka 65536. Also, we’ll end up with -128 to 127 that we want
      // to encode as 0..255 by adding 128, so we’ll add that before the shift
      x0 += 65536 + (128«17);
      x1 += 65536 + (128«17);
      x2 += 65536 + (128«17);
      x3 += 65536 + (128«17);
      // tried computing the shifts into temps, or’ing the temps to see
      // if any were out of range, but that was slower
      o[0] = stbi__clamp((x0+t3) » 17);
      o[7] = stbi__clamp((x0-t3) » 17);
      o[1] = stbi__clamp((x1+t2) » 17);
      o[6] = stbi__clamp((x1-t2) » 17);
      o[2] = stbi__clamp((x2+t1) » 17);
      o[5] = stbi__clamp((x2-t1) » 17);
      o[3] = stbi__clamp((x3+t0) » 17);
      o[4] = stbi__clamp((x3-t0) » 17);
   }
}

#ifdef STBI_SSE2
// sse2 integer IDCT. not the fastest possible implementation but it
// produces bit-identical results to the generic C version so it’s
// fully "transparent".
static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
{
   // This is constructed to match our regular (generic) integer IDCT exactly.
   __m128i row0, row1, row2, row3, row4, row5, row6, row7;
   __m128i tmp;

   // dot product constant: even elems=x, odd elems=y
   #define dct_const(x,y)  _mm_setr_epi16((x),(y),(x),(y),(x),(y),(x),(y))

   // out(0) = c0[even]*x + c0[odd]*y   (c0, x, y 16-bit, out 32-bit)
   // out(1) = c1[even]*x + c1[odd]*y
   #define dct_rot(out0,out1, x,y,c0,c1) \
      __m128i c0##lo = _mm_unpacklo_epi16((x),(y)); \
      __m128i c0##hi = _mm_unpackhi_epi16((x),(y)); \
      __m128i out0##_l = _mm_madd_epi16(c0##lo, c0); \
      __m128i out0##_h = _mm_madd_epi16(c0##hi, c0); \
      __m128i out1##_l = _mm_madd_epi16(c0##lo, c1); \
      __m128i out1##_h = _mm_madd_epi16(c0##hi, c1)

   // out = in « 12  (in 16-bit, out 32-bit)
   #define dct_widen(out, in) \
      __m128i out##_l = _mm_srai_epi32(_mm_unpacklo_epi16(_mm_setzero_si128(), (in)), 4); \
      __m128i out##_h = _mm_srai_epi32(_mm_unpackhi_epi16(_mm_setzero_si128(), (in)), 4)

   // wide add
   #define dct_wadd(out, a, b) \
      __m128i out##_l = _mm_add_epi32(a##_l, b##_l); \
      __m128i out##_h = _mm_add_epi32(a##_h, b##_h)

   // wide sub
   #define dct_wsub(out, a, b) \
      __m128i out##_l = _mm_sub_epi32(a##_l, b##_l); \
      __m128i out##_h = _mm_sub_epi32(a##_h, b##_h)

   // butterfly a/b, add bias, then shift by "s" and pack
   #define dct_bfly32o(out0, out1, a,b,bias,s) \
      { \
         __m128i abiased_l = _mm_add_epi32(a##_l, bias); \
         __m128i abiased_h = _mm_add_epi32(a##_h, bias); \
         dct_wadd(sum, abiased, b); \
         dct_wsub(dif, abiased, b); \
         out0 = _mm_packs_epi32(_mm_srai_epi32(sum_l, s), _mm_srai_epi32(sum_h, s)); \
         out1 = _mm_packs_epi32(_mm_srai_epi32(dif_l, s), _mm_srai_epi32(dif_h, s)); \
      }

   // 8-bit interleave step (for transposes)
   #define dct_interleave8(a, b) \
      tmp = a; \
      a = _mm_unpacklo_epi8(a, b); \
      b = _mm_unpackhi_epi8(tmp, b)

   // 16-bit interleave step (for transposes)
   #define dct_interleave16(a, b) \
      tmp = a; \
      a = _mm_unpacklo_epi16(a, b); \
      b = _mm_unpackhi_epi16(tmp, b)

   #define dct_pass(bias,shift) \
      { \
         /* even part */ \
         dct_rot(t2e,t3e, row2,row6, rot0_0,rot0_1); \
         __m128i sum04 = _mm_add_epi16(row0, row4); \
         __m128i dif04 = _mm_sub_epi16(row0, row4); \
         dct_widen(t0e, sum04); \
         dct_widen(t1e, dif04); \
         dct_wadd(x0, t0e, t3e); \
         dct_wsub(x3, t0e, t3e); \
         dct_wadd(x1, t1e, t2e); \
         dct_wsub(x2, t1e, t2e); \
         /* odd part */ \
         dct_rot(y0o,y2o, row7,row3, rot2_0,rot2_1); \
         dct_rot(y1o,y3o, row5,row1, rot3_0,rot3_1); \
         __m128i sum17 = _mm_add_epi16(row1, row7); \
         __m128i sum35 = _mm_add_epi16(row3, row5); \
         dct_rot(y4o,y5o, sum17,sum35, rot1_0,rot1_1); \
         dct_wadd(x4, y0o, y4o); \
         dct_wadd(x5, y1o, y5o); \
         dct_wadd(x6, y2o, y5o); \
         dct_wadd(x7, y3o, y4o); \
         dct_bfly32o(row0,row7, x0,x7,bias,shift); \
         dct_bfly32o(row1,row6, x1,x6,bias,shift); \
         dct_bfly32o(row2,row5, x2,x5,bias,shift); \
         dct_bfly32o(row3,row4, x3,x4,bias,shift); \
      }

   __m128i rot0_0 = dct_const(stbi__f2f(0.5411961f), stbi__f2f(0.5411961f) +
stbi__f2f(-1.847759065f));
   __m128i rot0_1 = dct_const(stbi__f2f(0.5411961f) + stbi__f2f( 0.765366865f),
stbi__f2f(0.5411961f));
   __m128i rot1_0 = dct_const(stbi__f2f(1.175875602f) + stbi__f2f(-0.899976223f),
stbi__f2f(1.175875602f));
   __m128i rot1_1 = dct_const(stbi__f2f(1.175875602f), stbi__f2f(1.175875602f) +
stbi__f2f(-2.562915447f));
   __m128i rot2_0 = dct_const(stbi__f2f(-1.961570560f) + stbi__f2f( 0.298631336f),
stbi__f2f(-1.961570560f));
   __m128i rot2_1 = dct_const(stbi__f2f(-1.961570560f), stbi__f2f(-1.961570560f) + stbi__f2f(
3.072711026f));
   __m128i rot3_0 = dct_const(stbi__f2f(-0.390180644f) + stbi__f2f( 2.053119869f),
stbi__f2f(-0.390180644f));
   __m128i rot3_1 = dct_const(stbi__f2f(-0.390180644f), stbi__f2f(-0.390180644f) + stbi__f2f(
1.501321110f));

   // rounding biases in column/row passes, see stbi__idct_block for explanation.
   __m128i bias_0 = _mm_set1_epi32(512);
   __m128i bias_1 = _mm_set1_epi32(65536 + (128«17));

   // load
   row0 = _mm_load_si128((const __m128i *) (data + 0*8));
   row1 = _mm_load_si128((const __m128i *) (data + 1*8));
   row2 = _mm_load_si128((const __m128i *) (data + 2*8));
   row3 = _mm_load_si128((const __m128i *) (data + 3*8));
   row4 = _mm_load_si128((const __m128i *) (data + 4*8));
   row5 = _mm_load_si128((const __m128i *) (data + 5*8));
   row6 = _mm_load_si128((const __m128i *) (data + 6*8));
   row7 = _mm_load_si128((const __m128i *) (data + 7*8));

   // column pass
   dct_pass(bias_0, 10);

   {
      // 16bit 8x8 transpose pass 1
      dct_interleave16(row0, row4);
      dct_interleave16(row1, row5);
      dct_interleave16(row2, row6);
      dct_interleave16(row3, row7);

      // transpose pass 2
      dct_interleave16(row0, row2);
      dct_interleave16(row1, row3);
      dct_interleave16(row4, row6);
      dct_interleave16(row5, row7);

      // transpose pass 3
      dct_interleave16(row0, row1);
      dct_interleave16(row2, row3);
      dct_interleave16(row4, row5);
      dct_interleave16(row6, row7);
   }

   // row pass
   dct_pass(bias_1, 17);

   {
      // pack
      __m128i p0 = _mm_packus_epi16(row0, row1); // a0a1a2a3...a7b0b1b2b3...b7
      __m128i p1 = _mm_packus_epi16(row2, row3);
      __m128i p2 = _mm_packus_epi16(row4, row5);
      __m128i p3 = _mm_packus_epi16(row6, row7);

      // 8bit 8x8 transpose pass 1
      dct_interleave8(p0, p2); // a0e0a1e1...
      dct_interleave8(p1, p3); // c0g0c1g1...

      // transpose pass 2
      dct_interleave8(p0, p1); // a0c0e0g0...
      dct_interleave8(p2, p3); // b0d0f0h0...

      // transpose pass 3
      dct_interleave8(p0, p2); // a0b0c0d0...
      dct_interleave8(p1, p3); // a4b4c4d4...

      // store
      _mm_storel_epi64((__m128i *) out, p0); out += out_stride;
      _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p0, 0x4e)); out += out_stride;
      _mm_storel_epi64((__m128i *) out, p2); out += out_stride;
      _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p2, 0x4e)); out += out_stride;
      _mm_storel_epi64((__m128i *) out, p1); out += out_stride;
      _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p1, 0x4e)); out += out_stride;
      _mm_storel_epi64((__m128i *) out, p3); out += out_stride;
      _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p3, 0x4e));
   }

#undef dct_const
```

```
#undef dct_rot
#undef dct_widen
#undef dct_wadd
#undef dct_wsub
#undef dct_bfly32o
#undef dct_interleave8
#undef dct_interleave16
#undef dct_pass
}

#endif // STBI_SSE2

#ifdef STBI_NEON

// NEON integer IDCT. should produce bit-identical
// results to the generic C version.
static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
{
   int16x8_t row0, row1, row2, row3, row4, row5, row6, row7;

   int16x4_t rot0_0 = vdup_n_s16(stbi__f2f(0.5411961f));
   int16x4_t rot0_1 = vdup_n_s16(stbi__f2f(-1.847759065f));
   int16x4_t rot0_2 = vdup_n_s16(stbi__f2f( 0.765366865f));
   int16x4_t rot1_0 = vdup_n_s16(stbi__f2f( 1.175875602f));
   int16x4_t rot1_1 = vdup_n_s16(stbi__f2f(-0.899976223f));
   int16x4_t rot1_2 = vdup_n_s16(stbi__f2f(-2.562915447f));
   int16x4_t rot2_0 = vdup_n_s16(stbi__f2f(-1.961570560f));
   int16x4_t rot2_1 = vdup_n_s16(stbi__f2f(-0.390180644f));
   int16x4_t rot3_0 = vdup_n_s16(stbi__f2f( 0.298631336f));
   int16x4_t rot3_1 = vdup_n_s16(stbi__f2f( 2.053119869f));
   int16x4_t rot3_2 = vdup_n_s16(stbi__f2f( 3.072711026f));
   int16x4_t rot3_3 = vdup_n_s16(stbi__f2f( 1.501321110f));

#define dct_long_mul(out, inq, coeff) \
   int32x4_t out##_l = vmull_s16(vget_low_s16(inq), coeff); \
   int32x4_t out##_h = vmull_s16(vget_high_s16(inq), coeff)

#define dct_long_mac(out, acc, inq, coeff) \
   int32x4_t out##_l = vmlal_s16(acc##_l, vget_low_s16(inq), coeff); \
   int32x4_t out##_h = vmlal_s16(acc##_h, vget_high_s16(inq), coeff)

#define dct_widen(out, inq) \
   int32x4_t out##_l = vshll_n_s16(vget_low_s16(inq), 12); \
   int32x4_t out##_h = vshll_n_s16(vget_high_s16(inq), 12)

// wide add
#define dct_wadd(out, a, b) \
   int32x4_t out##_l = vaddq_s32(a##_l, b##_l); \
   int32x4_t out##_h = vaddq_s32(a##_h, b##_h)

// wide sub
#define dct_wsub(out, a, b) \
   int32x4_t out##_l = vsubq_s32(a##_l, b##_l); \
   int32x4_t out##_h = vsubq_s32(a##_h, b##_h)

// butterfly a/b, then shift using "shiftop" by "s" and pack
#define dct_bfly32o(out0,out1, a,b,shiftop,s) \
   { \
      dct_wadd(sum, a, b); \
      dct_wsub(dif, a, b); \
      out0 = vcombine_s16(shiftop(sum_l, s), shiftop(sum_h, s)); \
      out1 = vcombine_s16(shiftop(dif_l, s), shiftop(dif_h, s)); \
   }

#define dct_pass(shiftop, shift) \
   { \
      /* even part */ \
      int16x8_t sum26 = vaddq_s16(row2, row6); \
      dct_long_mul(p1e, sum26, rot0_0); \
      dct_long_mac(t2e, p1e, row6, rot0_1); \
      dct_long_mac(t3e, p1e, row2, rot0_2); \
      int16x8_t sum04 = vaddq_s16(row0, row4); \
      int16x8_t dif04 = vsubq_s16(row0, row4); \
      dct_widen(t0e, sum04); \
      dct_widen(t1e, dif04); \
      dct_wadd(x0, t0e, t3e); \
      dct_wsub(x3, t0e, t3e); \
      dct_wadd(x1, t1e, t2e); \
      dct_wsub(x2, t1e, t2e); \
      /* odd part */ \
      int16x8_t sum15 = vaddq_s16(row1, row5); \
      int16x8_t sum17 = vaddq_s16(row1, row7); \
      int16x8_t sum35 = vaddq_s16(row3, row5); \
      int16x8_t sum37 = vaddq_s16(row3, row7); \
      int16x8_t sumodd = vaddq_s16(sum17, sum35); \
      dct_long_mul(p5o, sumodd, rot1_0); \
      dct_long_mac(p1o, p5o, sum17, rot1_1); \
      dct_long_mac(p2o, p5o, sum35, rot1_2); \
      dct_long_mul(p3o, sum37, rot2_0); \
      dct_long_mul(p4o, sum15, rot2_1); \
      dct_wadd(sump13o, p1o, p3o); \
      dct_wadd(sump24o, p2o, p4o); \
      dct_wadd(sump23o, p2o, p3o); \
      dct_wadd(sump14o, p1o, p4o); \
      dct_long_mac(x4, sump13o, row7, rot3_0); \
      dct_long_mac(x5, sump24o, row5, rot3_1); \
      dct_long_mac(x6, sump23o, row3, rot3_2); \
      dct_long_mac(x7, sump14o, row1, rot3_3); \
      dct_bfly32o(row0,row7, x0,x7,shiftop,shift); \
      dct_bfly32o(row1,row6, x1,x6,shiftop,shift); \
      dct_bfly32o(row2,row5, x2,x5,shiftop,shift); \
      dct_bfly32o(row3,row4, x3,x4,shiftop,shift); \
   }

   // load
   row0 = vld1q_s16(data + 0*8);
   row1 = vld1q_s16(data + 1*8);
   row2 = vld1q_s16(data + 2*8);
   row3 = vld1q_s16(data + 3*8);
   row4 = vld1q_s16(data + 4*8);
   row5 = vld1q_s16(data + 5*8);
   row6 = vld1q_s16(data + 6*8);
   row7 = vld1q_s16(data + 7*8);

   // add DC bias
   row0 = vaddq_s16(row0, vsetq_lane_s16(1024, vdupq_n_s16(0), 0));

   // column pass
   dct_pass(vrshrn_n_s32, 10);

   // 16bit 8x8 transpose
   {
// these three map to a single VTRN.16, VTRN.32, and VSWP, respectively.
// whether compilers actually get this is another story, sadly.
#define dct_trn16(x, y) { int16x8x2_t t = vtrnq_s16(x, y); x = t.val[0]; y = t.val[1]; }
#define dct_trn32(x, y) { int32x4x2_t t = vtrnq_s32(vreinterpretq_s32_s16(x), vreinterpretq_s32_s16(y)); x = vreinterpretq_s16_s32(t.val[0]); y = vreinterpretq_s16_s32(t.val[1]); }
#define dct_trn64(x, y) { int16x8_t x0 = x; int16x8_t y0 = y; x = vcombine_s16(vget_low_s16(x0), vget_low_s16(y0)); y = vcombine_s16(vget_high_s16(x0), vget_high_s16(y0)); }

      // pass 1
      dct_trn16(row0, row1); // a0b0a2b2a4b4a6b6
      dct_trn16(row2, row3);
      dct_trn16(row4, row5);
      dct_trn16(row6, row7);

      // pass 2
      dct_trn32(row0, row2); // a0b0c0d0a4b4c4d4
      dct_trn32(row1, row3);
      dct_trn32(row4, row6);
      dct_trn32(row5, row7);

      // pass 3
      dct_trn64(row0, row4); // a0b0c0d0e0f0g0h0
      dct_trn64(row1, row5);
      dct_trn64(row2, row6);
      dct_trn64(row3, row7);

#undef dct_trn16
#undef dct_trn32
#undef dct_trn64
   }

   // row pass
   // vrshrn_n_s32 only supports shifts up to 16, we need
   // 17. so do a non-rounding shift of 16 first then follow
   // up with a rounding shift by 1.
   dct_pass(vshrn_n_s32, 16);

   {
      // pack and round
      uint8x8_t p0 = vqrshrun_n_s16(row0, 1);
      uint8x8_t p1 = vqrshrun_n_s16(row1, 1);
      uint8x8_t p2 = vqrshrun_n_s16(row2, 1);
      uint8x8_t p3 = vqrshrun_n_s16(row3, 1);
      uint8x8_t p4 = vqrshrun_n_s16(row4, 1);
      uint8x8_t p5 = vqrshrun_n_s16(row5, 1);
      uint8x8_t p6 = vqrshrun_n_s16(row6, 1);
      uint8x8_t p7 = vqrshrun_n_s16(row7, 1);

      // again, these can translate into one instruction, but often don't.
#define dct_trn8_8(x, y) { uint8x8x2_t t = vtrn_u8(x, y); x = t.val[0]; y = t.val[1]; }
#define dct_trn8_16(x, y) { uint16x4x2_t t = vtrn_u16(vreinterpret_u16_u8(x), vreinterpret_u16_u8(y)); x = vreinterpret_u8_u16(t.val[0]); y = vreinterpret_u8_u16(t.val[1]); }
```

```
#define dct_trn8_32(x, y) { uint32x2x2_t t = vtrn_u32(vreinterpret_u32_u8(x), vreinterpret_u32_u8(y)); x = vreinterpret_u8_u32(t.val[0]); y = vreinterpret_u8_u32(t.val[1]); }

      // sadly can't use interleaved stores here since we only write
      // 8 bytes to each scan line!

      // 8x8 8-bit transpose pass 1
      dct_trn8_8(p0, p1);
      dct_trn8_8(p2, p3);
      dct_trn8_8(p4, p5);
      dct_trn8_8(p6, p7);

      // pass 2
      dct_trn8_16(p0, p2);
      dct_trn8_16(p1, p3);
      dct_trn8_16(p4, p6);
      dct_trn8_16(p5, p7);

      // pass 3
      dct_trn8_32(p0, p4);
      dct_trn8_32(p1, p5);
      dct_trn8_32(p2, p6);
      dct_trn8_32(p3, p7);

      // store
      vst1_u8(out, p0); out += out_stride;
      vst1_u8(out, p1); out += out_stride;
      vst1_u8(out, p2); out += out_stride;
      vst1_u8(out, p3); out += out_stride;
      vst1_u8(out, p4); out += out_stride;
      vst1_u8(out, p5); out += out_stride;
      vst1_u8(out, p6); out += out_stride;
      vst1_u8(out, p7);

#undef dct_trn8_8
#undef dct_trn8_16
#undef dct_trn8_32
   }

#undef dct_long_mul
#undef dct_long_mac
#undef dct_widen
#undef dct_wadd
#undef dct_wsub
#undef dct_bfly32o
#undef dct_pass
}

#endif // STBI_NEON

#define STBI__MARKER_none  0xff
// if there's a pending marker from the entropy stream, return that
// otherwise, fetch from the stream and get a marker. if there's no
// marker, return 0xff, which is never a valid marker value
static stbi_uc stbi__get_marker(stbi__jpeg *j)
{
   stbi_uc x;
   if (j->marker != STBI__MARKER_none) { x = j->marker; j->marker = STBI__MARKER_none; return x; }
   x = stbi__get8(j->s);
   if (x != 0xff) return STBI__MARKER_none;
   while (x == 0xff)
      x = stbi__get8(j->s); // consume repeated 0xff fill bytes
   return x;
}

// in each scan, we'll have scan_n components, and the order
// of the components is specified by order[]
#define STBI__RESTART(x)     ((x) >= 0xd0 && (x) <= 0xd7)

// after a restart interval, stbi__jpeg_reset the entropy decoder and
// the dc prediction
static void stbi__jpeg_reset(stbi__jpeg *j)
{
   j->code_bits = 0;
   j->code_buffer = 0;
   j->nomore = 0;
   j->img_comp[0].dc_pred = j->img_comp[1].dc_pred = j->img_comp[2].dc_pred = j->img_comp[3].dc_pred = 0;
   j->marker = STBI__MARKER_none;
   j->todo = j->restart_interval ? j->restart_interval : 0x7fffffff;
   j->eob_run = 0;
   // no more than 1«31 MCUs if no restart_interal? that's plenty safe,
   // since we don't even allow 1«30 pixels
}

static int stbi__parse_entropy_coded_data(stbi__jpeg *z)
{
   stbi__jpeg_reset(z);
   if (!z->progressive) {
      if (z->scan_n == 1) {
         int i,j;
         STBI_SIMD_ALIGN(short, data[64]);
         int n = z->order[0];
         // non-interleaved data, we just need to process one block at a time,
         // in trivial scanline order
         // number of blocks to do just depends on how many actual "pixels" this
         // component has, independent of interleaved MCU blocking and such
         int w = (z->img_comp[n].x+7) » 3;
         int h = (z->img_comp[n].y+7) » 3;
         for (j=0; j < h; ++j) {
            for (i=0; i < w; ++i) {
               int ha = z->img_comp[n].ha;
               if (!stbi__jpeg_decode_block(z, data, z->huff_dc+z->img_comp[n].hd, z->huff_ac+ha, z->fast_ac[ha], n, z->dequant[z->img_comp[n].tq])) return 0;
               z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*j*8+i*8, z->img_comp[n].w2, data);
               // every data block is an MCU, so countdown the restart interval
               if (--z->todo <= 0) {
                  if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                  // if it’s NOT a restart, then just bail, so we get corrupt data
                  // rather than no data
                  if (!STBI__RESTART(z->marker)) return 1;
                  stbi__jpeg_reset(z);
               }
            }
         }
         return 1;
      } else { // interleaved
         int i,j,k,x,y;
         STBI_SIMD_ALIGN(short, data[64]);
         for (j=0; j < z->img_mcu_y; ++j) {
            for (i=0; i < z->img_mcu_x; ++i) {
               // scan an interleaved mcu... process scan_n components in order
               for (k=0; k < z->scan_n; ++k) {
                  int n = z->order[k];
                  // scan out an mcu’s worth of this component; that’s just determined
                  // by the basic H and V specified for the component
                  for (y=0; y < z->img_comp[n].v; ++y) {
                     for (x=0; x < z->img_comp[n].h; ++x) {
                        int x2 = (i*z->img_comp[n].h + x)*8;
                        int y2 = (j*z->img_comp[n].v + y)*8;
                        int ha = z->img_comp[n].ha;
                        if (!stbi__jpeg_decode_block(z, data, z->huff_dc+z->img_comp[n].hd, z->huff_ac+ha, z->fast_ac[ha], n, z->dequant[z->img_comp[n].tq])) return 0;
                        z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*y2+x2, z->img_comp[n].w2, data);
                     }
                  }
               }
               // after all interleaved components, that’s an interleaved MCU,
               // so now count down the restart interval
               if (--z->todo <= 0) {
                  if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                  if (!STBI__RESTART(z->marker)) return 1;
                  stbi__jpeg_reset(z);
               }
            }
         }
         return 1;
      }
   } else {
      if (z->scan_n == 1) {
         int i,j;
         int n = z->order[0];
         // non-interleaved data, we just need to process one block at a time,
         // in trivial scanline order
         // number of blocks to do just depends on how many actual "pixels" this
         // component has, independent of interleaved MCU blocking and such
         int w = (z->img_comp[n].x+7) » 3;
         int h = (z->img_comp[n].y+7) » 3;
         for (j=0; j < h; ++j) {
            for (i=0; i < w; ++i) {
               short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
               if (z->spec_start == 0) {
                  if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd], n))
                     return 0;
               } else {
                  int ha = z->img_comp[n].ha;
                  if (!stbi__jpeg_decode_block_prog_ac(z, data, &z->huff_ac[ha], z->fast_ac[ha]))
                     return 0;
               }
               // every data block is an MCU, so countdown the restart interval
               if (--z->todo <= 0) {
                  if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                  if (!STBI__RESTART(z->marker)) return 1;
                  stbi__jpeg_reset(z);
               }
            }
         }
         return 1;
      } else { // interleaved
         int i,j,k,x,y;
         for (j=0; j < z->img_mcu_y; ++j) {
            for (i=0; i < z->img_mcu_x; ++i) {
               // scan an interleaved mcu... process scan_n components in order
               for (k=0; k < z->scan_n; ++k) {
                  int n = z->order[k];
                  // scan out an mcu’s worth of this component; that’s just determined
                  // by the basic H and V specified for the component
                  for (y=0; y < z->img_comp[n].v; ++y) {
                     for (x=0; x < z->img_comp[n].h; ++x) {
                        int x2 = (i*z->img_comp[n].h + x);
                        int y2 = (j*z->img_comp[n].v + y);
                        short *data = z->img_comp[n].coeff + 64 * (x2 + y2 * z->img_comp[n].coeff_w);
                        if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd], n))
                           return 0;
                     }
                  }
               }
               // after all interleaved components, that’s an interleaved MCU,
               // so now count down the restart interval
               if (--z->todo <= 0) {
                  if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                  if (!STBI__RESTART(z->marker)) return 1;
                  stbi__jpeg_reset(z);
               }
            }
         }
         return 1;
      }
   }
}

static void stbi__jpeg_dequantize(short *data, stbi__uint16 *dequant)
{
   int i;
   for (i=0; i < 64; ++i)
      data[i] *= dequant[i];
}

static void stbi__jpeg_finish(stbi__jpeg *z)
{
   if (z->progressive) {
      // dequantize and idct the data
      int i,j,n;
      for (n=0; n < z->s->img_n; ++n) {
         int w = (z->img_comp[n].x+7) » 3;
         int h = (z->img_comp[n].y+7) » 3;
         for (j=0; j < h; ++j) {
            for (i=0; i < w; ++i) {
               short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
               stbi__jpeg_dequantize(data, z->dequant[z->img_comp[n].tq]);
               z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*j*8+i*8, z->img_comp[n].w2, data);
            }
         }
      }
   }
}

static int stbi__process_marker(stbi__jpeg *z, int m)
{
   int L;
   switch (m) {
      case STBI__MARKER_none: // no marker found
         return stbi__err("expected marker","Corrupt JPEG");

      case 0xDD: // DRI - specify restart interval
         if (stbi__get16be(z->s) != 4) return stbi__err("bad DRI len","Corrupt JPEG");
         z->restart_interval = stbi__get16be(z->s);
         return 1;

      case 0xDB: // DQT - define quantization table
         L = stbi__get16be(z->s)-2;
         while (L > 0) {
            int q = stbi__get8(z->s);
            int p = q » 4, sixteen = (p != 0);
            int t = q & 15,i;
            if (p != 0 && p != 1) return stbi__err("bad DQT type","Corrupt JPEG");
            if (t > 3) return stbi__err("bad DQT table","Corrupt JPEG");

            for (i=0; i < 64; ++i)
               z->dequant[t][stbi__jpeg_dezigzag[i]] = (stbi__uint16)(sixteen ? stbi__get16be(z->s) : stbi__get8(z->s));
            L -= (sixteen ? 129 : 65);
         }
         return L==0;

      case 0xC4: // DHT - define huffman table
         L = stbi__get16be(z->s)-2;
         while (L > 0) {
            stbi_uc *v;
            int sizes[16],i,n=0;
            int q = stbi__get8(z->s);
            int tc = q » 4;
            int th = q & 15;
            if (tc > 1 || th > 3) return stbi__err("bad DHT header","Corrupt JPEG");
            for (i=0; i < 16; ++i) {
               sizes[i] = stbi__get8(z->s);
               n += sizes[i];
            }
            if(n > 256) return stbi__err("bad DHT header","Corrupt JPEG"); // Loop over i < n would write past end of values!
            L -= 17;
            if (tc == 0) {
               if (!stbi__build_huffman(z->huff_dc+th, sizes)) return 0;
               v = z->huff_dc[th].values;
            } else {
               if (!stbi__build_huffman(z->huff_ac+th, sizes)) return 0;
               v = z->huff_ac[th].values;
            }
            for (i=0; i < n; ++i)
               v[i] = stbi__get8(z->s);
            if (tc != 0)
               stbi__build_fast_ac(z->fast_ac[th], z->huff_ac + th);
            L -= n;
         }
         return L==0;
   }

   // check for comment block or APP blocks
   if ((m >= 0xE0 && m <= 0xEF) || m == 0xFE) {
      L = stbi__get16be(z->s);
      if (L < 2) {
         if (m == 0xFE)
            return stbi__err("bad COM len","Corrupt JPEG");
         else
            return stbi__err("bad APP len","Corrupt JPEG");
      }
      L -= 2;

      if (m == 0xE0 && L >= 5) { // JFIF APP0 segment
         static const unsigned char tag[5] = {'J','F','I','F','\0'};
         int ok = 1;
         int i;
         for (i=0; i < 5; ++i)
            if (stbi__get8(z->s) != tag[i])
               ok = 0;
         L -= 5;
         if (ok)
            z->jfif = 1;
      } else if (m == 0xEE && L >= 12) { // Adobe APP14 segment
         static const unsigned char tag[6] = {'A','d','o','b','e','\0'};
         int ok = 1;
         int i;
         for (i=0; i < 6; ++i)
            if (stbi__get8(z->s) != tag[i])
               ok = 0;
         L -= 6;
         if (ok) {
            stbi__get8(z->s); // version
            stbi__get16be(z->s); // flags0
            stbi__get16be(z->s); // flags1
            z->app14_color_transform = stbi__get8(z->s); // color transform
            L -= 6;
         }
      }

      stbi__skip(z->s, L);
      return 1;
   }

   return stbi__err("unknown marker","Corrupt JPEG");
}

// after we see SOS
static int stbi__process_scan_header(stbi__jpeg *z)
```

```
{
   int i;
   int Ls = stbi__get16be(z->s);
   z->scan_n = stbi__get8(z->s);
   if (z->scan_n < 1 || z->scan_n > 4 || z->scan_n > (int) z->s->img_n) return stbi__err("bad SOS component count","Corrupt JPEG");
   if (Ls != 6+2*z->scan_n) return stbi__err("bad SOS len","Corrupt JPEG");
   for (i=0; i < z->scan_n; ++i) {
      int id = stbi__get8(z->s), which;
      int q = stbi__get8(z->s);
      for (which = 0; which < z->s->img_n; ++which)
         if (z->img_comp[which].id == id)
            break;
      if (which == z->s->img_n) return 0; // no match
      z->img_comp[which].hd = q » 4;   if (z->img_comp[which].hd > 3) return stbi__err("bad DC huff","Corrupt JPEG");
      z->img_comp[which].ha = q & 15;   if (z->img_comp[which].ha > 3) return stbi__err("bad AC huff","Corrupt JPEG");
      z->order[i] = which;
   }

   {
      int aa;
      z->spec_start = stbi__get8(z->s);
      z->spec_end   = stbi__get8(z->s); // should be 63, but might be 0
      aa = stbi__get8(z->s);
      z->succ_high = (aa » 4);
      z->succ_low  = (aa & 15);
      if (z->progressive) {
         if (z->spec_start > 63 || z->spec_end > 63  || z->spec_start > z->spec_end || z->succ_high > 13 || z->succ_low > 13)
            return stbi__err("bad SOS", "Corrupt JPEG");
      } else {
         if (z->spec_start != 0) return stbi__err("bad SOS","Corrupt JPEG");
         if (z->succ_high != 0 || z->succ_low != 0) return stbi__err("bad SOS","Corrupt JPEG");
         z->spec_end = 63;
      }
   }

   return 1;
}

static int stbi__free_jpeg_components(stbi__jpeg *z, int ncomp, int why)
{
   int i;
   for (i=0; i < ncomp; ++i) {
      if (z->img_comp[i].raw_data) {
         STBI_FREE(z->img_comp[i].raw_data);
         z->img_comp[i].raw_data = NULL;
         z->img_comp[i].data = NULL;
      }
      if (z->img_comp[i].raw_coeff) {
         STBI_FREE(z->img_comp[i].raw_coeff);
         z->img_comp[i].raw_coeff = 0;
         z->img_comp[i].coeff = 0;
      }
      if (z->img_comp[i].linebuf) {
         STBI_FREE(z->img_comp[i].linebuf);
         z->img_comp[i].linebuf = NULL;
      }
   }
   return why;
}

static int stbi__process_frame_header(stbi__jpeg *z, int scan)
{
   stbi__context *s = z->s;
   int Lf,p,i,q, h_max=1,v_max=1,c;
   Lf = stbi__get16be(s);         if (Lf < 11) return stbi__err("bad SOF len","Corrupt JPEG"); // JPEG
   p  = stbi__get8(s);            if (p != 8) return stbi__err("only 8-bit","JPEG format not supported: 8-bit only"); // JPEG baseline
   s->img_y = stbi__get16be(s);   if (s->img_y == 0) return stbi__err("no header height", "JPEG format not supported: delayed height"); // Legal, but we don’t handle it--but neither does IJG
   s->img_x = stbi__get16be(s);   if (s->img_x == 0) return stbi__err("0 width","Corrupt JPEG"); // JPEG requires
   if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
   if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
   c = stbi__get8(s);
   if (c != 3 && c != 1 && c != 4) return stbi__err("bad component count","Corrupt JPEG");
   s->img_n = c;
   for (i=0; i < c; ++i) {
      z->img_comp[i].data = NULL;
      z->img_comp[i].linebuf = NULL;
   }

   if (Lf != 8+3*s->img_n) return stbi__err("bad SOF len","Corrupt JPEG");

   z->rgb = 0;
   for (i=0; i < s->img_n; ++i) {
      static const unsigned char rgb[3] = { 'R', 'G', 'B' };
      z->img_comp[i].id = stbi__get8(s);
      if (s->img_n == 3 && z->img_comp[i].id == rgb[i])
         ++z->rgb;
      q = stbi__get8(s);
      z->img_comp[i].h = (q » 4);  if (!z->img_comp[i].h || z->img_comp[i].h > 4) return stbi__err("bad H","Corrupt JPEG");
      z->img_comp[i].v = q & 15;    if (!z->img_comp[i].v || z->img_comp[i].v > 4) return stbi__err("bad V","Corrupt JPEG");
      z->img_comp[i].tq = stbi__get8(s);  if (z->img_comp[i].tq > 3) return stbi__err("bad TQ","Corrupt JPEG");
   }

   if (scan != STBI__SCAN_load) return 1;

   if (!stbi__mad3sizes_valid(s->img_x, s->img_y, s->img_n, 0)) return stbi__err("too large", "Image too large to decode");

   for (i=0; i < s->img_n; ++i) {
      if (z->img_comp[i].h > h_max) h_max = z->img_comp[i].h;
      if (z->img_comp[i].v > v_max) v_max = z->img_comp[i].v;
   }

   // check that plane subsampling factors are integer ratios; our resamplers can't deal with fractional ratios
   // and I've never seen a non-corrupted JPEG file actually use them
   for (i=0; i < s->img_n; ++i) {
      if (h_max % z->img_comp[i].h != 0) return stbi__err("bad H","Corrupt JPEG");
      if (v_max % z->img_comp[i].v != 0) return stbi__err("bad V","Corrupt JPEG");
   }

   // compute interleaved mcu info
   z->img_h_max = h_max;
   z->img_v_max = v_max;
   z->img_mcu_w = h_max * 8;
   z->img_mcu_h = v_max * 8;
   // these sizes can't be more than 17 bits
   z->img_mcu_x = (s->img_x + z->img_mcu_w-1) / z->img_mcu_w;
   z->img_mcu_y = (s->img_y + z->img_mcu_h-1) / z->img_mcu_h;

   for (i=0; i < s->img_n; ++i) {
      // number of effective pixels (e.g. for non-interleaved MCU)
      z->img_comp[i].x = (s->img_x * z->img_comp[i].h + h_max-1) / h_max;
      z->img_comp[i].y = (s->img_y * z->img_comp[i].v + v_max-1) / v_max;
      // to simplify generation, we'll allocate enough memory to decode
      // the bogus oversized data from using interleaved MCUs and their
      // big blocks (e.g. a 16x16 iMCU on an image of width 33); we won't
      // discard the extra data until colorspace conversion
      //
      // img_mcu_x, img_mcu_y: <=17 bits; comp[i].h and .v are <=4 (checked earlier)
      // so these muls can't overflow with 32-bit ints (which we require)
      z->img_comp[i].w2 = z->img_mcu_x * z->img_comp[i].h * 8;
      z->img_comp[i].h2 = z->img_mcu_y * z->img_comp[i].v * 8;
      z->img_comp[i].coeff = 0;
      z->img_comp[i].raw_coeff = 0;
      z->img_comp[i].linebuf = NULL;
      z->img_comp[i].raw_data = stbi__malloc_mad2(z->img_comp[i].w2, z->img_comp[i].h2, 15);
      if (z->img_comp[i].raw_data == NULL)
         return stbi__free_jpeg_components(z, i+1, stbi__err("outofmem", "Out of memory"));
      // align blocks for idct using mmx/sse
      z->img_comp[i].data = (stbi_uc*) (((size_t) z->img_comp[i].raw_data + 15) & ~15);
      if (z->progressive) {
         // w2, h2 are multiples of 8 (see above)
         z->img_comp[i].coeff_w = z->img_comp[i].w2 / 8;
         z->img_comp[i].coeff_h = z->img_comp[i].h2 / 8;
         z->img_comp[i].raw_coeff = stbi__malloc_mad3(z->img_comp[i].w2, z->img_comp[i].h2, sizeof(short), 15);
         if (z->img_comp[i].raw_coeff == NULL)
            return stbi__free_jpeg_components(z, i+1, stbi__err("outofmem", "Out of memory"));
         z->img_comp[i].coeff = (short*) (((size_t) z->img_comp[i].raw_coeff + 15) & ~15);
      }
   }

   return 1;
}

// use comparisons since in some cases we handle more than one case (e.g. SOF)
#define stbi__DNL(x)         ((x) == 0xdc)
#define stbi__SOI(x)         ((x) == 0xd8)
#define stbi__EOI(x)         ((x) == 0xd9)
#define stbi__SOF(x)         ((x) == 0xc0 || (x) == 0xc1 || (x) == 0xc2)
#define stbi__SOS(x)         ((x) == 0xda)

#define stbi__SOF_progressive(x)   ((x) == 0xc2)

```

```
static int stbi__decode_jpeg_header(stbi__jpeg *z, int scan)
{
   int m;
   z->jfif = 0;
   z->app14_color_transform = -1; // valid values are 0,1,2
   z->marker = STBI__MARKER_none; // initialize cached marker to empty
   m = stbi__get_marker(z);
   if (!stbi__SOI(m)) return stbi__err("no SOI","Corrupt JPEG");
   if (scan == STBI__SCAN_type) return 1;
   m = stbi__get_marker(z);
   while (!stbi__SOF(m)) {
      if (!stbi__process_marker(z,m)) return 0;
      m = stbi__get_marker(z);
      while (m == STBI__MARKER_none) {
         // some files have extra padding after their blocks, so ok, we'll scan
         if (stbi__at_eof(z->s)) return stbi__err("no SOF", "Corrupt JPEG");
         m = stbi__get_marker(z);
      }
   }
   z->progressive = stbi__SOF_progressive(m);
   if (!stbi__process_frame_header(z, scan)) return 0;
   return 1;
}

static int stbi__skip_jpeg_junk_at_end(stbi__jpeg *j)
{
   // some JPEGs have junk at end, skip over it but if we find what looks
   // like a valid marker, resume there
   while (!stbi__at_eof(j->s)) {
      int x = stbi__get8(j->s);
      while (x == 255) { // might be a marker
         if (stbi__at_eof(j->s)) return STBI__MARKER_none;
         x = stbi__get8(j->s);
         if (x != 0x00 && x != 0xff) {
            // not a stuffed zero or lead-in to another marker, looks
            // like an actual marker, return it
            return x;
         }
         // stuffed zero has x=0 now which ends the loop, meaning we go
         // back to regular scan loop.
         // repeated 0xff keeps trying to read the next byte of the marker.
      }
   }
   return STBI__MARKER_none;
}

// decode image to YCbCr format
static int stbi__decode_jpeg_image(stbi__jpeg *j)
{
   int m;
   for (m = 0; m < 4; m++) {
      j->img_comp[m].raw_data = NULL;
      j->img_comp[m].raw_coeff = NULL;
   }
   j->restart_interval = 0;
   if (!stbi__decode_jpeg_header(j, STBI__SCAN_load)) return 0;
   m = stbi__get_marker(j);
   while (!stbi__EOI(m)) {
      if (stbi__SOS(m)) {
         if (!stbi__process_scan_header(j)) return 0;
         if (!stbi__parse_entropy_coded_data(j)) return 0;
         if (j->marker == STBI__MARKER_none ) {
         j->marker = stbi__skip_jpeg_junk_at_end(j);
            // if we reach eof without hitting a marker, stbi__get_marker() below will fail and we'll eventually return 0
         }
         m = stbi__get_marker(j);
         if (STBI__RESTART(m))
            m = stbi__get_marker(j);
      } else if (stbi__DNL(m)) {
         int Ld = stbi__get16be(j->s);
         stbi__uint32 NL = stbi__get16be(j->s);
         if (Ld != 4) return stbi__err("bad DNL len", "Corrupt JPEG");
         if (NL != j->s->img_y) return stbi__err("bad DNL height", "Corrupt JPEG");
         m = stbi__get_marker(j);
      } else {
         if (!stbi__process_marker(j, m)) return 1;
         m = stbi__get_marker(j);
      }
   }
   if (j->progressive)
      stbi__jpeg_finish(j);
   return 1;
}

// static jfif-centered resampling (across block boundaries)

```

```
typedef stbi_uc *(*resample_row_func)(stbi_uc *out, stbi_uc *in0, stbi_uc *in1,
                                    int w, int hs);

#define stbi__div4(x) ((stbi_uc) ((x) » 2))

static stbi_uc *resample_row_1(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   STBI_NOTUSED(out);
   STBI_NOTUSED(in_far);
   STBI_NOTUSED(w);
   STBI_NOTUSED(hs);
   return in_near;
}

static stbi_uc* stbi__resample_row_v_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   // need to generate two samples vertically for every one in input
   int i;
   STBI_NOTUSED(hs);
   for (i=0; i < w; ++i)
      out[i] = stbi__div4(3*in_near[i] + in_far[i] + 2);
   return out;
}

static stbi_uc*  stbi__resample_row_h_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   // need to generate two samples horizontally for every one in input
   int i;
   stbi_uc *input = in_near;

   if (w == 1) {
      // if only one sample, can’t do any interpolation
      out[0] = out[1] = input[0];
      return out;
   }

   out[0] = input[0];
   out[1] = stbi__div4(input[0]*3 + input[1] + 2);
   for (i=1; i < w-1; ++i) {
      int n = 3*input[i]+2;
      out[i*2+0] = stbi__div4(n+input[i-1]);
      out[i*2+1] = stbi__div4(n+input[i+1]);
   }
   out[i*2+0] = stbi__div4(input[w-2]*3 + input[w-1] + 2);
   out[i*2+1] = input[w-1];

   STBI_NOTUSED(in_far);
   STBI_NOTUSED(hs);

   return out;
}

#define stbi__div16(x) ((stbi_uc) ((x) » 4))

static stbi_uc *stbi__resample_row_hv_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   // need to generate 2x2 samples for every one in input
   int i,t0,t1;
   if (w == 1) {
      out[0] = out[1] = stbi__div4(3*in_near[0] + in_far[0] + 2);
      return out;
   }

   t1 = 3*in_near[0] + in_far[0];
   out[0] = stbi__div4(t1+2);
   for (i=1; i < w; ++i) {
      t0 = t1;
      t1 = 3*in_near[i]+in_far[i];
      out[i*2-1] = stbi__div16(3*t0 + t1 + 8);
      out[i*2  ] = stbi__div16(3*t1 + t0 + 8);
   }
   out[w*2-1] = stbi__div4(t1+2);

   STBI_NOTUSED(hs);

   return out;
}

#if defined(STBI_SSE2) || defined(STBI_NEON)
static stbi_uc *stbi__resample_row_hv_2_simd(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   // need to generate 2x2 samples for every one in input
   int i=0,t0,t1;

   if (w == 1) {
      out[0] = out[1] = stbi__div4(3*in_near[0] + in_far[0] + 2);
      return out;
   }

   t1 = 3*in_near[0] + in_far[0];
   // process groups of 8 pixels for as long as we can.
   // note we can’t handle the last pixel in a row in this loop
   // because we need to handle the filter boundary conditions.
   for (; i < ((w-1) & ~7); i += 8) {
#if defined(STBI_SSE2)
      // load and perform the vertical filtering pass
      // this uses 3*x + y = 4*x + (y - x)
      __m128i zero  = _mm_setzero_si128();
      __m128i farb  = _mm_loadl_epi64((__m128i *) (in_far + i));
      __m128i nearb = _mm_loadl_epi64((__m128i *) (in_near + i));
      __m128i farw  = _mm_unpacklo_epi8(farb, zero);
      __m128i nearw = _mm_unpacklo_epi8(nearb, zero);
      __m128i diff  = _mm_sub_epi16(farw, nearw);
      __m128i nears = _mm_slli_epi16(nearw, 2);
      __m128i curr  = _mm_add_epi16(nears, diff); // current row

      // horizontal filter works the same based on shifted vers of current
      // row. "prev" is current row shifted right by 1 pixel; we need to
      // insert the previous pixel value (from t1).
      // "next" is current row shifted left by 1 pixel, with first pixel
      // of next block of 8 pixels added in.
      __m128i prv0 = _mm_slli_si128(curr, 2);
      __m128i nxt0 = _mm_srli_si128(curr, 2);
      __m128i prev = _mm_insert_epi16(prv0, t1, 0);
      __m128i next = _mm_insert_epi16(nxt0, 3*in_near[i+8] + in_far[i+8], 7);

      // horizontal filter, polyphase implementation since it’s convenient:
      // even pixels = 3*cur + prev = cur*4 + (prev - cur)
      // odd  pixels = 3*cur + next = cur*4 + (next - cur)
      // note the shared term.
      __m128i bias  = _mm_set1_epi16(8);
      __m128i curs = _mm_slli_epi16(curr, 2);
      __m128i prvd = _mm_sub_epi16(prev, curr);
      __m128i nxtd = _mm_sub_epi16(next, curr);
      __m128i curb = _mm_add_epi16(curs, bias);
      __m128i even = _mm_add_epi16(prvd, curb);
      __m128i odd  = _mm_add_epi16(nxtd, curb);

      // interleave even and odd pixels, then undo scaling.
      __m128i int0 = _mm_unpacklo_epi16(even, odd);
      __m128i int1 = _mm_unpackhi_epi16(even, odd);
      __m128i de0  = _mm_srli_epi16(int0, 4);
      __m128i de1  = _mm_srli_epi16(int1, 4);

      // pack and write output
      __m128i outv = _mm_packus_epi16(de0, de1);
      _mm_storeu_si128((__m128i *) (out + i*2), outv);
#elif defined(STBI_NEON)
      // load and perform the vertical filtering pass
      // this uses 3*x + y = 4*x + (y - x)
      uint8x8_t farb  = vld1_u8(in_far + i);
      uint8x8_t nearb = vld1_u8(in_near + i);
      int16x8_t diff  = vreinterpretq_s16_u16(vsubl_u8(farb, nearb));
      int16x8_t nears = vreinterpretq_s16_u16(vshll_n_u8(nearb, 2));
      int16x8_t curr  = vaddq_s16(nears, diff); // current row

      // horizontal filter works the same based on shifted vers of current
      // row. "prev" is current row shifted right by 1 pixel; we need to
      // insert the previous pixel value (from t1).
      // "next" is current row shifted left by 1 pixel, with first pixel
      // of next block of 8 pixels added in.
      int16x8_t prv0 = vextq_s16(curr, curr, 7);
      int16x8_t nxt0 = vextq_s16(curr, curr, 1);
      int16x8_t prev = vsetq_lane_s16(t1, prv0, 0);
      int16x8_t next = vsetq_lane_s16(3*in_near[i+8] + in_far[i+8], nxt0, 7);

      // horizontal filter, polyphase implementation since it’s convenient:
      // even pixels = 3*cur + prev = cur*4 + (prev - cur)
      // odd  pixels = 3*cur + next = cur*4 + (next - cur)
      // note the shared term.
      int16x8_t curs = vshlq_n_s16(curr, 2);
      int16x8_t prvd = vsubq_s16(prev, curr);
      int16x8_t nxtd = vsubq_s16(next, curr);
      int16x8_t even = vaddq_s16(curs, prvd);
      int16x8_t odd  = vaddq_s16(curs, nxtd);

      // undo scaling and round, then store with even/odd phases interleaved
      uint8x8x2_t o;
      o.val[0] = vqrshrun_n_s16(even, 4);
      o.val[1] = vqrshrun_n_s16(odd,  4);
      vst2_u8(out + i*2, o);
```

```
#endif

      // "previous" value for next iter
      t1 = 3*in_near[i+7] + in_far[i+7];
   }

   t0 = t1;
   t1 = 3*in_near[i] + in_far[i];
   out[i*2] = stbi__div16(3*t1 + t0 + 8);

   for (++i; i < w; ++i) {
      t0 = t1;
      t1 = 3*in_near[i]+in_far[i];
      out[i*2-1] = stbi__div16(3*t0 + t1 + 8);
      out[i*2  ] = stbi__div16(3*t1 + t0 + 8);
   }
   out[w*2-1] = stbi__div4(t1+2);

   STBI_NOTUSED(hs);

   return out;
}
#endif

static stbi_uc *stbi__resample_row_generic(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
   // resample with nearest-neighbor
   int i,j;
   STBI_NOTUSED(in_far);
   for (i=0; i < w; ++i)
      for (j=0; j < hs; ++j)
         out[i*hs+j] = in_near[i];
   return out;
}

// this is a reduced-precision calculation of YCbCr-to-RGB introduced
// to make sure the code produces the same results in both SIMD and scalar
#define stbi__float2fixed(x)  (((int) ((x) * 4096.0f + 0.5f)) « 8)
static void stbi__YCbCr_to_RGB_row(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc *pcr, int count, int step)
{
   int i;
   for (i=0; i < count; ++i) {
      int y_fixed = (y[i] « 20) + (1«19); // rounding
      int r,g,b;
      int cr = pcr[i] - 128;
      int cb = pcb[i] - 128;
      r = y_fixed +  cr* stbi__float2fixed(1.40200f);
      g = y_fixed + (cr*-stbi__float2fixed(0.71414f)) + ((cb*-stbi__float2fixed(0.34414f)) & 0xffff0000);
      b = y_fixed                                     +   cb* stbi__float2fixed(1.77200f);
      r »= 20;
      g »= 20;
      b »= 20;
      if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }
      if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }
      if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }
      out[0] = (stbi_uc)r;
      out[1] = (stbi_uc)g;
      out[2] = (stbi_uc)b;
      out[3] = 255;
      out += step;
   }
}

#if defined(STBI_SSE2) || defined(STBI_NEON)
static void stbi__YCbCr_to_RGB_simd(stbi_uc *out, stbi_uc const *y, stbi_uc const *pcb, stbi_uc const *pcr, int count, int step)
{
   int i = 0;

#ifdef STBI_SSE2
   // step == 3 is pretty ugly on the final interleave, and i’m not convinced
   // it’s useful in practice (you wouldn’t use it for textures, for example).
   // so just accelerate step == 4 case.
   if (step == 4) {
      // this is a fairly straightforward implementation and not super-optimized.
      __m128i signflip  = _mm_set1_epi8(-0x80);
      __m128i cr_const0 = _mm_set1_epi16(   (short) ( 1.40200f*4096.0f+0.5f));
      __m128i cr_const1 = _mm_set1_epi16( - (short) ( 0.71414f*4096.0f+0.5f));
      __m128i cb_const0 = _mm_set1_epi16( - (short) ( 0.34414f*4096.0f+0.5f));
      __m128i cb_const1 = _mm_set1_epi16(   (short) ( 1.77200f*4096.0f+0.5f));
      __m128i y_bias = _mm_set1_epi8((char) (unsigned char) 128);
      __m128i xw = _mm_set1_epi16(255); // alpha channel

      for (; i+7 < count; i += 8) {
         // load
         __m128i y_bytes = _mm_loadl_epi64((__m128i *) (y+i));
         __m128i cr_bytes = _mm_loadl_epi64((__m128i *) (pcr+i));
         __m128i cb_bytes = _mm_loadl_epi64((__m128i *) (pcb+i));
         __m128i cr_biased = _mm_xor_si128(cr_bytes, signflip); // -128
         __m128i cb_biased = _mm_xor_si128(cb_bytes, signflip); // -128

         // unpack to short (and left-shift cr, cb by 8)
         __m128i yw  = _mm_unpacklo_epi8(y_bias, y_bytes);
         __m128i crw = _mm_unpacklo_epi8(_mm_setzero_si128(), cr_biased);
         __m128i cbw = _mm_unpacklo_epi8(_mm_setzero_si128(), cb_biased);

         // color transform
         __m128i yws = _mm_srli_epi16(yw, 4);
         __m128i cr0 = _mm_mulhi_epi16(cr_const0, crw);
         __m128i cb0 = _mm_mulhi_epi16(cb_const0, cbw);
         __m128i cb1 = _mm_mulhi_epi16(cbw, cb_const1);
         __m128i cr1 = _mm_mulhi_epi16(crw, cr_const1);
         __m128i rws = _mm_add_epi16(cr0, yws);
         __m128i gwt = _mm_add_epi16(cb0, yws);
         __m128i bws = _mm_add_epi16(yws, cb1);
         __m128i gws = _mm_add_epi16(gwt, cr1);

         // descale
         __m128i rw = _mm_srai_epi16(rws, 4);
         __m128i bw = _mm_srai_epi16(bws, 4);
         __m128i gw = _mm_srai_epi16(gws, 4);

         // back to byte, set up for transpose
         __m128i brb = _mm_packus_epi16(rw, bw);
         __m128i gxb = _mm_packus_epi16(gw, xw);

         // transpose to interleave channels
         __m128i t0 = _mm_unpacklo_epi8(brb, gxb);
         __m128i t1 = _mm_unpackhi_epi8(brb, gxb);
         __m128i o0 = _mm_unpacklo_epi16(t0, t1);
         __m128i o1 = _mm_unpackhi_epi16(t0, t1);

         // store
         _mm_storeu_si128((__m128i *) (out + 0), o0);
         _mm_storeu_si128((__m128i *) (out + 16), o1);
         out += 32;
      }
   }
#endif

#ifdef STBI_NEON
   // in this version, step=3 support would be easy to add. but is there demand?
   if (step == 4) {
      // this is a fairly straightforward implementation and not super-optimized.
      uint8x8_t signflip = vdup_n_u8(0x80);
      int16x8_t cr_const0 = vdupq_n_s16(   (short) ( 1.40200f*4096.0f+0.5f));
      int16x8_t cr_const1 = vdupq_n_s16( - (short) ( 0.71414f*4096.0f+0.5f));
      int16x8_t cb_const0 = vdupq_n_s16( - (short) ( 0.34414f*4096.0f+0.5f));
      int16x8_t cb_const1 = vdupq_n_s16(   (short) ( 1.77200f*4096.0f+0.5f));

      for (; i+7 < count; i += 8) {
         // load
         uint8x8_t y_bytes  = vld1_u8(y + i);
         uint8x8_t cr_bytes = vld1_u8(pcr + i);
         uint8x8_t cb_bytes = vld1_u8(pcb + i);
         int8x8_t cr_biased = vreinterpret_s8_u8(vsub_u8(cr_bytes, signflip));
         int8x8_t cb_biased = vreinterpret_s8_u8(vsub_u8(cb_bytes, signflip));

         // expand to s16
         int16x8_t yws = vreinterpretq_s16_u16(vshll_n_u8(y_bytes, 4));
         int16x8_t crw = vshll_n_s8(cr_biased, 7);
         int16x8_t cbw = vshll_n_s8(cb_biased, 7);

         // color transform
         int16x8_t cr0 = vqdmulhq_s16(crw, cr_const0);
         int16x8_t cb0 = vqdmulhq_s16(cbw, cb_const0);
         int16x8_t cr1 = vqdmulhq_s16(crw, cr_const1);
         int16x8_t cb1 = vqdmulhq_s16(cbw, cb_const1);
         int16x8_t rws = vaddq_s16(yws, cr0);
         int16x8_t gws = vaddq_s16(vaddq_s16(yws, cb0), cr1);
         int16x8_t bws = vaddq_s16(yws, cb1);

         // undo scaling, round, convert to byte
         uint8x8x4_t o;
         o.val[0] = vqrshrun_n_s16(rws, 4);
         o.val[1] = vqrshrun_n_s16(gws, 4);
         o.val[2] = vqrshrun_n_s16(bws, 4);
         o.val[3] = vdup_n_u8(255);

         // store, interleaving r/g/b/a
         vst4_u8(out, o);
         out += 8*4;
      }
   }
#endif

   for (; i < count; ++i) {
      int y_fixed = (y[i] « 20) + (1«19); // rounding
      int r,g,b;
      int cr = pcr[i] - 128;
      int cb = pcb[i] - 128;
      r = y_fixed + cr* stbi__float2fixed(1.40200f);
      g = y_fixed + cr*-stbi__float2fixed(0.71414f) + ((cb*-stbi__float2fixed(0.34414f)) &
0xffff0000);
      b = y_fixed                                   +   cb* stbi__float2fixed(1.77200f);
      r »= 20;
      g »= 20;
      b »= 20;
      if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }
      if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }
      if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }
      out[0] = (stbi_uc)r;
      out[1] = (stbi_uc)g;
      out[2] = (stbi_uc)b;
      out[3] = 255;
      out += step;
   }
}
#endif

// set up the kernels
static void stbi__setup_jpeg(stbi__jpeg *j)
{
   j->idct_block_kernel = stbi__idct_block;
   j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_row;
   j->resample_row_hv_2_kernel = stbi__resample_row_hv_2;

#ifdef STBI_SSE2
   if (stbi__sse2_available()) {
      j->idct_block_kernel = stbi__idct_simd;
      j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
      j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
   }
#endif

#ifdef STBI_NEON
   j->idct_block_kernel = stbi__idct_simd;
   j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
   j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
#endif
}

// clean up the temporary component buffers
static void stbi__cleanup_jpeg(stbi__jpeg *j)
{
   stbi__free_jpeg_components(j, j->s->img_n, 0);
}

typedef struct
{
   resample_row_func resample;
   stbi_uc *line0,*line1;
   int hs,vs;   // expansion factor in each axis
   int w_lores; // horizontal pixels pre-expansion
   int ystep;   // how far through vertical expansion we are
   int ypos;    // which pre-expansion row we're on
} stbi__resample;

// fast 0..255 * 0..255 => 0..255 rounded multiplication
static stbi_uc stbi__blinn_8x8(stbi_uc x, stbi_uc y)
{
   unsigned int t = x*y + 128;
   return (stbi_uc) ((t + (t »8)) » 8);
}

static stbi_uc *load_jpeg_image(stbi__jpeg *z, int *out_x, int *out_y, int *comp, int req_comp)
{
   int n, decode_n, is_rgb;
   z->s->img_n = 0; // make stbi__cleanup_jpeg safe

   // validate req_comp
   if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");

   // load a jpeg image from whichever source, but leave in YCbCr format
   if (!stbi__decode_jpeg_image(z)) { stbi__cleanup_jpeg(z); return NULL; }

   // determine actual number of components to generate
   n = req_comp ? req_comp : z->s->img_n >= 3 ? 3 : 1;

   is_rgb = z->s->img_n == 3 && (z->rgb == 3 || (z->app14_color_transform == 0 && !z->jfif));

   if (z->s->img_n == 3 && n < 3 && !is_rgb)
      decode_n = 1;
   else
      decode_n = z->s->img_n;

   // nothing to do if no components requested; check this now to avoid
   // accessing uninitialized coutput[0] later
   if (decode_n <= 0) { stbi__cleanup_jpeg(z); return NULL; }

   // resample and color-convert
   {
      int k;
      unsigned int i,j;
      stbi_uc *output;
      stbi_uc *coutput[4] = { NULL, NULL, NULL, NULL };

      stbi__resample res_comp[4];

      for (k=0; k < decode_n; ++k) {
         stbi__resample *r = &res_comp[k];

         // allocate line buffer big enough for upsampling off the edges
         // with upsample factor of 4
         z->img_comp[k].linebuf = (stbi_uc *) stbi__malloc(z->s->img_x + 3);
         if (!z->img_comp[k].linebuf) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of memory"); }

         r->hs      = z->img_h_max / z->img_comp[k].h;
         r->vs      = z->img_v_max / z->img_comp[k].v;
         r->ystep   = r->vs » 1;
         r->w_lores = (z->s->img_x + r->hs-1) / r->hs;
         r->ypos    = 0;
         r->line0   = r->line1 = z->img_comp[k].data;

         if      (r->hs == 1 && r->vs == 1) r->resample = resample_row_1;
         else if (r->hs == 1 && r->vs == 2) r->resample = stbi__resample_row_v_2;
         else if (r->hs == 2 && r->vs == 1) r->resample = stbi__resample_row_h_2;
         else if (r->hs == 2 && r->vs == 2) r->resample = z->resample_row_hv_2_kernel;
         else                               r->resample = stbi__resample_row_generic;
      }

      // can't error after this so, this is safe
      output = (stbi_uc *) stbi__malloc_mad3(n, z->s->img_x, z->s->img_y, 1);
      if (!output) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of memory"); }

      // now go ahead and resample
      for (j=0; j < z->s->img_y; ++j) {
         stbi_uc *out = output + n * z->s->img_x * j;
         for (k=0; k < decode_n; ++k) {
            stbi__resample *r = &res_comp[k];
            int y_bot = r->ystep >= (r->vs » 1);
            coutput[k] = r->resample(z->img_comp[k].linebuf,
                                     y_bot ? r->line1 : r->line0,
                                     y_bot ? r->line0 : r->line1,
                                     r->w_lores, r->hs);
            if (++r->ystep >= r->vs) {
               r->ystep = 0;
               r->line0 = r->line1;
               if (++r->ypos < z->img_comp[k].y)
                  r->line1 += z->img_comp[k].w2;
            }
         }
         if (n >= 3) {
            stbi_uc *y = coutput[0];
            if (z->s->img_n == 3) {
               if (is_rgb) {
                  for (i=0; i < z->s->img_x; ++i) {
                     out[0] = y[i];
                     out[1] = coutput[1][i];
                     out[2] = coutput[2][i];
                     out[3] = 255;
                     out += n;
                  }
               } else {
                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
               }
            } else if (z->s->img_n == 4) {
               if (z->app14_color_transform == 0) { // CMYK
                  for (i=0; i < z->s->img_x; ++i) {
                     stbi_uc m = coutput[3][i];
                     out[0] = stbi__blinn_8x8(coutput[0][i], m);
                     out[1] = stbi__blinn_8x8(coutput[1][i], m);
                     out[2] = stbi__blinn_8x8(coutput[2][i], m);
                     out[3] = 255;
                     out += n;
                  }
               } else if (z->app14_color_transform == 2) { // YCCK
                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
                  for (i=0; i < z->s->img_x; ++i) {
                     stbi_uc m = coutput[3][i];
                     out[0] = stbi__blinn_8x8(255 - out[0], m);
                     out[1] = stbi__blinn_8x8(255 - out[1], m);
                     out[2] = stbi__blinn_8x8(255 - out[2], m);
                     out += n;
                  }
               } else { // YCbCr + alpha?  Ignore the fourth channel for now
                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
               }
            } else
               for (i=0; i < z->s->img_x; ++i) {
                  out[0] = out[1] = out[2] = y[i];
                  out[3] = 255; // not used if n==3
                  out += n;
               }
         } else {
            if (is_rgb) {
               if (n == 1)
                  for (i=0; i < z->s->img_x; ++i)
                     *out++ = stbi__compute_y(coutput[0][i], coutput[1][i], coutput[2][i]);
               else {
                  for (i=0; i < z->s->img_x; ++i, out += 2) {
                     out[0] = stbi__compute_y(coutput[0][i], coutput[1][i], coutput[2][i]);
                     out[1] = 255;
                  }
               }
            } else if (z->s->img_n == 4 && z->app14_color_transform == 0) {
               for (i=0; i < z->s->img_x; ++i) {
                  stbi_uc m = coutput[3][i];
                  stbi_uc r = stbi__blinn_8x8(coutput[0][i], m);
                  stbi_uc g = stbi__blinn_8x8(coutput[1][i], m);
                  stbi_uc b = stbi__blinn_8x8(coutput[2][i], m);
                  out[0] = stbi__compute_y(r, g, b);
                  out[1] = 255;
                  out += n;
               }
            } else if (z->s->img_n == 4 && z->app14_color_transform == 2) {
               for (i=0; i < z->s->img_x; ++i) {
                  out[0] = stbi__blinn_8x8(255 - coutput[0][i], coutput[3][i]);
                  out[1] = 255;
                  out += n;
               }
            } else {
               stbi_uc *y = coutput[0];
               if (n == 1)
                  for (i=0; i < z->s->img_x; ++i) out[i] = y[i];
               else
                  for (i=0; i < z->s->img_x; ++i) { *out++ = y[i]; *out++ = 255; }
            }
         }
      }
      stbi__cleanup_jpeg(z);
      *out_x = z->s->img_x;
      *out_y = z->s->img_y;
      if (comp) *comp = z->s->img_n >= 3 ? 3 : 1; // report original components, not output
      return output;
   }
}

static void *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   unsigned char* result;
   stbi__jpeg* j = (stbi__jpeg*) stbi__malloc(sizeof(stbi__jpeg));
   if (!j) return stbi__errpuc("outofmem", "Out of memory");
   memset(j, 0, sizeof(stbi__jpeg));
   STBI_NOTUSED(ri);
   j->s = s;
   stbi__setup_jpeg(j);
   result = load_jpeg_image(j, x,y,comp,req_comp);
   STBI_FREE(j);
   return result;
}

static int stbi__jpeg_test(stbi__context *s)
{
   int r;
   stbi__jpeg* j = (stbi__jpeg*)stbi__malloc(sizeof(stbi__jpeg));
   if (!j) return stbi__err("outofmem", "Out of memory");
   memset(j, 0, sizeof(stbi__jpeg));
   j->s = s;
   stbi__setup_jpeg(j);
   r = stbi__decode_jpeg_header(j, STBI__SCAN_type);
   stbi__rewind(s);
   STBI_FREE(j);
   return r;
}

static int stbi__jpeg_info_raw(stbi__jpeg *j, int *x, int *y, int *comp)
{
   if (!stbi__decode_jpeg_header(j, STBI__SCAN_header)) {
      stbi__rewind( j->s );
      return 0;
   }
   if (x) *x = j->s->img_x;
   if (y) *y = j->s->img_y;
   if (comp) *comp = j->s->img_n >= 3 ? 3 : 1;
   return 1;
}

static int stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp)
{
   int result;
   stbi__jpeg* j = (stbi__jpeg*) (stbi__malloc(sizeof(stbi__jpeg)));
   if (!j) return stbi__err("outofmem", "Out of memory");
   memset(j, 0, sizeof(stbi__jpeg));
   j->s = s;
   result = stbi__jpeg_info_raw(j, x, y, comp);
   STBI_FREE(j);
   return result;
}
#endif

// public domain zlib decode    v0.2  Sean Barrett 2006-11-18
//    simple implementation
//      - all input must be provided in an upfront buffer
//      - all output is written to a single output buffer (can malloc/realloc)
//    performance
//      - fast huffman

#ifndef STBI_NO_ZLIB

// fast-way is faster to check than jpeg huffman, but slow way is slower
#define STBI__ZFAST_BITS  9 // accelerate all cases in default tables
#define STBI__ZFAST_MASK  ((1 « STBI__ZFAST_BITS) - 1)
#define STBI__ZNSYMS 288 // number of symbols in literal/length alphabet

// zlib-style huffman encoding
// (jpegs packs from left, zlib from right, so can't share code)
typedef struct
{
   stbi__uint16 fast[1 « STBI__ZFAST_BITS];
   stbi__uint16 firstcode[16];
   int maxcode[17];
   stbi__uint16 firstsymbol[16];
   stbi_uc  size[STBI__ZNSYMS];
   stbi__uint16 value[STBI__ZNSYMS];
} stbi__zhuffman;

stbi_inline static int stbi__bitreverse16(int n)
{
  n = ((n & 0xAAAA) »  1) | ((n & 0x5555) « 1);
  n = ((n & 0xCCCC) »  2) | ((n & 0x3333) « 2);
  n = ((n & 0xF0F0) »  4) | ((n & 0x0F0F) « 4);
  n = ((n & 0xFF00) »  8) | ((n & 0x00FF) « 8);
  return n;
}

stbi_inline static int stbi__bit_reverse(int v, int bits)
{
   STBI_ASSERT(bits <= 16);
   // to bit reverse n bits, reverse 16 and shift
   // e.g. 11 bits, bit reverse and shift away 5
   return stbi__bitreverse16(v) » (16-bits);
}

static int stbi__zbuild_huffman(stbi__zhuffman *z, const stbi_uc *sizelist, int num)
{
   int i,k=0;
   int code, next_code[16], sizes[17];

   // DEFLATE spec for generating codes
   memset(sizes, 0, sizeof(sizes));
   memset(z->fast, 0, sizeof(z->fast));
   for (i=0; i < num; ++i)
      ++sizes[sizelist[i]];
   sizes[0] = 0;
   for (i=1; i < 16; ++i)
      if (sizes[i] > (1 « i))
```

```
         return stbi__err("bad sizes", "Corrupt PNG");
   code = 0;
   for (i=1; i < 16; ++i) {
      next_code[i] = code;
      z->firstcode[i] = (stbi__uint16) code;
      z->firstsymbol[i] = (stbi__uint16) k;
      code = (code + sizes[i]);
      if (sizes[i])
         if (code-1 >= (1 « i)) return stbi__err("bad codelengths","Corrupt PNG");
      z->maxcode[i] = code « (16-i); // preshift for inner loop
      code «= 1;
      k += sizes[i];
   }
   z->maxcode[16] = 0x10000; // sentinel
   for (i=0; i < num; ++i) {
      int s = sizelist[i];
      if (s) {
         int c = next_code[s] - z->firstcode[s] + z->firstsymbol[s];
         stbi__uint16 fastv = (stbi__uint16) ((s « 9) | i);
         z->size [c] = (stbi_uc     ) s;
         z->value[c] = (stbi__uint16) i;
         if (s <= STBI__ZFAST_BITS) {
            int j = stbi__bit_reverse(next_code[s],s);
            while (j < (1 « STBI__ZFAST_BITS)) {
               z->fast[j] = fastv;
               j += (1 « s);
            }
         }
         ++next_code[s];
      }
   }
   return 1;
}

// zlib-from-memory implementation for PNG reading
//    because PNG allows splitting the zlib stream arbitrarily,
//    and it's annoying structurally to have PNG call ZLIB call PNG,
//    we require PNG read all the IDATs and combine them into a single
//    memory buffer

typedef struct
{
   stbi_uc *zbuffer, *zbuffer_end;
   int num_bits;
   stbi__uint32 code_buffer;

   char *zout;
   char *zout_start;
   char *zout_end;
   int   z_expandable;

   stbi__zhuffman z_length, z_distance;
} stbi__zbuf;

stbi_inline static int stbi__zeof(stbi__zbuf *z)
{
   return (z->zbuffer >= z->zbuffer_end);
}

stbi_inline static stbi_uc stbi__zget8(stbi__zbuf *z)
{
   return stbi__zeof(z) ? 0 : *z->zbuffer++;
}

static void stbi__fill_bits(stbi__zbuf *z)
{
   do {
      if (z->code_buffer >= (1U « z->num_bits)) {
        z->zbuffer = z->zbuffer_end;  /* treat this as EOF so we fail. */
        return;
      }
      z->code_buffer |= (unsigned int) stbi__zget8(z) « z->num_bits;
      z->num_bits += 8;
   } while (z->num_bits <= 24);
}

stbi_inline static unsigned int stbi__zreceive(stbi__zbuf *z, int n)
{
   unsigned int k;
   if (z->num_bits < n) stbi__fill_bits(z);
   k = z->code_buffer & ((1 « n) - 1);
   z->code_buffer »= n;
   z->num_bits -= n;
   return k;
}

static int stbi__zhuffman_decode_slowpath(stbi__zbuf *a, stbi__zhuffman *z)
```

```
{
   int b,s,k;
   // not resolved by fast table, so compute it the slow way
   // use jpeg approach, which requires MSbits at top
   k = stbi__bit_reverse(a->code_buffer, 16);
   for (s=STBI__ZFAST_BITS+1; ; ++s)
      if (k < z->maxcode[s])
         break;
   if (s >= 16) return -1; // invalid code!
   // code size is s, so:
   b = (k » (16-s)) - z->firstcode[s] + z->firstsymbol[s];
   if (b >= STBI__ZNSYMS) return -1; // some data was corrupt somewhere!
   if (z->size[b] != s) return -1;  // was originally an assert, but report failure instead.
   a->code_buffer »= s;
   a->num_bits -= s;
   return z->value[b];
}

stbi_inline static int stbi__zhuffman_decode(stbi__zbuf *a, stbi__zhuffman *z)
{
   int b,s;
   if (a->num_bits < 16) {
      if (stbi__zeof(a)) {
         return -1;   /* report error for unexpected end of data. */
      }
      stbi__fill_bits(a);
   }
   b = z->fast[a->code_buffer & STBI__ZFAST_MASK];
   if (b) {
      s = b » 9;
      a->code_buffer »= s;
      a->num_bits -= s;
      return b & 511;
   }
   return stbi__zhuffman_decode_slowpath(a, z);
}

static int stbi__zexpand(stbi__zbuf *z, char *zout, int n)  // need to make room for n bytes
{
   char *q;
   unsigned int cur, limit, old_limit;
   z->zout = zout;
   if (!z->z_expandable) return stbi__err("output buffer limit","Corrupt PNG");
   cur   = (unsigned int) (z->zout - z->zout_start);
   limit = old_limit = (unsigned) (z->zout_end - z->zout_start);
   if (UINT_MAX - cur < (unsigned) n) return stbi__err("outofmem", "Out of memory");
   while (cur + n > limit) {
      if(limit > UINT_MAX / 2) return stbi__err("outofmem", "Out of memory");
      limit *= 2;
   }
   q = (char *) STBI_REALLOC_SIZED(z->zout_start, old_limit, limit);
   STBI_NOTUSED(old_limit);
   if (q == NULL) return stbi__err("outofmem", "Out of memory");
   z->zout_start = q;
   z->zout       = q + cur;
   z->zout_end   = q + limit;
   return 1;
}

static const int stbi__zlength_base[31] = {
   3,4,5,6,7,8,9,10,11,13,
   15,17,19,23,27,31,35,43,51,59,
   67,83,99,115,131,163,195,227,258,0,0 };

static const int stbi__zlength_extra[31]=
{ 0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,0,0,0 };

static const int stbi__zdist_base[32] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,
257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,0,0};

static const int stbi__zdist_extra[32] =
{ 0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13};

static int stbi__parse_huffman_block(stbi__zbuf *a)
{
   char *zout = a->zout;
   for(;;) {
      int z = stbi__zhuffman_decode(a, &a->z_length);
      if (z < 256) {
         if (z < 0) return stbi__err("bad huffman code","Corrupt PNG"); // error in huffman codes
         if (zout >= a->zout_end) {
            if (!stbi__zexpand(a, zout, 1)) return 0;
            zout = a->zout;
         }
         *zout++ = (char) z;
      } else {
         stbi_uc *p;
         int len,dist;
         if (z == 256) {
            a->zout = zout;
            return 1;
         }
         if (z >= 286) return stbi__err("bad huffman code","Corrupt PNG"); // per DEFLATE, length codes 286 and 287 must not appear in compressed data
         z -= 257;
         len = stbi__zlength_base[z];
         if (stbi__zlength_extra[z]) len += stbi__zreceive(a, stbi__zlength_extra[z]);
         z = stbi__zhuffman_decode(a, &a->z_distance);
         if (z < 0 || z >= 30) return stbi__err("bad huffman code","Corrupt PNG"); // per DEFLATE, distance codes 30 and 31 must not appear in compressed data
         dist = stbi__zdist_base[z];
         if (stbi__zdist_extra[z]) dist += stbi__zreceive(a, stbi__zdist_extra[z]);
         if (zout - a->zout_start < dist) return stbi__err("bad dist","Corrupt PNG");
         if (zout + len > a->zout_end) {
            if (!stbi__zexpand(a, zout, len)) return 0;
            zout = a->zout;
         }
         p = (stbi_uc *) (zout - dist);
         if (dist == 1) { // run of one byte; common in images.
            stbi_uc v = *p;
            if (len) { do *zout++ = v; while (--len); }
         } else {
            if (len) { do *zout++ = *p++; while (--len); }
         }
      }
   }
}

static int stbi__compute_huffman_codes(stbi__zbuf *a)
{
   static const stbi_uc length_dezigzag[19] = { 16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15 };
   stbi__zhuffman z_codelength;
   stbi_uc lencodes[286+32+137];//padding for maximum single op
   stbi_uc codelength_sizes[19];
   int i,n;

   int hlit  = stbi__zreceive(a,5) + 257;
   int hdist = stbi__zreceive(a,5) + 1;
   int hclen = stbi__zreceive(a,4) + 4;
   int ntot  = hlit + hdist;

   memset(codelength_sizes, 0, sizeof(codelength_sizes));
   for (i=0; i < hclen; ++i) {
      int s = stbi__zreceive(a,3);
      codelength_sizes[length_dezigzag[i]] = (stbi_uc) s;
   }
   if (!stbi__zbuild_huffman(&z_codelength, codelength_sizes, 19)) return 0;

   n = 0;
   while (n < ntot) {
      int c = stbi__zhuffman_decode(a, &z_codelength);
      if (c < 0 || c >= 19) return stbi__err("bad codelengths", "Corrupt PNG");
      if (c < 16)
         lencodes[n++] = (stbi_uc) c;
      else {
         stbi_uc fill = 0;
         if (c == 16) {
            c = stbi__zreceive(a,2)+3;
            if (n == 0) return stbi__err("bad codelengths", "Corrupt PNG");
            fill = lencodes[n-1];
         } else if (c == 17) {
            c = stbi__zreceive(a,3)+3;
         } else if (c == 18) {
            c = stbi__zreceive(a,7)+11;
         } else {
            return stbi__err("bad codelengths", "Corrupt PNG");
         }
         if (ntot - n < c) return stbi__err("bad codelengths", "Corrupt PNG");
         memset(lencodes+n, fill, c);
         n += c;
      }
   }
   if (n != ntot) return stbi__err("bad codelengths","Corrupt PNG");
   if (!stbi__zbuild_huffman(&a->z_length, lencodes, hlit)) return 0;
   if (!stbi__zbuild_huffman(&a->z_distance, lencodes+hlit, hdist)) return 0;
   return 1;
}

static int stbi__parse_uncompressed_block(stbi__zbuf *a)
{
   stbi_uc header[4];
   int len,nlen,k;
   if (a->num_bits & 7)
      stbi__zreceive(a, a->num_bits & 7); // discard
   // drain the bit-packed data into header
   k = 0;
   while (a->num_bits > 0) {
      header[k++] = (stbi_uc) (a->code_buffer & 255); // suppress MSVC run-time check
      a->code_buffer »= 8;
      a->num_bits -= 8;
   }
   if (a->num_bits < 0) return stbi__err("zlib corrupt","Corrupt PNG");
   // now fill header the normal way
   while (k < 4)
      header[k++] = stbi__zget8(a);
   len  = header[1] * 256 + header[0];
   nlen = header[3] * 256 + header[2];
   if (nlen != (len ^ 0xffff)) return stbi__err("zlib corrupt","Corrupt PNG");
   if (a->zbuffer + len > a->zbuffer_end) return stbi__err("read past buffer","Corrupt PNG");
   if (a->zout + len > a->zout_end)
      if (!stbi__zexpand(a, a->zout, len)) return 0;
   memcpy(a->zout, a->zbuffer, len);
   a->zbuffer += len;
   a->zout += len;
   return 1;
}

static int stbi__parse_zlib_header(stbi__zbuf *a)
{
   int cmf   = stbi__zget8(a);
   int cm    = cmf & 15;
   /* int cinfo = cmf » 4; */
   int flg   = stbi__zget8(a);
   if (stbi__zeof(a)) return stbi__err("bad zlib header","Corrupt PNG"); // zlib spec
   if ((cmf*256+flg) % 31 != 0) return stbi__err("bad zlib header","Corrupt PNG"); // zlib spec
   if (flg & 32) return stbi__err("no preset dict","Corrupt PNG"); // preset dictionary not allowed in png
   if (cm != 8) return stbi__err("bad compression","Corrupt PNG"); // DEFLATE required for png
   // window = 1 « (8 + cinfo)... but who cares, we fully buffer output
   return 1;
}

static const stbi_uc stbi__zdefault_length[STBI__ZNSYMS] =
{
   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,  8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,  8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,  8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,  8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,  9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
   9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,  9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
   9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,  9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
   9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,  9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
   7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,  7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8
};
static const stbi_uc stbi__zdefault_distance[32] =
{
   5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
};
/*
Init algorithm:
{
   int i;   // use <= to match clearly with spec
   for (i=0; i <= 143; ++i)     stbi__zdefault_length[i]   = 8;
   for (   ; i <= 255; ++i)     stbi__zdefault_length[i]   = 9;
   for (   ; i <= 279; ++i)     stbi__zdefault_length[i]   = 7;
   for (   ; i <= 287; ++i)     stbi__zdefault_length[i]   = 8;

   for (i=0; i <=  31; ++i)     stbi__zdefault_distance[i] = 5;
}
*/

static int stbi__parse_zlib(stbi__zbuf *a, int parse_header)
{
   int final, type;
   if (parse_header)
      if (!stbi__parse_zlib_header(a)) return 0;
   a->num_bits = 0;
   a->code_buffer = 0;
   do {
      final = stbi__zreceive(a,1);
      type = stbi__zreceive(a,2);
      if (type == 0) {
         if (!stbi__parse_uncompressed_block(a)) return 0;
      } else if (type == 3) {
         return 0;
      } else {
         if (type == 1) {
            // use fixed code lengths
            if (!stbi__zbuild_huffman(&a->z_length  , stbi__zdefault_length  , STBI__ZNSYMS)) return 0;
            if (!stbi__zbuild_huffman(&a->z_distance, stbi__zdefault_distance,  32)) return 0;
         } else {
            if (!stbi__compute_huffman_codes(a)) return 0;
         }
         if (!stbi__parse_huffman_block(a)) return 0;
      }
   } while (!final);
   return 1;
}

static int stbi__do_zlib(stbi__zbuf *a, char *obuf, int olen, int exp, int parse_header)
{
   a->zout_start = obuf;
   a->zout       = obuf;
   a->zout_end   = obuf + olen;
   a->z_expandable = exp;

   return stbi__parse_zlib(a, parse_header);
}

STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int *outlen)
{
   stbi__zbuf a;
   char *p = (char *) stbi__malloc(initial_size);
   if (p == NULL) return NULL;
   a.zbuffer = (stbi_uc *) buffer;
   a.zbuffer_end = (stbi_uc *) buffer + len;
   if (stbi__do_zlib(&a, p, initial_size, 1, 1)) {
      if (outlen) *outlen = (int) (a.zout - a.zout_start);
      return a.zout_start;
   } else {
      STBI_FREE(a.zout_start);
      return NULL;
   }
}

STBIDEF char *stbi_zlib_decode_malloc(char const *buffer, int len, int *outlen)
{
   return stbi_zlib_decode_malloc_guesssize(buffer, len, 16384, outlen);
}

STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int initial_size, int *outlen, int parse_header)
{
   stbi__zbuf a;
   char *p = (char *) stbi__malloc(initial_size);
   if (p == NULL) return NULL;
   a.zbuffer = (stbi_uc *) buffer;
   a.zbuffer_end = (stbi_uc *) buffer + len;
   if (stbi__do_zlib(&a, p, initial_size, 1, parse_header)) {
      if (outlen) *outlen = (int) (a.zout - a.zout_start);
      return a.zout_start;
   } else {
      STBI_FREE(a.zout_start);
      return NULL;
   }
}

STBIDEF int stbi_zlib_decode_buffer(char *obuffer, int olen, char const *ibuffer, int ilen)
{
   stbi__zbuf a;
   a.zbuffer = (stbi_uc *) ibuffer;
   a.zbuffer_end = (stbi_uc *) ibuffer + ilen;
   if (stbi__do_zlib(&a, obuffer, olen, 0, 1))
      return (int) (a.zout - a.zout_start);
   else
      return -1;
}

STBIDEF char *stbi_zlib_decode_noheader_malloc(char const *buffer, int len, int *outlen)
{
   stbi__zbuf a;
   char *p = (char *) stbi__malloc(16384);
   if (p == NULL) return NULL;
   a.zbuffer = (stbi_uc *) buffer;
   a.zbuffer_end = (stbi_uc *) buffer+len;
   if (stbi__do_zlib(&a, p, 16384, 1, 0)) {
      if (outlen) *outlen = (int) (a.zout - a.zout_start);
      return a.zout_start;
   } else {
      STBI_FREE(a.zout_start);
      return NULL;
   }
}

STBIDEF int stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen)
{
   stbi__zbuf a;
   a.zbuffer = (stbi_uc *) ibuffer;
   a.zbuffer_end = (stbi_uc *) ibuffer + ilen;
   if (stbi__do_zlib(&a, obuffer, olen, 0, 0))
      return (int) (a.zout - a.zout_start);
   else
      return -1;
}
#endif

// public domain "baseline" PNG decoder   v0.10  Sean Barrett 2006-11-18
//    simple implementation
//      - only 8-bit samples
//      - no CRC checking
//      - allocates lots of intermediate memory
//        - avoids problem of streaming data between subsystems
//        - avoids explicit window management
//    performance
//      - uses stb_zlib, a PD zlib implementation with fast huffman decoding

#ifndef STBI_NO_PNG
typedef struct
{
   stbi__uint32 length;
   stbi__uint32 type;
} stbi__pngchunk;

static stbi__pngchunk stbi__get_chunk_header(stbi__context *s)
{
   stbi__pngchunk c;
   c.length = stbi__get32be(s);
   c.type   = stbi__get32be(s);
   return c;
}

static int stbi__check_png_header(stbi__context *s)
{
   static const stbi_uc png_sig[8] = { 137,80,78,71,13,10,26,10 };
   int i;
   for (i=0; i < 8; ++i)
      if (stbi__get8(s) != png_sig[i]) return stbi__err("bad png sig","Not a PNG");
   return 1;
}

typedef struct
{
   stbi__context *s;
   stbi_uc *idata, *expanded, *out;
   int depth;
} stbi__png;


enum {
   STBI__F_none=0,
   STBI__F_sub=1,
   STBI__F_up=2,
   STBI__F_avg=3,
   STBI__F_paeth=4,
   // synthetic filters used for first scanline to avoid needing a dummy row of 0s
   STBI__F_avg_first,
   STBI__F_paeth_first
};

static stbi_uc first_row_filter[5] =
{
   STBI__F_none,
   STBI__F_sub,
   STBI__F_none,
   STBI__F_avg_first,
   STBI__F_paeth_first
};

static int stbi__paeth(int a, int b, int c)
{
   int p = a + b - c;
   int pa = abs(p-a);
   int pb = abs(p-b);
   int pc = abs(p-c);
   if (pa <= pb && pa <= pc) return a;
   if (pb <= pc) return b;
   return c;
}

static const stbi_uc stbi__depth_scale_table[9] = { 0, 0xff, 0x55, 0, 0x11, 0,0,0, 0x01 };

// create the png data from post-deflated data
static int stbi__create_png_image_raw(stbi__png *a, stbi_uc *raw, stbi__uint32 raw_len, int out_n,
stbi__uint32 x, stbi__uint32 y, int depth, int color)
{
   int bytes = (depth == 16? 2 : 1);
   stbi__context *s = a->s;
   stbi__uint32 i,j,stride = x*out_n*bytes;
   stbi__uint32 img_len, img_width_bytes;
   int k;
   int img_n = s->img_n; // copy it into a local for later

   int output_bytes = out_n*bytes;
   int filter_bytes = img_n*bytes;
   int width = x;

   STBI_ASSERT(out_n == s->img_n || out_n == s->img_n+1);
   a->out = (stbi_uc *) stbi__malloc_mad3(x, y, output_bytes, 0); // extra bytes to write off the end into
   if (!a->out) return stbi__err("outofmem", "Out of memory");

   if (!stbi__mad3sizes_valid(img_n, x, depth, 7)) return stbi__err("too large", "Corrupt PNG");
   img_width_bytes = (((img_n * x * depth) + 7) » 3);
   img_len = (img_width_bytes + 1) * y;

   // we used to check for exact match between raw_len and img_len on non-interlaced PNGs,
   // but issue #276 reported a PNG in the wild that had extra data at the end (all zeros),
   // so just check for raw_len < img_len always.
   if (raw_len < img_len) return stbi__err("not enough pixels","Corrupt PNG");

   for (j=0; j < y; ++j) {
      stbi_uc *cur = a->out + stride*j;
      stbi_uc *prior;
      int filter = *raw++;

      if (filter > 4)
         return stbi__err("invalid filter","Corrupt PNG");

      if (depth < 8) {
         if (img_width_bytes > x) return stbi__err("invalid width","Corrupt PNG");
         cur += x*out_n - img_width_bytes; // store output to the rightmost img_len bytes, so we can decode in place
         filter_bytes = 1;
         width = img_width_bytes;
      }
      prior = cur - stride; // bugfix: need to compute this after ’cur +=’ computation above

      // if first row, use special filter that doesn’t sample previous row
      if (j == 0) filter = first_row_filter[filter];

      // handle first byte explicitly
      for (k=0; k < filter_bytes; ++k) {
         switch (filter) {
            case STBI__F_none       : cur[k] = raw[k]; break;
            case STBI__F_sub        : cur[k] = raw[k]; break;
            case STBI__F_up         : cur[k] = STBI__BYTECAST(raw[k] + prior[k]); break;
            case STBI__F_avg        : cur[k] = STBI__BYTECAST(raw[k] + (prior[k]»1)); break;
            case STBI__F_paeth      : cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(0,prior[k],0)); break;
            case STBI__F_avg_first  : cur[k] = raw[k]; break;
            case STBI__F_paeth_first: cur[k] = raw[k]; break;
         }
      }

      if (depth == 8) {
         if (img_n != out_n)
            cur[img_n] = 255; // first pixel
         raw += img_n;
         cur += out_n;
         prior += out_n;
      } else if (depth == 16) {
         if (img_n != out_n) {
            cur[filter_bytes]   = 255; // first pixel top byte
            cur[filter_bytes+1] = 255; // first pixel bottom byte
         }
         raw += filter_bytes;
         cur += output_bytes;
         prior += output_bytes;
      } else {
         raw += 1;
         cur += 1;
         prior += 1;
      }

      // this is a little gross, so that we don’t switch per-pixel or per-component
      if (depth < 8 || img_n == out_n) {
         int nk = (width - 1)*filter_bytes;
         #define STBI__CASE(f) \
             case f:     \
                for (k=0; k < nk; ++k)
         switch (filter) {
            // "none" filter turns into a memcpy here; make that explicit.
            case STBI__F_none:         memcpy(cur, raw, nk); break;
            STBI__CASE(STBI__F_sub)          { cur[k] = STBI__BYTECAST(raw[k] + cur[k-filter_bytes]); } break;
            STBI__CASE(STBI__F_up)           { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
            STBI__CASE(STBI__F_avg)          { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k-filter_bytes])»1)); } break;
            STBI__CASE(STBI__F_paeth)        { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k-filter_bytes],prior[k],prior[k-filter_bytes])); } break;
            STBI__CASE(STBI__F_avg_first)    { cur[k] = STBI__BYTECAST(raw[k] + (cur[k-filter_bytes] » 1)); } break;
            STBI__CASE(STBI__F_paeth_first)  { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k-filter_bytes],0,0)); } break;
         }
         #undef STBI__CASE
         raw += nk;
      } else {
         STBI_ASSERT(img_n+1 == out_n);
         #define STBI__CASE(f) \
             case f:     \
                for (i=x-1; i >= 1; --i, cur[filter_bytes]=255,raw+=filter_bytes,cur+=output_bytes,prior+=output_bytes) \
                   for (k=0; k < filter_bytes; ++k)
         switch (filter) {
            STBI__CASE(STBI__F_none)         { cur[k] = raw[k]; } break;
            STBI__CASE(STBI__F_sub)          { cur[k] = STBI__BYTECAST(raw[k] + cur[k- output_bytes]); } break;
            STBI__CASE(STBI__F_up)           { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
            STBI__CASE(STBI__F_avg)          { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k- output_bytes])»1)); } break;
            STBI__CASE(STBI__F_paeth)        { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k- output_bytes],prior[k],prior[k- output_bytes])); } break;
            STBI__CASE(STBI__F_avg_first)    { cur[k] = STBI__BYTECAST(raw[k] + (cur[k- output_bytes] » 1)); } break;
            STBI__CASE(STBI__F_paeth_first)  { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k- output_bytes],0,0)); } break;
         }
         #undef STBI__CASE

         // the loop above sets the high byte of the pixels' alpha, but for
         // 16 bit png files we also need the low byte set. we'll do that here.
         if (depth == 16) {
            cur = a->out + stride*j; // start at the beginning of the row again
            for (i=0; i < x; ++i,cur+=output_bytes) {
               cur[filter_bytes+1] = 255;
            }
         }
      }
   }

   // we make a separate pass to expand bits to pixels; for performance,
   // this could run two scanlines behind the above code, so it won't
   // intefere with filtering but will still be in the cache.
   if (depth < 8) {
      for (j=0; j < y; ++j) {
         stbi_uc *cur = a->out + stride*j;
         stbi_uc *in  = a->out + stride*j + x*out_n - img_width_bytes;
         // unpack 1/2/4-bit into a 8-bit buffer. allows us to keep the common 8-bit path optimal at minimal cost for 1/2/4-bit
         // png guarante byte alignment, if width is not multiple of 8/4/2 we'll decode dummy trailing data that will be skipped in the later loop
         stbi_uc scale = (color == 0) ? stbi__depth_scale_table[depth] : 1; // scale grayscale values to 0..255 range

         // note that the final byte might overshoot and write more data than desired.
         // we can allocate enough data that this never writes out of memory, but it
         // could also overwrite the next scanline. can it overwrite non-empty data
         // on the next scanline? yes, consider 1-pixel-wide scanlines with 1-bit-per-pixel.
         // so we need to explicitly clamp the final ones

         if (depth == 4) {
            for (k=x*img_n; k >= 2; k-=2, ++in) {
               *cur++ = scale * ((*in » 4)       );
               *cur++ = scale * ((*in      ) & 0x0f);
            }
            if (k > 0) *cur++ = scale * ((*in » 4)       );
         } else if (depth == 2) {
            for (k=x*img_n; k >= 4; k-=4, ++in) {
               *cur++ = scale * ((*in » 6)       );
               *cur++ = scale * ((*in » 4) & 0x03);
               *cur++ = scale * ((*in » 2) & 0x03);
               *cur++ = scale * ((*in      ) & 0x03);
            }
            if (k > 0) *cur++ = scale * ((*in » 6)       );
            if (k > 1) *cur++ = scale * ((*in » 4) & 0x03);
            if (k > 2) *cur++ = scale * ((*in » 2) & 0x03);
         } else if (depth == 1) {
            for (k=x*img_n; k >= 8; k-=8, ++in) {
               *cur++ = scale * ((*in » 7)       );
               *cur++ = scale * ((*in » 6) & 0x01);
               *cur++ = scale * ((*in » 5) & 0x01);
               *cur++ = scale * ((*in » 4) & 0x01);
               *cur++ = scale * ((*in » 3) & 0x01);
               *cur++ = scale * ((*in » 2) & 0x01);
               *cur++ = scale * ((*in » 1) & 0x01);
               *cur++ = scale * ((*in     ) & 0x01);
            }
            if (k > 0) *cur++ = scale * ((*in » 7)       );
            if (k > 1) *cur++ = scale * ((*in » 6) & 0x01);
            if (k > 2) *cur++ = scale * ((*in » 5) & 0x01);
            if (k > 3) *cur++ = scale * ((*in » 4) & 0x01);
            if (k > 4) *cur++ = scale * ((*in » 3) & 0x01);
            if (k > 5) *cur++ = scale * ((*in » 2) & 0x01);
            if (k > 6) *cur++ = scale * ((*in » 1) & 0x01);
         }
         if (img_n != out_n) {
            int q;
            // insert alpha = 255
            cur = a->out + stride*j;
            if (img_n == 1) {
               for (q=x-1; q >= 0; --q) {
                  cur[q*2+1] = 255;
                  cur[q*2+0] = cur[q];
               }
            } else {
               STBI_ASSERT(img_n == 3);
               for (q=x-1; q >= 0; --q) {
                  cur[q*4+3] = 255;
                  cur[q*4+2] = cur[q*3+2];
                  cur[q*4+1] = cur[q*3+1];
                  cur[q*4+0] = cur[q*3+0];
               }
            }
         }
      }
   } else if (depth == 16) {
      // force the image data from big-endian to platform-native.
      // this is done in a separate pass due to the decoding relying
      // on the data being untouched, but could probably be done
      // per-line during decode if care is taken.
      stbi_uc *cur = a->out;
      stbi__uint16 *cur16 = (stbi__uint16*)cur;

      for(i=0; i < x*y*out_n; ++i,cur16++,cur+=2) {
         *cur16 = (cur[0] « 8) | cur[1];
      }
   }

   return 1;
}

static int stbi__create_png_image(stbi__png *a, stbi_uc *image_data, stbi__uint32 image_data_len, int out_n, int depth, int color, int interlaced)
{
   int bytes = (depth == 16 ? 2 : 1);
   int out_bytes = out_n * bytes;
   stbi_uc *final;
   int p;
   if (!interlaced)
      return stbi__create_png_image_raw(a, image_data, image_data_len, out_n, a->s->img_x, a->s->img_y, depth, color);

   // de-interlacing
   final = (stbi_uc *) stbi__malloc_mad3(a->s->img_x, a->s->img_y, out_bytes, 0);
   if (!final) return stbi__err("outofmem", "Out of memory");
   for (p=0; p < 7; ++p) {
      int xorig[] = { 0,4,0,2,0,1,0 };
      int yorig[] = { 0,0,4,0,2,0,1 };
      int xspc[]  = { 8,8,4,4,2,2,1 };
      int yspc[]  = { 8,8,8,4,4,2,2 };
      int i,j,x,y;
      // pass1_x[4] = 0, pass1_x[5] = 1, pass1_x[12] = 1
      x = (a->s->img_x - xorig[p] + xspc[p]-1) / xspc[p];
      y = (a->s->img_y - yorig[p] + yspc[p]-1) / yspc[p];
      if (x && y) {
         stbi__uint32 img_len = ((((a->s->img_n * x * depth) + 7) » 3) + 1) * y;
         if (!stbi__create_png_image_raw(a, image_data, image_data_len, out_n, x, y, depth, color)) {
            STBI_FREE(final);
            return 0;
         }
         for (j=0; j < y; ++j) {
            for (i=0; i < x; ++i) {
               int out_y = j*yspc[p]+yorig[p];
               int out_x = i*xspc[p]+xorig[p];
               memcpy(final + out_y*a->s->img_x*out_bytes + out_x*out_bytes,
                      a->out + (j*x+i)*out_bytes, out_bytes);
            }
         }
         STBI_FREE(a->out);
         image_data += img_len;
         image_data_len -= img_len;
      }
   }
   a->out = final;

   return 1;
}

static int stbi__compute_transparency(stbi__png *z, stbi_uc tc[3], int out_n)
{
   stbi__context *s = z->s;
   stbi__uint32 i, pixel_count = s->img_x * s->img_y;
   stbi_uc *p = z->out;

   // compute color-based transparency, assuming we’ve
   // already got 255 as the alpha value in the output
   STBI_ASSERT(out_n == 2 || out_n == 4);

   if (out_n == 2) {
      for (i=0; i < pixel_count; ++i) {
         p[1] = (p[0] == tc[0] ? 0 : 255);
         p += 2;
      }
   } else {
      for (i=0; i < pixel_count; ++i) {
         if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
            p[3] = 0;
         p += 4;
      }
   }
   return 1;
}

static int stbi__compute_transparency16(stbi__png *z, stbi__uint16 tc[3], int out_n)
{
   stbi__context *s = z->s;
   stbi__uint32 i, pixel_count = s->img_x * s->img_y;
   stbi__uint16 *p = (stbi__uint16*) z->out;

   // compute color-based transparency, assuming we’ve
   // already got 65535 as the alpha value in the output
   STBI_ASSERT(out_n == 2 || out_n == 4);

   if (out_n == 2) {
      for (i = 0; i < pixel_count; ++i) {
         p[1] = (p[0] == tc[0] ? 0 : 65535);
         p += 2;
      }
   } else {
      for (i = 0; i < pixel_count; ++i) {
         if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
            p[3] = 0;
         p += 4;
      }
   }
   return 1;
}

static int stbi__expand_png_palette(stbi__png *a, stbi_uc *palette, int len, int pal_img_n)
{
   stbi__uint32 i, pixel_count = a->s->img_x * a->s->img_y;
   stbi_uc *p, *temp_out, *orig = a->out;

   p = (stbi_uc *) stbi__malloc_mad2(pixel_count, pal_img_n, 0);
   if (p == NULL) return stbi__err("outofmem", "Out of memory");

   // between here and free(out) below, exitting would leak
   temp_out = p;

   if (pal_img_n == 3) {
      for (i=0; i < pixel_count; ++i) {
         int n = orig[i]*4;
         p[0] = palette[n  ];
         p[1] = palette[n+1];
         p[2] = palette[n+2];
         p += 3;
      }
   } else {
      for (i=0; i < pixel_count; ++i) {
         int n = orig[i]*4;
         p[0] = palette[n  ];
         p[1] = palette[n+1];
         p[2] = palette[n+2];
         p[3] = palette[n+3];
         p += 4;
      }
   }
   STBI_FREE(a->out);
   a->out = temp_out;

   STBI_NOTUSED(len);

   return 1;
}

static int stbi__unpremultiply_on_load_global = 0;
static int stbi__de_iphone_flag_global = 0;

STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply)
{
   stbi__unpremultiply_on_load_global = flag_true_if_should_unpremultiply;
}

STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert)
{
   stbi__de_iphone_flag_global = flag_true_if_should_convert;
}

#ifndef STBI_THREAD_LOCAL
#define stbi__unpremultiply_on_load  stbi__unpremultiply_on_load_global
#define stbi__de_iphone_flag  stbi__de_iphone_flag_global
#else
static STBI_THREAD_LOCAL int stbi__unpremultiply_on_load_local, stbi__unpremultiply_on_load_set;
static STBI_THREAD_LOCAL int stbi__de_iphone_flag_local, stbi__de_iphone_flag_set;

STBIDEF void stbi_set_unpremultiply_on_load_thread(int flag_true_if_should_unpremultiply)
{
   stbi__unpremultiply_on_load_local = flag_true_if_should_unpremultiply;
   stbi__unpremultiply_on_load_set = 1;
}

STBIDEF void stbi_convert_iphone_png_to_rgb_thread(int flag_true_if_should_convert)
{
   stbi__de_iphone_flag_local = flag_true_if_should_convert;
   stbi__de_iphone_flag_set = 1;
}

#define stbi__unpremultiply_on_load  (stbi__unpremultiply_on_load_set           \
                                       ? stbi__unpremultiply_on_load_local      \
                                       : stbi__unpremultiply_on_load_global)
#define stbi__de_iphone_flag  (stbi__de_iphone_flag_set                         \
                                ? stbi__de_iphone_flag_local                    \
                                : stbi__de_iphone_flag_global)
#endif // STBI_THREAD_LOCAL

static void stbi__de_iphone(stbi__png *z)
{
   stbi__context *s = z->s;
   stbi__uint32 i, pixel_count = s->img_x * s->img_y;
   stbi_uc *p = z->out;

   if (s->img_out_n == 3) {  // convert bgr to rgb
      for (i=0; i < pixel_count; ++i) {
         stbi_uc t = p[0];
         p[0] = p[2];
         p[2] = t;
         p += 3;
      }
   } else {
      STBI_ASSERT(s->img_out_n == 4);
      if (stbi__unpremultiply_on_load) {
         // convert bgr to rgb and unpremultiply
         for (i=0; i < pixel_count; ++i) {
            stbi_uc a = p[3];
            stbi_uc t = p[0];
            if (a) {
               stbi_uc half = a / 2;
               p[0] = (p[2] * 255 + half) / a;
               p[1] = (p[1] * 255 + half) / a;
               p[2] = ( t   * 255 + half) / a;
            } else {
               p[0] = p[2];
               p[2] = t;
            }
            p += 4;
         }
      } else {
         // convert bgr to rgb
         for (i=0; i < pixel_count; ++i) {
            stbi_uc t = p[0];
            p[0] = p[2];
            p[2] = t;
            p += 4;
         }
      }
   }
}

#define STBI__PNG_TYPE(a,b,c,d)  (((unsigned) (a) « 24) + ((unsigned) (b) « 16) + ((unsigned) (c) « 8) + (unsigned) (d))

static int stbi__parse_png_file(stbi__png *z, int scan, int req_comp)
{
   stbi_uc palette[1024], pal_img_n=0;
   stbi_uc has_trans=0, tc[3]={0};
   stbi__uint16 tc16[3];
   stbi__uint32 ioff=0, idata_limit=0, i, pal_len=0;
   int first=1,k,interlace=0, color=0, is_iphone=0;
   stbi__context *s = z->s;

   z->expanded = NULL;
   z->idata = NULL;
   z->out = NULL;

   if (!stbi__check_png_header(s)) return 0;

   if (scan == STBI__SCAN_type) return 1;

   for (;;) {
      stbi__pngchunk c = stbi__get_chunk_header(s);
      switch (c.type) {
         case STBI__PNG_TYPE(’C’,’g’,’B’,’I’):
            is_iphone = 1;
            stbi__skip(s, c.length);
            break;
         case STBI__PNG_TYPE(’I’,’H’,’D’,’R’): {
            int comp,filter;
            if (!first) return stbi__err("multiple IHDR","Corrupt PNG");
            first = 0;
            if (c.length != 13) return stbi__err("bad IHDR len","Corrupt PNG");
            s->img_x = stbi__get32be(s);
            s->img_y = stbi__get32be(s);
            if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
            if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
            z->depth = stbi__get8(s);  if (z->depth != 1 && z->depth != 2 && z->depth != 4 && z->depth != 8 && z->depth != 16)  return stbi__err("1/2/4/8/16-bit only","PNG not supported: 1/2/4/8/16-bit only");
            color = stbi__get8(s);  if (color > 6)         return stbi__err("bad ctype","Corrupt PNG");
            if (color == 3 && z->depth == 16)                  return stbi__err("bad ctype","Corrupt PNG");
            if (color == 3) pal_img_n = 3; else if (color & 1) return stbi__err("bad ctype","Corrupt PNG");
            comp  = stbi__get8(s);  if (comp) return stbi__err("bad comp method","Corrupt PNG");
            filter= stbi__get8(s);  if (filter) return stbi__err("bad filter method","Corrupt PNG");
            interlace = stbi__get8(s); if (interlace>1) return stbi__err("bad interlace method","Corrupt PNG");
            if (!s->img_x || !s->img_y) return stbi__err("0-pixel image","Corrupt PNG");
            if (!pal_img_n) {
               s->img_n = (color & 2 ? 3 : 1) + (color & 4 ? 1 : 0);
               if ((1 « 30) / s->img_x / s->img_n < s->img_y) return stbi__err("too large", "Image too large to decode");
            } else {
               // if paletted, then pal_n is our final components, and
               // img_n is # components to decompress/filter.
               s->img_n = 1;
               if ((1 « 30) / s->img_x / 4 < s->img_y) return stbi__err("too large","Corrupt PNG");
            }
            // even with SCAN_header, have to scan to see if we have a tRNS
            break;
         }

         case STBI__PNG_TYPE(’P’,’L’,’T’,’E’):  {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (c.length > 256*3) return stbi__err("invalid PLTE","Corrupt PNG");
            pal_len = c.length / 3;
            if (pal_len * 3 != c.length) return stbi__err("invalid PLTE","Corrupt PNG");
            for (i=0; i < pal_len; ++i) {
               palette[i*4+0] = stbi__get8(s);
               palette[i*4+1] = stbi__get8(s);
               palette[i*4+2] = stbi__get8(s);
               palette[i*4+3] = 255;
            }
            break;
         }

         case STBI__PNG_TYPE('t','R','N','S'): {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (z->idata) return stbi__err("tRNS after IDAT","Corrupt PNG");
            if (pal_img_n) {
               if (scan == STBI__SCAN_header) { s->img_n = 4; return 1; }
               if (pal_len == 0) return stbi__err("tRNS before PLTE","Corrupt PNG");
               if (c.length > pal_len) return stbi__err("bad tRNS len","Corrupt PNG");
               pal_img_n = 4;
               for (i=0; i < c.length; ++i)
                  palette[i*4+3] = stbi__get8(s);
            } else {
               if (!(s->img_n & 1)) return stbi__err("tRNS with alpha","Corrupt PNG");
               if (c.length != (stbi__uint32) s->img_n*2) return stbi__err("bad tRNS len","Corrupt PNG");
               has_trans = 1;
               // non-paletted with tRNS = constant alpha. if header-scanning, we can stop now.
               if (scan == STBI__SCAN_header) { ++s->img_n; return 1; }
               if (z->depth == 16) {
                  for (k = 0; k < s->img_n; ++k) tc16[k] = (stbi__uint16)stbi__get16be(s); // copy the values as-is
               } else {
                  for (k = 0; k < s->img_n; ++k) tc[k] = (stbi_uc)(stbi__get16be(s) & 255) * stbi__depth_scale_table[z->depth]; // non 8-bit images will be larger
               }
            }
            break;
         }

         case STBI__PNG_TYPE('I','D','A','T'): {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (pal_img_n && !pal_len) return stbi__err("no PLTE","Corrupt PNG");
            if (scan == STBI__SCAN_header) {
               // header scan definitely stops at first IDAT
               if (pal_img_n)
                  s->img_n = pal_img_n;
               return 1;
            }
            if (c.length > (1u « 30)) return stbi__err("IDAT size limit", "IDAT section larger than 2^30 bytes");
            if ((int)(ioff + c.length) < (int)ioff) return 0;
            if (ioff + c.length > idata_limit) {
               stbi__uint32 idata_limit_old = idata_limit;
               stbi_uc *p;
               if (idata_limit == 0) idata_limit = c.length > 4096 ? c.length : 4096;
               while (ioff + c.length > idata_limit)
                  idata_limit *= 2;
               STBI_NOTUSED(idata_limit_old);
               p = (stbi_uc *) STBI_REALLOC_SIZED(z->idata, idata_limit_old, idata_limit); if (p == NULL) return stbi__err("outofmem", "Out of memory");
               z->idata = p;
            }
            if (!stbi__getn(s, z->idata+ioff,c.length)) return stbi__err("outofdata","Corrupt PNG");
            ioff += c.length;
            break;
         }

         case STBI__PNG_TYPE('I','E','N','D'): {
            stbi__uint32 raw_len, bpl;
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (scan != STBI__SCAN_load) return 1;
            if (z->idata == NULL) return stbi__err("no IDAT","Corrupt PNG");
            // initial guess for decoded data size to avoid unnecessary reallocs
            bpl = (s->img_x * z->depth + 7) / 8; // bytes per line, per component
            raw_len = bpl * s->img_y * s->img_n /* pixels */ + s->img_y /* filter mode per row */;
            z->expanded = (stbi_uc *) stbi_zlib_decode_malloc_guesssize_headerflag((char *) z->idata, ioff, raw_len, (int *) &raw_len, !is_iphone);
            if (z->expanded == NULL) return 0; // zlib should set error
            STBI_FREE(z->idata); z->idata = NULL;
            if ((req_comp == s->img_n+1 && req_comp != 3 && !pal_img_n) || has_trans)
               s->img_out_n = s->img_n+1;
            else
               s->img_out_n = s->img_n;
            if (!stbi__create_png_image(z, z->expanded, raw_len, s->img_out_n, z->depth, color, interlace)) return 0;
            if (has_trans) {
               if (z->depth == 16) {
                  if (!stbi__compute_transparency16(z, tc16, s->img_out_n)) return 0;
               } else {
                  if (!stbi__compute_transparency(z, tc, s->img_out_n)) return 0;
               }
            }
            if (is_iphone && stbi__de_iphone_flag && s->img_out_n > 2)
               stbi__de_iphone(z);
            if (pal_img_n) {
               // pal_img_n == 3 or 4
               s->img_n = pal_img_n; // record the actual colors we had
               s->img_out_n = pal_img_n;
               if (req_comp >= 3) s->img_out_n = req_comp;
               if (!stbi__expand_png_palette(z, palette, pal_len, s->img_out_n))
                  return 0;
            } else if (has_trans) {
               // non-paletted image with tRNS -> source image has (constant) alpha
               ++s->img_n;
            }
            STBI_FREE(z->expanded); z->expanded = NULL;
            // end of PNG chunk, read and skip CRC
            stbi__get32be(s);
            return 1;
         }

         default:
            // if critical, fail
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if ((c.type & (1 « 29)) == 0) {
               #ifndef STBI_NO_FAILURE_STRINGS
               // not threadsafe
               static char invalid_chunk[] = "XXXX PNG chunk not known";
               invalid_chunk[0] = STBI__BYTECAST(c.type » 24);
               invalid_chunk[1] = STBI__BYTECAST(c.type » 16);
               invalid_chunk[2] = STBI__BYTECAST(c.type »  8);
               invalid_chunk[3] = STBI__BYTECAST(c.type »  0);
               #endif
               return stbi__err(invalid_chunk, "PNG not supported: unknown PNG chunk type");
            }
            stbi__skip(s, c.length);
            break;
      }
      // end of PNG chunk, read and skip CRC
      stbi__get32be(s);
   }
}

static void *stbi__do_png(stbi__png *p, int *x, int *y, int *n, int req_comp, stbi__result_info *ri)
{
   void *result=NULL;
   if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
   if (stbi__parse_png_file(p, STBI__SCAN_load, req_comp)) {
      if (p->depth <= 8)
         ri->bits_per_channel = 8;
      else if (p->depth == 16)
         ri->bits_per_channel = 16;
      else
         return stbi__errpuc("bad bits_per_channel", "PNG not supported: unsupported color depth");
      result = p->out;
      p->out = NULL;
      if (req_comp && req_comp != p->s->img_out_n) {
         if (ri->bits_per_channel == 8)
            result = stbi__convert_format((unsigned char *) result, p->s->img_out_n, req_comp,
p->s->img_x, p->s->img_y);
         else
            result = stbi__convert_format16((stbi__uint16 *) result, p->s->img_out_n, req_comp,
p->s->img_x, p->s->img_y);
         p->s->img_out_n = req_comp;
         if (result == NULL) return result;
      }
      *x = p->s->img_x;
      *y = p->s->img_y;
      if (n) *n = p->s->img_n;
   }
   STBI_FREE(p->out);      p->out      = NULL;
   STBI_FREE(p->expanded); p->expanded = NULL;
   STBI_FREE(p->idata);    p->idata    = NULL;

   return result;
}

static void *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri)
{
   stbi__png p;
   p.s = s;
   return stbi__do_png(&p, x,y,comp,req_comp, ri);
}

static int stbi__png_test(stbi__context *s)
{
   int r;
   r = stbi__check_png_header(s);
   stbi__rewind(s);
   return r;
```

```
}

static int stbi__png_info_raw(stbi__png *p, int *x, int *y, int *comp)
{
   if (!stbi__parse_png_file(p, STBI__SCAN_header, 0)) {
      stbi__rewind( p->s );
      return 0;
   }
   if (x) *x = p->s->img_x;
   if (y) *y = p->s->img_y;
   if (comp) *comp = p->s->img_n;
   return 1;
}

static int stbi__png_info(stbi__context *s, int *x, int *y, int *comp)
{
   stbi__png p;
   p.s = s;
   return stbi__png_info_raw(&p, x, y, comp);
}

static int stbi__png_is16(stbi__context *s)
{
   stbi__png p;
   p.s = s;
   if (!stbi__png_info_raw(&p, NULL, NULL, NULL))
       return 0;
   if (p.depth != 16) {
      stbi__rewind(p.s);
      return 0;
   }
   return 1;
}
#endif

// Microsoft/Windows BMP image

#ifndef STBI_NO_BMP
static int stbi__bmp_test_raw(stbi__context *s)
{
   int r;
   int sz;
   if (stbi__get8(s) != ’B’) return 0;
   if (stbi__get8(s) != ’M’) return 0;
   stbi__get32le(s); // discard filesize
   stbi__get16le(s); // discard reserved
   stbi__get16le(s); // discard reserved
   stbi__get32le(s); // discard data offset
   sz = stbi__get32le(s);
   r = (sz == 12 || sz == 40 || sz == 56 || sz == 108 || sz == 124);
   return r;
}

static int stbi__bmp_test(stbi__context *s)
{
   int r = stbi__bmp_test_raw(s);
   stbi__rewind(s);
   return r;
}


// returns 0..31 for the highest set bit
static int stbi__high_bit(unsigned int z)
{
   int n=0;
   if (z == 0) return -1;
   if (z >= 0x10000) { n += 16; z »= 16; }
   if (z >= 0x00100) { n +=  8; z »=  8; }
   if (z >= 0x00010) { n +=  4; z »=  4; }
   if (z >= 0x00004) { n +=  2; z »=  2; }
   if (z >= 0x00002) { n +=  1;/* »=  1;*/ }
   return n;
}

static int stbi__bitcount(unsigned int a)
{
   a = (a & 0x55555555) + ((a »  1) & 0x55555555); // max 2
   a = (a & 0x33333333) + ((a »  2) & 0x33333333); // max 4
   a = (a + (a » 4)) & 0x0f0f0f0f; // max 8 per 4, now 8 bits
   a = (a + (a » 8)); // max 16 per 8 bits
   a = (a + (a » 16)); // max 32 per 8 bits
   return a & 0xff;
}

// extract an arbitrarily-aligned N-bit value (N=bits)
// from v, and then make it 8-bits long and fractionally
// extend it to full full range.
```

```
static int stbi__shiftsigned(unsigned int v, int shift, int bits)
{
   static unsigned int mul_table[9] = {
      0,
      0xff/*0b11111111*/, 0x55/*0b01010101*/, 0x49/*0b01001001*/, 0x11/*0b00010001*/,
      0x21/*0b00100001*/, 0x41/*0b01000001*/, 0x81/*0b10000001*/, 0x01/*0b00000001*/,
   };
   static unsigned int shift_table[9] = {
      0, 0,0,1,0,2,4,6,0,
   };
   if (shift < 0)
      v «= -shift;
   else
      v »= shift;
   STBI_ASSERT(v < 256);
   v »= (8-bits);
   STBI_ASSERT(bits >= 0 && bits <= 8);
   return (int) ((unsigned) v * mul_table[bits]) » shift_table[bits];
}

typedef struct
{
   int bpp, offset, hsz;
   unsigned int mr,mg,mb,ma, all_a;
   int extra_read;
} stbi__bmp_data;

static int stbi__bmp_set_mask_defaults(stbi__bmp_data *info, int compress)
{
   // BI_BITFIELDS specifies masks explicitly, don’t override
   if (compress == 3)
      return 1;

   if (compress == 0) {
      if (info->bpp == 16) {
         info->mr = 31u « 10;
         info->mg = 31u «  5;
         info->mb = 31u «  0;
      } else if (info->bpp == 32) {
         info->mr = 0xffu « 16;
         info->mg = 0xffu «  8;
         info->mb = 0xffu «  0;
         info->ma = 0xffu « 24;
         info->all_a = 0; // if all_a is 0 at end, then we loaded alpha channel but it was all 0
      } else {
         // otherwise, use defaults, which is all-0
         info->mr = info->mg = info->mb = info->ma = 0;
      }
      return 1;
   }
   return 0; // error
}

static void *stbi__bmp_parse_header(stbi__context *s, stbi__bmp_data *info)
{
   int hsz;
   if (stbi__get8(s) != ’B’ || stbi__get8(s) != ’M’) return stbi__errpuc("not BMP", "Corrupt BMP");
   stbi__get32le(s); // discard filesize
   stbi__get16le(s); // discard reserved
   stbi__get16le(s); // discard reserved
   info->offset = stbi__get32le(s);
   info->hsz = hsz = stbi__get32le(s);
   info->mr = info->mg = info->mb = info->ma = 0;
   info->extra_read = 14;

   if (info->offset < 0) return stbi__errpuc("bad BMP", "bad BMP");

   if (hsz != 12 && hsz != 40 && hsz != 56 && hsz != 108 && hsz != 124) return stbi__errpuc("unknown BMP", "BMP type not supported: unknown");
   if (hsz == 12) {
      s->img_x = stbi__get16le(s);
      s->img_y = stbi__get16le(s);
   } else {
      s->img_x = stbi__get32le(s);
      s->img_y = stbi__get32le(s);
   }
   if (stbi__get16le(s) != 1) return stbi__errpuc("bad BMP", "bad BMP");
   info->bpp = stbi__get16le(s);
   if (hsz != 12) {
      int compress = stbi__get32le(s);
      if (compress == 1 || compress == 2) return stbi__errpuc("BMP RLE", "BMP type not supported: RLE");
      if (compress >= 4) return stbi__errpuc("BMP JPEG/PNG", "BMP type not supported: unsupported compression"); // this includes PNG/JPEG modes
      if (compress == 3 && info->bpp != 16 && info->bpp != 32) return stbi__errpuc("bad BMP", "bad BMP"); // bitfields requires 16 or 32 bits/pixel
      stbi__get32le(s); // discard sizeof
      stbi__get32le(s); // discard hres
      stbi__get32le(s); // discard vres
      stbi__get32le(s); // discard colorsused
      stbi__get32le(s); // discard max important
      if (hsz == 40 || hsz == 56) {
         if (hsz == 56) {
            stbi__get32le(s);
            stbi__get32le(s);
            stbi__get32le(s);
            stbi__get32le(s);
         }
         if (info->bpp == 16 || info->bpp == 32) {
            if (compress == 0) {
               stbi__bmp_set_mask_defaults(info, compress);
            } else if (compress == 3) {
               info->mr = stbi__get32le(s);
               info->mg = stbi__get32le(s);
               info->mb = stbi__get32le(s);
               info->extra_read += 12;
               // not documented, but generated by photoshop and handled by mspaint
               if (info->mr == info->mg && info->mg == info->mb) {
                  // ?!?!?
                  return stbi__errpuc("bad BMP", "bad BMP");
               }
            } else
               return stbi__errpuc("bad BMP", "bad BMP");
         }
      } else {
         // V4/V5 header
         int i;
         if (hsz != 108 && hsz != 124)
            return stbi__errpuc("bad BMP", "bad BMP");
         info->mr = stbi__get32le(s);
         info->mg = stbi__get32le(s);
         info->mb = stbi__get32le(s);
         info->ma = stbi__get32le(s);
         if (compress != 3) // override mr/mg/mb unless in BI_BITFIELDS mode, as per docs
            stbi__bmp_set_mask_defaults(info, compress);
         stbi__get32le(s); // discard color space
         for (i=0; i < 12; ++i)
            stbi__get32le(s); // discard color space parameters
         if (hsz == 124) {
            stbi__get32le(s); // discard rendering intent
            stbi__get32le(s); // discard offset of profile data
            stbi__get32le(s); // discard size of profile data
            stbi__get32le(s); // discard reserved
         }
      }
   }
   return (void *) 1;
}


static void *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   stbi_uc *out;
   unsigned int mr=0,mg=0,mb=0,ma=0, all_a;
   stbi_uc pal[256][4];
   int psize=0,i,j,width;
   int flip_vertically, pad, target;
   stbi__bmp_data info;
   STBI_NOTUSED(ri);

   info.all_a = 255;
   if (stbi__bmp_parse_header(s, &info) == NULL)
      return NULL; // error code already set

   flip_vertically = ((int) s->img_y) > 0;
   s->img_y = abs((int) s->img_y);

   if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
   if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");

   mr = info.mr;
   mg = info.mg;
   mb = info.mb;
   ma = info.ma;
   all_a = info.all_a;

   if (info.hsz == 12) {
      if (info.bpp < 24)
         psize = (info.offset - info.extra_read - 24) / 3;
   } else {
      if (info.bpp < 16)
         psize = (info.offset - info.extra_read - info.hsz) » 2;
   }
   if (psize == 0) {
      // accept some number of extra bytes after the header, but if the offset points either to before
      // the header ends or implies a large amount of extra data, reject the file as malformed
      int bytes_read_so_far = s->callback_already_read + (int)(s->img_buffer - s->img_buffer_original);
      int header_limit = 1024; // max we actually read is below 256 bytes currently.
      int extra_data_limit = 256*4; // what ordinarily goes here is a palette; 256 entries*4 bytes is its max size.
      if (bytes_read_so_far <= 0 || bytes_read_so_far > header_limit) {
         return stbi__errpuc("bad header", "Corrupt BMP");
      }
      // we established that bytes_read_so_far is positive and sensible.
      // the first half of this test rejects offsets that are either too small positives, or
      // negative, and guarantees that info.offset >= bytes_read_so_far > 0. this in turn
      // ensures the number computed in the second half of the test can't overflow.
      if (info.offset < bytes_read_so_far || info.offset - bytes_read_so_far > extra_data_limit) {
         return stbi__errpuc("bad offset", "Corrupt BMP");
      } else {
         stbi__skip(s, info.offset - bytes_read_so_far);
      }
   }

   if (info.bpp == 24 && ma == 0xff000000)
      s->img_n = 3;
   else
      s->img_n = ma ? 4 : 3;
   if (req_comp && req_comp >= 3) // we can directly decode 3 or 4
      target = req_comp;
   else
      target = s->img_n; // if they want monochrome, we'll post-convert

   // sanity-check size
   if (!stbi__mad3sizes_valid(target, s->img_x, s->img_y, 0))
      return stbi__errpuc("too large", "Corrupt BMP");

   out = (stbi_uc *) stbi__malloc_mad3(target, s->img_x, s->img_y, 0);
   if (!out) return stbi__errpuc("outofmem", "Out of memory");
   if (info.bpp < 16) {
      int z=0;
      if (psize == 0 || psize > 256) { STBI_FREE(out); return stbi__errpuc("invalid", "Corrupt BMP"); }
      for (i=0; i < psize; ++i) {
         pal[i][2] = stbi__get8(s);
         pal[i][1] = stbi__get8(s);
         pal[i][0] = stbi__get8(s);
         if (info.hsz != 12) stbi__get8(s);
         pal[i][3] = 255;
      }
      stbi__skip(s, info.offset - info.extra_read - info.hsz - psize * (info.hsz == 12 ? 3 : 4));
      if (info.bpp == 1) width = (s->img_x + 7) » 3;
      else if (info.bpp == 4) width = (s->img_x + 1) » 1;
      else if (info.bpp == 8) width = s->img_x;
      else { STBI_FREE(out); return stbi__errpuc("bad bpp", "Corrupt BMP"); }
      pad = (-width)&3;
      if (info.bpp == 1) {
         for (j=0; j < (int) s->img_y; ++j) {
            int bit_offset = 7, v = stbi__get8(s);
            for (i=0; i < (int) s->img_x; ++i) {
               int color = (v»bit_offset)&0x1;
               out[z++] = pal[color][0];
               out[z++] = pal[color][1];
               out[z++] = pal[color][2];
               if (target == 4) out[z++] = 255;
               if (i+1 == (int) s->img_x) break;
               if((--bit_offset) < 0) {
                  bit_offset = 7;
                  v = stbi__get8(s);
               }
            }
            stbi__skip(s, pad);
         }
      } else {
         for (j=0; j < (int) s->img_y; ++j) {
            for (i=0; i < (int) s->img_x; i += 2) {
               int v=stbi__get8(s),v2=0;
               if (info.bpp == 4) {
                  v2 = v & 15;
                  v »= 4;
               }
               out[z++] = pal[v][0];
               out[z++] = pal[v][1];
               out[z++] = pal[v][2];
               if (target == 4) out[z++] = 255;
               if (i+1 == (int) s->img_x) break;
               v = (info.bpp == 8) ? stbi__get8(s) : v2;
               out[z++] = pal[v][0];
               out[z++] = pal[v][1];
               out[z++] = pal[v][2];
               if (target == 4) out[z++] = 255;
            }
            stbi__skip(s, pad);
         }
      }
   } else {
      int rshift=0,gshift=0,bshift=0,ashift=0,rcount=0,gcount=0,bcount=0,acount=0;
      int z = 0;
      int easy=0;
      stbi__skip(s, info.offset - info.extra_read - info.hsz);
      if (info.bpp == 24) width = 3 * s->img_x;
      else if (info.bpp == 16) width = 2*s->img_x;
      else /* bpp = 32 and pad = 0 */ width=0;
      pad = (-width) & 3;
      if (info.bpp == 24) {
         easy = 1;
      } else if (info.bpp == 32) {
         if (mb == 0xff && mg == 0xff00 && mr == 0x00ff0000 && ma == 0xff000000)
            easy = 2;
      }
      if (!easy) {
         if (!mr || !mg || !mb) { STBI_FREE(out); return stbi__errpuc("bad masks", "Corrupt BMP"); }
         // right shift amt to put high bit in position #7
         rshift = stbi__high_bit(mr)-7; rcount = stbi__bitcount(mr);
         gshift = stbi__high_bit(mg)-7; gcount = stbi__bitcount(mg);
         bshift = stbi__high_bit(mb)-7; bcount = stbi__bitcount(mb);
         ashift = stbi__high_bit(ma)-7; acount = stbi__bitcount(ma);
         if (rcount > 8 || gcount > 8 || bcount > 8 || acount > 8) { STBI_FREE(out); return
stbi__errpuc("bad masks", "Corrupt BMP"); }
      }
      for (j=0; j < (int) s->img_y; ++j) {
         if (easy) {
            for (i=0; i < (int) s->img_x; ++i) {
               unsigned char a;
               out[z+2] = stbi__get8(s);
               out[z+1] = stbi__get8(s);
               out[z+0] = stbi__get8(s);
               z += 3;
               a = (easy == 2 ? stbi__get8(s) : 255);
               all_a |= a;
               if (target == 4) out[z++] = a;
            }
         } else {
            int bpp = info.bpp;
            for (i=0; i < (int) s->img_x; ++i) {
               stbi__uint32 v = (bpp == 16 ? (stbi__uint32) stbi__get16le(s) : stbi__get32le(s));
               unsigned int a;
               out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mr, rshift, rcount));
               out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mg, gshift, gcount));
               out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mb, bshift, bcount));
               a = (ma ? stbi__shiftsigned(v & ma, ashift, acount) : 255);
               all_a |= a;
               if (target == 4) out[z++] = STBI__BYTECAST(a);
            }
         }
         stbi__skip(s, pad);
      }
   }

   // if alpha channel is all 0s, replace with all 255s
   if (target == 4 && all_a == 0)
      for (i=4*s->img_x*s->img_y-1; i >= 0; i -= 4)
         out[i] = 255;

   if (flip_vertically) {
      stbi_uc t;
      for (j=0; j < (int) s->img_y»1; ++j) {
         stbi_uc *p1 = out +      j     *s->img_x*target;
         stbi_uc *p2 = out + (s->img_y-1-j)*s->img_x*target;
         for (i=0; i < (int) s->img_x*target; ++i) {
            t = p1[i]; p1[i] = p2[i]; p2[i] = t;
         }
      }
   }

   if (req_comp && req_comp != target) {
      out = stbi__convert_format(out, target, req_comp, s->img_x, s->img_y);
      if (out == NULL) return out; // stbi__convert_format frees input on failure
   }

   *x = s->img_x;
   *y = s->img_y;
   if (comp) *comp = s->img_n;
   return out;
}
#endif
```

```
// Targa Truevision - TGA
// by Jonathan Dummer
#ifndef STBI_NO_TGA
// returns STBI_rgb or whatever, 0 on error
static int stbi__tga_get_comp(int bits_per_pixel, int is_grey, int* is_rgb16)
{
   // only RGB or RGBA (incl. 16bit) or grey allowed
   if (is_rgb16) *is_rgb16 = 0;
   switch(bits_per_pixel) {
      case 8:  return STBI_grey;
      case 16: if(is_grey) return STBI_grey_alpha;
               // fallthrough
      case 15: if(is_rgb16) *is_rgb16 = 1;
               return STBI_rgb;
      case 24: // fallthrough
      case 32: return bits_per_pixel/8;
      default: return 0;
   }
}

static int stbi__tga_info(stbi__context *s, int *x, int *y, int *comp)
{
    int tga_w, tga_h, tga_comp, tga_image_type, tga_bits_per_pixel, tga_colormap_bpp;
    int sz, tga_colormap_type;
    stbi__get8(s);                   // discard Offset
    tga_colormap_type = stbi__get8(s); // colormap type
    if( tga_colormap_type > 1 ) {
        stbi__rewind(s);
        return 0;      // only RGB or indexed allowed
    }
    tga_image_type = stbi__get8(s); // image type
    if ( tga_colormap_type == 1 ) { // colormapped (paletted) image
        if (tga_image_type != 1 && tga_image_type != 9) {
            stbi__rewind(s);
            return 0;
        }
        stbi__skip(s,4);       // skip index of first colormap entry and number of entries
        sz = stbi__get8(s);    //   check bits per palette color entry
        if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) {
            stbi__rewind(s);
            return 0;
        }
        stbi__skip(s,4);       // skip image x and y origin
        tga_colormap_bpp = sz;
    } else { // "normal" image w/o colormap - only RGB or grey allowed, +/- RLE
        if ( (tga_image_type != 2) && (tga_image_type != 3) && (tga_image_type != 10) && (tga_image_type != 11) ) {
            stbi__rewind(s);
            return 0; // only RGB or grey allowed, +/- RLE
        }
        stbi__skip(s,9); // skip colormap specification and image x/y origin
        tga_colormap_bpp = 0;
    }
    tga_w = stbi__get16le(s);
    if( tga_w < 1 ) {
        stbi__rewind(s);
        return 0;   // test width
    }
    tga_h = stbi__get16le(s);
    if( tga_h < 1 ) {
        stbi__rewind(s);
        return 0;   // test height
    }
    tga_bits_per_pixel = stbi__get8(s); // bits per pixel
    stbi__get8(s); // ignore alpha bits
    if (tga_colormap_bpp != 0) {
        if((tga_bits_per_pixel != 8) && (tga_bits_per_pixel != 16)) {
            // when using a colormap, tga_bits_per_pixel is the size of the indexes
            // I don't think anything but 8 or 16bit indexes makes sense
            stbi__rewind(s);
            return 0;
        }
        tga_comp = stbi__tga_get_comp(tga_colormap_bpp, 0, NULL);
    } else {
        tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3) || (tga_image_type == 11), NULL);
    }
    if(!tga_comp) {
      stbi__rewind(s);
      return 0;
    }
    if (x) *x = tga_w;
    if (y) *y = tga_h;
    if (comp) *comp = tga_comp;
    return 1;                   // seems to have passed everything
}
```

```

static int stbi__tga_test(stbi__context *s)
{
   int res = 0;
   int sz, tga_color_type;
   stbi__get8(s);      //   discard Offset
   tga_color_type = stbi__get8(s);   //   color type
   if ( tga_color_type > 1 ) goto errorEnd;   //   only RGB or indexed allowed
   sz = stbi__get8(s);   //   image type
   if ( tga_color_type == 1 ) { // colormapped (paletted) image
      if (sz != 1 && sz != 9) goto errorEnd; // colortype 1 demands image type 1 or 9
      stbi__skip(s,4);       // skip index of first colormap entry and number of entries
      sz = stbi__get8(s);    //   check bits per palette color entry
      if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;
      stbi__skip(s,4);       // skip image x and y origin
   } else { // "normal" image w/o colormap
      if ( (sz != 2) && (sz != 3) && (sz != 10) && (sz != 11) ) goto errorEnd; // only RGB or grey allowed, +/- RLE
      stbi__skip(s,9); // skip colormap specification and image x/y origin
   }
   if ( stbi__get16le(s) < 1 ) goto errorEnd;      //   test width
   if ( stbi__get16le(s) < 1 ) goto errorEnd;      //   test height
   sz = stbi__get8(s);   //   bits per pixel
   if ( (tga_color_type == 1) && (sz != 8) && (sz != 16) ) goto errorEnd; // for colormapped images, bpp is size of an index
   if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;

   res = 1; // if we got this far, everything's good and we can return 1 instead of 0

errorEnd:
   stbi__rewind(s);
   return res;
}

// read 16bit value and convert to 24bit RGB
static void stbi__tga_read_rgb16(stbi__context *s, stbi_uc* out)
{
   stbi__uint16 px = (stbi__uint16)stbi__get16le(s);
   stbi__uint16 fiveBitMask = 31;
   // we have 3 channels with 5bits each
   int r = (px » 10) & fiveBitMask;
   int g = (px » 5) & fiveBitMask;
   int b = px & fiveBitMask;
   // Note that this saves the data in RGB(A) order, so it doesn't need to be swapped later
   out[0] = (stbi_uc)((r * 255)/31);
   out[1] = (stbi_uc)((g * 255)/31);
   out[2] = (stbi_uc)((b * 255)/31);

   // some people claim that the most significant bit might be used for alpha
   // (possibly if an alpha-bit is set in the "image descriptor byte")
   // but that only made 16bit test images completely translucent..
   // so let's treat all 15 and 16bit TGAs as RGB with no alpha.
}

static void *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   //   read in the TGA header stuff
   int tga_offset = stbi__get8(s);
   int tga_indexed = stbi__get8(s);
   int tga_image_type = stbi__get8(s);
   int tga_is_RLE = 0;
   int tga_palette_start = stbi__get16le(s);
   int tga_palette_len = stbi__get16le(s);
   int tga_palette_bits = stbi__get8(s);
   int tga_x_origin = stbi__get16le(s);
   int tga_y_origin = stbi__get16le(s);
   int tga_width = stbi__get16le(s);
   int tga_height = stbi__get16le(s);
   int tga_bits_per_pixel = stbi__get8(s);
   int tga_comp, tga_rgb16=0;
   int tga_inverted = stbi__get8(s);
   // int tga_alpha_bits = tga_inverted & 15; // the 4 lowest bits - unused (useless?)
   //   image data
   unsigned char *tga_data;
   unsigned char *tga_palette = NULL;
   int i, j;
   unsigned char raw_data[4] = {0};
   int RLE_count = 0;
   int RLE_repeating = 0;
   int read_next_pixel = 1;
   STBI_NOTUSED(ri);
   STBI_NOTUSED(tga_x_origin); // @TODO
   STBI_NOTUSED(tga_y_origin); // @TODO

   if (tga_height > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
   if (tga_width > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");

   //   do a tiny bit of precessing
   if ( tga_image_type >= 8 )
   {
      tga_image_type -= 8;
      tga_is_RLE = 1;
   }
   tga_inverted = 1 - ((tga_inverted » 5) & 1);

   //   If I'm paletted, then I'll use the number of bits from the palette
   if ( tga_indexed ) tga_comp = stbi__tga_get_comp(tga_palette_bits, 0, &tga_rgb16);
   else tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3), &tga_rgb16);

   if(!tga_comp) // shouldn't really happen, stbi__tga_test() should have ensured basic consistency
      return stbi__errpuc("bad format", "Can't find out TGA pixelformat");

   //   tga info
   *x = tga_width;
   *y = tga_height;
   if (comp) *comp = tga_comp;

   if (!stbi__mad3sizes_valid(tga_width, tga_height, tga_comp, 0))
      return stbi__errpuc("too large", "Corrupt TGA");

   tga_data = (unsigned char*)stbi__malloc_mad3(tga_width, tga_height, tga_comp, 0);
   if (!tga_data) return stbi__errpuc("outofmem", "Out of memory");

   // skip to the data's starting position (offset usually = 0)
   stbi__skip(s, tga_offset );

   if ( !tga_indexed && !tga_is_RLE && !tga_rgb16 ) {
      for (i=0; i < tga_height; ++i) {
         int row = tga_inverted ? tga_height -i - 1 : i;
         stbi_uc *tga_row = tga_data + row*tga_width*tga_comp;
         stbi__getn(s, tga_row, tga_width * tga_comp);
      }
   } else  {
      //   do I need to load a palette?
      if ( tga_indexed)
      {
         if (tga_palette_len == 0) {  /* you have to have at least one entry! */
            STBI_FREE(tga_data);
            return stbi__errpuc("bad palette", "Corrupt TGA");
         }

         //   any data to skip? (offset usually = 0)
         stbi__skip(s, tga_palette_start );
         //   load the palette
         tga_palette = (unsigned char*)stbi__malloc_mad2(tga_palette_len, tga_comp, 0);
         if (!tga_palette) {
            STBI_FREE(tga_data);
            return stbi__errpuc("outofmem", "Out of memory");
         }
         if (tga_rgb16) {
            stbi_uc *pal_entry = tga_palette;
            STBI_ASSERT(tga_comp == STBI_rgb);
            for (i=0; i < tga_palette_len; ++i) {
               stbi__tga_read_rgb16(s, pal_entry);
               pal_entry += tga_comp;
            }
         } else if (!stbi__getn(s, tga_palette, tga_palette_len * tga_comp)) {
               STBI_FREE(tga_data);
               STBI_FREE(tga_palette);
               return stbi__errpuc("bad palette", "Corrupt TGA");
         }
      }
      //   load the data
      for (i=0; i < tga_width * tga_height; ++i)
      {
         //   if I'm in RLE mode, do I need to get a RLE stbi__pngchunk?
         if ( tga_is_RLE )
         {
            if ( RLE_count == 0 )
            {
               //   yep, get the next byte as a RLE command
               int RLE_cmd = stbi__get8(s);
               RLE_count = 1 + (RLE_cmd & 127);
               RLE_repeating = RLE_cmd » 7;
               read_next_pixel = 1;
            } else if ( !RLE_repeating )
            {
               read_next_pixel = 1;
            }
         } else
         {
            read_next_pixel = 1;
         }
         //   OK, if I need to read a pixel, do it now
         if ( read_next_pixel )
         {
            //   load however much data we did have
            if ( tga_indexed )
            {
               // read in index, then perform the lookup
               int pal_idx = (tga_bits_per_pixel == 8) ? stbi__get8(s) : stbi__get16le(s);
               if ( pal_idx >= tga_palette_len ) {
                  // invalid index
                  pal_idx = 0;
               }
               pal_idx *= tga_comp;
               for (j = 0; j < tga_comp; ++j) {
                  raw_data[j] = tga_palette[pal_idx+j];
               }
            } else if(tga_rgb16) {
               STBI_ASSERT(tga_comp == STBI_rgb);
               stbi__tga_read_rgb16(s, raw_data);
            } else {
               //   read in the data raw
               for (j = 0; j < tga_comp; ++j) {
                  raw_data[j] = stbi__get8(s);
               }
            }
            //   clear the reading flag for the next pixel
            read_next_pixel = 0;
         } // end of reading a pixel

         // copy data
         for (j = 0; j < tga_comp; ++j)
           tga_data[i*tga_comp+j] = raw_data[j];

         //   in case we're in RLE mode, keep counting down
         --RLE_count;
      }
      //   do I need to invert the image?
      if ( tga_inverted )
      {
         for (j = 0; j*2 < tga_height; ++j)
         {
            int index1 = j * tga_width * tga_comp;
            int index2 = (tga_height - 1 - j) * tga_width * tga_comp;
            for (i = tga_width * tga_comp; i > 0; --i)
            {
               unsigned char temp = tga_data[index1];
               tga_data[index1] = tga_data[index2];
               tga_data[index2] = temp;
               ++index1;
               ++index2;
            }
         }
      }
      //   clear my palette, if I had one
      if ( tga_palette != NULL )
      {
         STBI_FREE( tga_palette );
      }
   }

   // swap RGB - if the source data was RGB16, it already is in the right order
   if (tga_comp >= 3 && !tga_rgb16)
   {
      unsigned char* tga_pixel = tga_data;
      for (i=0; i < tga_width * tga_height; ++i)
      {
         unsigned char temp = tga_pixel[0];
         tga_pixel[0] = tga_pixel[2];
         tga_pixel[2] = temp;
         tga_pixel += tga_comp;
      }
   }

   // convert to target component count
   if (req_comp && req_comp != tga_comp)
      tga_data = stbi__convert_format(tga_data, tga_comp, req_comp, tga_width, tga_height);

   //   the things I do to get rid of an error message, and yet keep
   //   Microsoft's C compilers happy... [8^(
   tga_palette_start = tga_palette_len = tga_palette_bits =
         tga_x_origin = tga_y_origin = 0;
   STBI_NOTUSED(tga_palette_start);
   //   OK, done
   return tga_data;
}
```

```
#endif

// *************************************************************************************************
// Photoshop PSD loader -- PD by Thatcher Ulrich, integration by Nicolas Schulz, tweaked by STB

#ifndef STBI_NO_PSD
static int stbi__psd_test(stbi__context *s)
{
   int r = (stbi__get32be(s) == 0x38425053);
   stbi__rewind(s);
   return r;
}

static int stbi__psd_decode_rle(stbi__context *s, stbi_uc *p, int pixelCount)
{
   int count, nleft, len;

   count = 0;
   while ((nleft = pixelCount - count) > 0) {
      len = stbi__get8(s);
      if (len == 128) {
         // No-op.
      } else if (len < 128) {
         // Copy next len+1 bytes literally.
         len++;
         if (len > nleft) return 0; // corrupt data
         count += len;
         while (len) {
            *p = stbi__get8(s);
            p += 4;
            len--;
         }
      } else if (len > 128) {
         stbi_uc   val;
         // Next -len+1 bytes in the dest are replicated from next source byte.
         // (Interpret len as a negative 8-bit int.)
         len = 257 - len;
         if (len > nleft) return 0; // corrupt data
         val = stbi__get8(s);
         count += len;
         while (len) {
            *p = val;
            p += 4;
            len--;
         }
      }
   }

   return 1;
}

static void *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri, int bpc)
{
   int pixelCount;
   int channelCount, compression;
   int channel, i;
   int bitdepth;
   int w,h;
   stbi_uc *out;
   STBI_NOTUSED(ri);

   // Check identifier
   if (stbi__get32be(s) != 0x38425053)   // "8BPS"
      return stbi__errpuc("not PSD", "Corrupt PSD image");

   // Check file type version.
   if (stbi__get16be(s) != 1)
      return stbi__errpuc("wrong version", "Unsupported version of PSD image");

   // Skip 6 reserved bytes.
   stbi__skip(s, 6 );

   // Read the number of channels (R, G, B, A, etc).
   channelCount = stbi__get16be(s);
   if (channelCount < 0 || channelCount > 16)
      return stbi__errpuc("wrong channel count", "Unsupported number of channels in PSD image");

   // Read the rows and columns of the image.
   h = stbi__get32be(s);
   w = stbi__get32be(s);

   if (h > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
   if (w > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");

   // Make sure the depth is 8 bits.
   bitdepth = stbi__get16be(s);
   if (bitdepth != 8 && bitdepth != 16)
      return stbi__errpuc("unsupported bit depth", "PSD bit depth is not 8 or 16 bit");

   // Make sure the color mode is RGB.
   // Valid options are:
   //   0: Bitmap
   //   1: Grayscale
   //   2: Indexed color
   //   3: RGB color
   //   4: CMYK color
   //   7: Multichannel
   //   8: Duotone
   //   9: Lab color
   if (stbi__get16be(s) != 3)
      return stbi__errpuc("wrong color format", "PSD is not in RGB color format");

   // Skip the Mode Data.  (It’s the palette for indexed color; other info for other modes.)
   stbi__skip(s,stbi__get32be(s) );

   // Skip the image resources.  (resolution, pen tool paths, etc)
   stbi__skip(s, stbi__get32be(s) );

   // Skip the reserved data.
   stbi__skip(s, stbi__get32be(s) );

   // Find out if the data is compressed.
   // Known values:
   //   0: no compression
   //   1: RLE compressed
   compression = stbi__get16be(s);
   if (compression > 1)
      return stbi__errpuc("bad compression", "PSD has an unknown compression format");

   // Check size
   if (!stbi__mad3sizes_valid(4, w, h, 0))
      return stbi__errpuc("too large", "Corrupt PSD");

   // Create the destination image.

   if (!compression && bitdepth == 16 && bpc == 16) {
      out = (stbi_uc *) stbi__malloc_mad3(8, w, h, 0);
      ri->bits_per_channel = 16;
   } else
      out = (stbi_uc *) stbi__malloc(4 * w*h);

   if (!out) return stbi__errpuc("outofmem", "Out of memory");
   pixelCount = w*h;

   // Initialize the data to zero.
   //memset( out, 0, pixelCount * 4 );

   // Finally, the image data.
   if (compression) {
      // RLE as used by .PSD and .TIFF
      // Loop until you get the number of unpacked bytes you are expecting:
      //     Read the next source byte into n.
      //     If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
      //     Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
      //     Else if n is 128, noop.
      // Endloop

      // The RLE-compressed data is preceded by a 2-byte data count for each row in the data,
      // which we’re going to just skip.
      stbi__skip(s, h * channelCount * 2 );

      // Read the RLE data by channel.
      for (channel = 0; channel < 4; channel++) {
         stbi_uc *p;

         p = out+channel;
         if (channel >= channelCount) {
            // Fill this channel with default data.
            for (i = 0; i < pixelCount; i++, p += 4)
               *p = (channel == 3 ? 255 : 0);
         } else {
            // Read the RLE data.
            if (!stbi__psd_decode_rle(s, p, pixelCount)) {
               STBI_FREE(out);
               return stbi__errpuc("corrupt", "bad RLE data");
            }
         }
      }

   } else {
      // We’re at the raw image data.  It’s each channel in order (Red, Green, Blue, Alpha, ...)
      // where each channel consists of an 8-bit (or 16-bit) value for each pixel in the image.

      // Read the data by channel.
      for (channel = 0; channel < 4; channel++) {
         if (channel >= channelCount) {
            // Fill this channel with default data.
            if (bitdepth == 16 && bpc == 16) {
               stbi__uint16 *q = ((stbi__uint16 *) out) + channel;
               stbi__uint16 val = channel == 3 ? 65535 : 0;
               for (i = 0; i < pixelCount; i++, q += 4)
                  *q = val;
            } else {
               stbi_uc *p = out+channel;
               stbi_uc val = channel == 3 ? 255 : 0;
               for (i = 0; i < pixelCount; i++, p += 4)
                  *p = val;
            }
         } else {
            if (ri->bits_per_channel == 16) {    // output bpc
               stbi__uint16 *q = ((stbi__uint16 *) out) + channel;
               for (i = 0; i < pixelCount; i++, q += 4)
                  *q = (stbi__uint16) stbi__get16be(s);
            } else {
               stbi_uc *p = out+channel;
               if (bitdepth == 16) {  // input bpc
                  for (i = 0; i < pixelCount; i++, p += 4)
                     *p = (stbi_uc) (stbi__get16be(s) » 8);
               } else {
                  for (i = 0; i < pixelCount; i++, p += 4)
                     *p = stbi__get8(s);
               }
            }
         }
      }
   }

   // remove weird white matte from PSD
   if (channelCount >= 4) {
      if (ri->bits_per_channel == 16) {
         for (i=0; i < w*h; ++i) {
            stbi__uint16 *pixel = (stbi__uint16 *) out + 4*i;
            if (pixel[3] != 0 && pixel[3] != 65535) {
               float a = pixel[3] / 65535.0f;
               float ra = 1.0f / a;
               float inv_a = 65535.0f * (1 - ra);
               pixel[0] = (stbi__uint16) (pixel[0]*ra + inv_a);
               pixel[1] = (stbi__uint16) (pixel[1]*ra + inv_a);
               pixel[2] = (stbi__uint16) (pixel[2]*ra + inv_a);
            }
         }
      } else {
         for (i=0; i < w*h; ++i) {
            unsigned char *pixel = out + 4*i;
            if (pixel[3] != 0 && pixel[3] != 255) {
               float a = pixel[3] / 255.0f;
               float ra = 1.0f / a;
               float inv_a = 255.0f * (1 - ra);
               pixel[0] = (unsigned char) (pixel[0]*ra + inv_a);
               pixel[1] = (unsigned char) (pixel[1]*ra + inv_a);
               pixel[2] = (unsigned char) (pixel[2]*ra + inv_a);
            }
         }
      }
   }

   // convert to desired output format
   if (req_comp && req_comp != 4) {
      if (ri->bits_per_channel == 16)
         out = (stbi_uc *) stbi__convert_format16((stbi__uint16 *) out, 4, req_comp, w, h);
      else
         out = stbi__convert_format(out, 4, req_comp, w, h);
      if (out == NULL) return out; // stbi__convert_format frees input on failure
   }

   if (comp) *comp = 4;
   *y = h;
   *x = w;

   return out;
}
#endif

// *************************************************************************************************
// Softimage PIC loader
// by Tom Seddon
//
// See http://softimage.wiki.softimage.com/index.php/INFO:_PIC_file_format
// See http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/softimagepic/

```

```
#ifndef STBI_NO_PIC
static int stbi__pic_is4(stbi__context *s,const char *str)
{
   int i;
   for (i=0; i<4; ++i)
      if (stbi__get8(s) != (stbi_uc)str[i])
         return 0;

   return 1;
}

static int stbi__pic_test_core(stbi__context *s)
{
   int i;

   if (!stbi__pic_is4(s,"\x53\x80\xF6\x34"))
      return 0;

   for(i=0;i<84;++i)
      stbi__get8(s);

   if (!stbi__pic_is4(s,"PICT"))
      return 0;

   return 1;
}

typedef struct
{
   stbi_uc size,type,channel;
} stbi__pic_packet;

static stbi_uc *stbi__readval(stbi__context *s, int channel, stbi_uc *dest)
{
   int mask=0x80, i;

   for (i=0; i<4; ++i, mask»=1) {
      if (channel & mask) {
         if (stbi__at_eof(s)) return stbi__errpuc("bad file","PIC file too short");
         dest[i]=stbi__get8(s);
      }
   }

   return dest;
}

static void stbi__copyval(int channel,stbi_uc *dest,const stbi_uc *src)
{
   int mask=0x80,i;

   for (i=0;i<4; ++i, mask»=1)
      if (channel&mask)
         dest[i]=src[i];
}

static stbi_uc *stbi__pic_load_core(stbi__context *s,int width,int height,int *comp, stbi_uc *result)
{
   int act_comp=0,num_packets=0,y,chained;
   stbi__pic_packet packets[10];

   // this will (should...) cater for even some bizarre stuff like having data
    // for the same channel in multiple packets.
   do {
      stbi__pic_packet *packet;

      if (num_packets==sizeof(packets)/sizeof(packets[0]))
         return stbi__errpuc("bad format","too many packets");

      packet = &packets[num_packets++];

      chained = stbi__get8(s);
      packet->size    = stbi__get8(s);
      packet->type    = stbi__get8(s);
      packet->channel = stbi__get8(s);

      act_comp |= packet->channel;

      if (stbi__at_eof(s))          return stbi__errpuc("bad file","file too short (reading packets)");
      if (packet->size != 8)  return stbi__errpuc("bad format","packet isn't 8bpp");
   } while (chained);

   *comp = (act_comp & 0x10 ? 4 : 3); // has alpha channel?

   for(y=0; y<height; ++y) {
      int packet_idx;

      for(packet_idx=0; packet_idx < num_packets; ++packet_idx) {
         stbi__pic_packet *packet = &packets[packet_idx];
         stbi_uc *dest = result+y*width*4;

         switch (packet->type) {
            default:
               return stbi__errpuc("bad format","packet has bad compression type");

            case 0: {//uncompressed
               int x;

               for(x=0;x<width;++x, dest+=4)
                  if (!stbi__readval(s,packet->channel,dest))
                     return 0;
               break;
            }

            case 1://Pure RLE
               {
                  int left=width, i;

                  while (left>0) {
                     stbi_uc count,value[4];

                     count=stbi__get8(s);
                     if (stbi__at_eof(s))   return stbi__errpuc("bad file","file too short (pure read count)");

                     if (count > left)
                        count = (stbi_uc) left;

                     if (!stbi__readval(s,packet->channel,value))  return 0;

                     for(i=0; i<count; ++i,dest+=4)
                        stbi__copyval(packet->channel,dest,value);
                     left -= count;
                  }
               }
               break;

            case 2: {//Mixed RLE
               int left=width;
               while (left>0) {
                  int count = stbi__get8(s), i;
                  if (stbi__at_eof(s))  return stbi__errpuc("bad file","file too short (mixed read count)");

                  if (count >= 128) { // Repeated
                     stbi_uc value[4];

                     if (count==128)
                        count = stbi__get16be(s);
                     else
                        count -= 127;
                     if (count > left)
                        return stbi__errpuc("bad file","scanline overrun");

                     if (!stbi__readval(s,packet->channel,value))
                        return 0;

                     for(i=0;i<count;++i, dest += 4)
                        stbi__copyval(packet->channel,dest,value);
                  } else { // Raw
                     ++count;
                     if (count>left) return stbi__errpuc("bad file","scanline overrun");

                     for(i=0;i<count;++i, dest+=4)
                        if (!stbi__readval(s,packet->channel,dest))
                           return 0;
                  }
                  left-=count;
               }
               break;
            }
         }
      }
   }

   return result;
}

static void *stbi__pic_load(stbi__context *s,int *px,int *py,int *comp,int req_comp, stbi__result_info *ri)
{
   stbi_uc *result;
   int i, x,y, internal_comp;
   STBI_NOTUSED(ri);

   if (!comp) comp = &internal_comp;

   for (i=0; i<92; ++i)
      stbi__get8(s);

   x = stbi__get16be(s);
   y = stbi__get16be(s);

   if (y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
   if (x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");

   if (stbi__at_eof(s))  return stbi__errpuc("bad file","file too short (pic header)");
   if (!stbi__mad3sizes_valid(x, y, 4, 0)) return stbi__errpuc("too large", "PIC image too large to decode");

   stbi__get32be(s); //skip `ratio’
   stbi__get16be(s); //skip `fields’
   stbi__get16be(s); //skip `pad’

   // intermediate buffer is RGBA
   result = (stbi_uc *) stbi__malloc_mad3(x, y, 4, 0);
   if (!result) return stbi__errpuc("outofmem", "Out of memory");
   memset(result, 0xff, x*y*4);

   if (!stbi__pic_load_core(s,x,y,comp, result)) {
      STBI_FREE(result);
      result=0;
   }
   *px = x;
   *py = y;
   if (req_comp == 0) req_comp = *comp;
   result=stbi__convert_format(result,4,req_comp,x,y);

   return result;
}

static int stbi__pic_test(stbi__context *s)
{
   int r = stbi__pic_test_core(s);
   stbi__rewind(s);
   return r;
}
#endif

// *************************************************************************************************
// GIF loader -- public domain by Jean-Marc Lienher -- simplified/shrunk by stb

#ifndef STBI_NO_GIF
typedef struct
{
   stbi__int16 prefix;
   stbi_uc first;
   stbi_uc suffix;
} stbi__gif_lzw;

typedef struct
{
   int w,h;
   stbi_uc *out;                 // output buffer (always 4 components)
   stbi_uc *background;          // The current "background" as far as a gif is concerned
   stbi_uc *history;
   int flags, bgindex, ratio, transparent, eflags;
   stbi_uc  pal[256][4];
   stbi_uc lpal[256][4];
   stbi__gif_lzw codes[8192];
   stbi_uc *color_table;
   int parse, step;
   int lflags;
   int start_x, start_y;
   int max_x, max_y;
   int cur_x, cur_y;
   int line_size;
   int delay;
} stbi__gif;

static int stbi__gif_test_raw(stbi__context *s)
{
   int sz;
   if (stbi__get8(s) != ’G’ || stbi__get8(s) != ’I’ || stbi__get8(s) != ’F’ || stbi__get8(s) != ’8’) return 0;
   sz = stbi__get8(s);
   if (sz != ’9’ && sz != ’7’) return 0;
   if (stbi__get8(s) != ’a’) return 0;
   return 1;
}

```

```
static int stbi__gif_test(stbi__context *s)
{
   int r = stbi__gif_test_raw(s);
   stbi__rewind(s);
   return r;
}

static void stbi__gif_parse_colortable(stbi__context *s, stbi_uc pal[256][4], int num_entries, int transp)
{
   int i;
   for (i=0; i < num_entries; ++i) {
      pal[i][2] = stbi__get8(s);
      pal[i][1] = stbi__get8(s);
      pal[i][0] = stbi__get8(s);
      pal[i][3] = transp == i ? 0 : 255;
   }
}

static int stbi__gif_header(stbi__context *s, stbi__gif *g, int *comp, int is_info)
{
   stbi_uc version;
   if (stbi__get8(s) != ’G’ || stbi__get8(s) != ’I’ || stbi__get8(s) != ’F’ || stbi__get8(s) != ’8’)
      return stbi__err("not GIF", "Corrupt GIF");

   version = stbi__get8(s);
   if (version != ’7’ && version != ’9’)    return stbi__err("not GIF", "Corrupt GIF");
   if (stbi__get8(s) != ’a’)                return stbi__err("not GIF", "Corrupt GIF");

   stbi__g_failure_reason = "";
   g->w = stbi__get16le(s);
   g->h = stbi__get16le(s);
   g->flags = stbi__get8(s);
   g->bgindex = stbi__get8(s);
   g->ratio = stbi__get8(s);
   g->transparent = -1;

   if (g->w > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
   if (g->h > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");

   if (comp != 0) *comp = 4;  // can’t actually tell whether it’s 3 or 4 until we parse the comments

   if (is_info) return 1;

   if (g->flags & 0x80)
      stbi__gif_parse_colortable(s,g->pal, 2 « (g->flags & 7), -1);

   return 1;
}

static int stbi__gif_info_raw(stbi__context *s, int *x, int *y, int *comp)
{
   stbi__gif* g = (stbi__gif*) stbi__malloc(sizeof(stbi__gif));
   if (!g) return stbi__err("outofmem", "Out of memory");
   if (!stbi__gif_header(s, g, comp, 1)) {
      STBI_FREE(g);
      stbi__rewind( s );
      return 0;
   }
   if (x) *x = g->w;
   if (y) *y = g->h;
   STBI_FREE(g);
   return 1;
}

static void stbi__out_gif_code(stbi__gif *g, stbi__uint16 code)
{
   stbi_uc *p, *c;
   int idx;

   // recurse to decode the prefixes, since the linked-list is backwards,
   // and working backwards through an interleaved image would be nasty
   if (g->codes[code].prefix >= 0)
      stbi__out_gif_code(g, g->codes[code].prefix);

   if (g->cur_y >= g->max_y) return;

   idx = g->cur_x + g->cur_y;
   p = &g->out[idx];
   g->history[idx / 4] = 1;

   c = &g->color_table[g->codes[code].suffix * 4];
   if (c[3] > 128) { // don’t render transparent pixels;
      p[0] = c[2];
      p[1] = c[1];
      p[2] = c[0];
      p[3] = c[3];
   }
   g->cur_x += 4;

   if (g->cur_x >= g->max_x) {
      g->cur_x = g->start_x;
      g->cur_y += g->step;

      while (g->cur_y >= g->max_y && g->parse > 0) {
         g->step = (1 « g->parse) * g->line_size;
         g->cur_y = g->start_y + (g->step » 1);
         --g->parse;
      }
   }
}

static stbi_uc *stbi__process_gif_raster(stbi__context *s, stbi__gif *g)
{
   stbi_uc lzw_cs;
   stbi__int32 len, init_code;
   stbi__uint32 first;
   stbi__int32 codesize, codemask, avail, oldcode, bits, valid_bits, clear;
   stbi__gif_lzw *p;

   lzw_cs = stbi__get8(s);
   if (lzw_cs > 12) return NULL;
   clear = 1 « lzw_cs;
   first = 1;
   codesize = lzw_cs + 1;
   codemask = (1 « codesize) - 1;
   bits = 0;
   valid_bits = 0;
   for (init_code = 0; init_code < clear; init_code++) {
      g->codes[init_code].prefix = -1;
      g->codes[init_code].first = (stbi_uc) init_code;
      g->codes[init_code].suffix = (stbi_uc) init_code;
   }

   // support no starting clear code
   avail = clear+2;
   oldcode = -1;

   len = 0;
   for(;;) {
      if (valid_bits < codesize) {
         if (len == 0) {
            len = stbi__get8(s); // start new block
            if (len == 0)
               return g->out;
         }
         --len;
         bits |= (stbi__int32) stbi__get8(s) « valid_bits;
         valid_bits += 8;
      } else {
         stbi__int32 code = bits & codemask;
         bits »= codesize;
         valid_bits -= codesize;
         // @OPTIMIZE: is there some way we can accelerate the non-clear path?
         if (code == clear) {  // clear code
            codesize = lzw_cs + 1;
            codemask = (1 « codesize) - 1;
            avail = clear + 2;
            oldcode = -1;
            first = 0;
         } else if (code == clear + 1) { // end of stream code
            stbi__skip(s, len);
            while ((len = stbi__get8(s)) > 0)
               stbi__skip(s,len);
            return g->out;
         } else if (code <= avail) {
            if (first) {
               return stbi__errpuc("no clear code", "Corrupt GIF");
            }

            if (oldcode >= 0) {
               p = &g->codes[avail++];
               if (avail > 8192) {
                  return stbi__errpuc("too many codes", "Corrupt GIF");
               }

               p->prefix = (stbi__int16) oldcode;
               p->first = g->codes[oldcode].first;
               p->suffix = (code == avail) ? p->first : g->codes[code].first;
            } else if (code == avail)
               return stbi__errpuc("illegal code in raster", "Corrupt GIF");

            stbi__out_gif_code(g, (stbi__uint16) code);

            if ((avail & codemask) == 0 && avail <= 0x0FFF) {
               codesize++;
               codemask = (1 « codesize) - 1;
            }

            oldcode = code;
         } else {
            return stbi__errpuc("illegal code in raster", "Corrupt GIF");
         }
      }
   }
}

// this function is designed to support animated gifs, although stb_image doesn’t support it
// two back is the image from two frames ago, used for a very specific disposal format
static stbi_uc *stbi__gif_load_next(stbi__context *s, stbi__gif *g, int *comp, int req_comp, stbi_uc *two_back)
{
   int dispose;
   int first_frame;
   int pi;
   int pcount;
   STBI_NOTUSED(req_comp);

   // on first frame, any non-written pixels get the background colour (non-transparent)
   first_frame = 0;
   if (g->out == 0) {
      if (!stbi__gif_header(s, g, comp,0)) return 0; // stbi__g_failure_reason set by stbi__gif_header
      if (!stbi__mad3sizes_valid(4, g->w, g->h, 0))
         return stbi__errpuc("too large", "GIF image is too large");
      pcount = g->w * g->h;
      g->out = (stbi_uc *) stbi__malloc(4 * pcount);
      g->background = (stbi_uc *) stbi__malloc(4 * pcount);
      g->history = (stbi_uc *) stbi__malloc(pcount);
      if (!g->out || !g->background || !g->history)
         return stbi__errpuc("outofmem", "Out of memory");

      // image is treated as "transparent" at the start - ie, nothing overwrites the current background;
      // background colour is only used for pixels that are not rendered first frame, after that "background"
      // color refers to the color that was there the previous frame.
      memset(g->out, 0x00, 4 * pcount);
      memset(g->background, 0x00, 4 * pcount); // state of the background (starts transparent)
      memset(g->history, 0x00, pcount);        // pixels that were affected previous frame
      first_frame = 1;
   } else {
      // second frame - how do we dispose of the previous one?
      dispose = (g->eflags & 0x1C) » 2;
      pcount = g->w * g->h;

      if ((dispose == 3) && (two_back == 0)) {
         dispose = 2; // if I don’t have an image to revert back to, default to the old background
      }

      if (dispose == 3) { // use previous graphic
         for (pi = 0; pi < pcount; ++pi) {
            if (g->history[pi]) {
               memcpy( &g->out[pi * 4], &two_back[pi * 4], 4 );
            }
         }
      } else if (dispose == 2) {
         // restore what was changed last frame to background before that frame;
         for (pi = 0; pi < pcount; ++pi) {
            if (g->history[pi]) {
               memcpy( &g->out[pi * 4], &g->background[pi * 4], 4 );
            }
         }
      } else {
         // This is a non-disposal case eithe way, so just
         // leave the pixels as is, and they will become the new background
         // 1: do not dispose
         // 0:  not specified.
      }

      // background is what out is after the undoing of the previou frame;
      memcpy( g->background, g->out, 4 * g->w * g->h );
   }

   // clear my history;
   memset( g->history, 0x00, g->w * g->h );        // pixels that were affected previous frame

   for (;;) {
      int tag = stbi__get8(s);
      switch (tag) {
         case 0x2C: /* Image Descriptor */
         {
            stbi__int32 x, y, w, h;
            stbi_uc *o;

            x = stbi__get16le(s);
            y = stbi__get16le(s);
            w = stbi__get16le(s);
            h = stbi__get16le(s);
            if (((x + w) > (g->w)) || ((y + h) > (g->h)))
               return stbi__errpuc("bad Image Descriptor", "Corrupt GIF");

            g->line_size = g->w * 4;
            g->start_x = x * 4;
            g->start_y = y * g->line_size;
            g->max_x   = g->start_x + w * 4;
            g->max_y   = g->start_y + h * g->line_size;
            g->cur_x   = g->start_x;
            g->cur_y   = g->start_y;

            // if the width of the specified rectangle is 0, that means
            // we may not see *any* pixels or the image is malformed;
            // to make sure this is caught, move the current y down to
            // max_y (which is what out_gif_code checks).
            if (w == 0)
               g->cur_y = g->max_y;

            g->lflags = stbi__get8(s);

            if (g->lflags & 0x40) {
               g->step = 8 * g->line_size; // first interlaced spacing
               g->parse = 3;
            } else {
               g->step = g->line_size;
               g->parse = 0;
            }

            if (g->lflags & 0x80) {
               stbi__gif_parse_colortable(s,g->lpal, 2 « (g->lflags & 7), g->eflags & 0x01 ? g->transparent : -1);
               g->color_table = (stbi_uc *) g->lpal;
            } else if (g->flags & 0x80) {
               g->color_table = (stbi_uc *) g->pal;
            } else
               return stbi__errpuc("missing color table", "Corrupt GIF");

            o = stbi__process_gif_raster(s, g);
            if (!o) return NULL;

            // if this was the first frame,
            pcount = g->w * g->h;
            if (first_frame && (g->bgindex > 0)) {
               // if first frame, any pixel not drawn to gets the background color
               for (pi = 0; pi < pcount; ++pi) {
                  if (g->history[pi] == 0) {
                     g->pal[g->bgindex][3] = 255; // just in case it was made transparent, undo that; It will be reset next frame if need be;
                     memcpy( &g->out[pi * 4], &g->pal[g->bgindex], 4 );
                  }
               }
            }

            return o;
         }

         case 0x21: // Comment Extension.
         {
            int len;
            int ext = stbi__get8(s);
            if (ext == 0xF9) { // Graphic Control Extension.
               len = stbi__get8(s);
               if (len == 4) {
                  g->eflags = stbi__get8(s);
                  g->delay = 10 * stbi__get16le(s); // delay - 1/100th of a second, saving as 1/1000ths.

                  // unset old transparent
                  if (g->transparent >= 0) {
                     g->pal[g->transparent][3] = 255;
                  }
                  if (g->eflags & 0x01) {
                     g->transparent = stbi__get8(s);
                     if (g->transparent >= 0) {
                        g->pal[g->transparent][3] = 0;
                     }
                  } else {
                     // don't need transparent
                     stbi__skip(s, 1);
                     g->transparent = -1;
                  }
               } else {
                  stbi__skip(s, len);
                  break;
               }
            }
            while ((len = stbi__get8(s)) != 0) {
               stbi__skip(s, len);
            }
            break;
         }

         case 0x3B: // gif stream termination code
            return (stbi_uc *) s; // using '1' causes warning on some compilers

         default:
            return stbi__errpuc("unknown code", "Corrupt GIF");
      }
   }
}

static void *stbi__load_gif_main_outofmem(stbi__gif *g, stbi_uc *out, int **delays)
{
   STBI_FREE(g->out);
   STBI_FREE(g->history);
   STBI_FREE(g->background);

   if (out) STBI_FREE(out);
   if (delays && *delays) STBI_FREE(*delays);
   return stbi__errpuc("outofmem", "Out of memory");
}

static void *stbi__load_gif_main(stbi__context *s, int **delays, int *x, int *y, int *z, int *comp, int req_comp)
{
   if (stbi__gif_test(s)) {
      int layers = 0;
      stbi_uc *u = 0;
      stbi_uc *out = 0;
      stbi_uc *two_back = 0;
      stbi__gif g;
      int stride;
      int out_size = 0;
      int delays_size = 0;

      STBI_NOTUSED(out_size);
      STBI_NOTUSED(delays_size);

      memset(&g, 0, sizeof(g));
      if (delays) {
         *delays = 0;
      }

      do {
         u = stbi__gif_load_next(s, &g, comp, req_comp, two_back);
         if (u == (stbi_uc *) s) u = 0;  // end of animated gif marker

         if (u) {
            *x = g.w;
            *y = g.h;
            ++layers;
            stride = g.w * g.h * 4;

            if (out) {
               void *tmp = (stbi_uc*) STBI_REALLOC_SIZED( out, out_size, layers * stride );
               if (!tmp)
                  return stbi__load_gif_main_outofmem(&g, out, delays);
               else {
                   out = (stbi_uc*) tmp;
                   out_size = layers * stride;
               }

               if (delays) {
                  int *new_delays = (int*) STBI_REALLOC_SIZED( *delays, delays_size, sizeof(int) * layers );
                  if (!new_delays)
                     return stbi__load_gif_main_outofmem(&g, out, delays);
                  *delays = new_delays;
                  delays_size = layers * sizeof(int);
               }
            } else {
               out = (stbi_uc*)stbi__malloc( layers * stride );
               if (!out)
                  return stbi__load_gif_main_outofmem(&g, out, delays);
               out_size = layers * stride;
               if (delays) {
                  *delays = (int*) stbi__malloc( layers * sizeof(int) );
                  if (!*delays)
                     return stbi__load_gif_main_outofmem(&g, out, delays);
                  delays_size = layers * sizeof(int);
               }
            }
            memcpy( out + ((layers - 1) * stride), u, stride );
            if (layers >= 2) {
               two_back = out - 2 * stride;
            }

            if (delays) {
               (*delays)[layers - 1U] = g.delay;
            }
         }
      } while (u != 0);

      // free temp buffer;
      STBI_FREE(g.out);
      STBI_FREE(g.history);
      STBI_FREE(g.background);

      // do the final conversion after loading everything;
      if (req_comp && req_comp != 4)
         out = stbi__convert_format(out, 4, req_comp, layers * g.w, g.h);

      *z = layers;
      return out;
   } else {
      return stbi__errpuc("not GIF", "Image was not as a gif type.");
   }
}

static void *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   stbi_uc *u = 0;
   stbi__gif g;
   memset(&g, 0, sizeof(g));
   STBI_NOTUSED(ri);

   u = stbi__gif_load_next(s, &g, comp, req_comp, 0);
   if (u == (stbi_uc *) s) u = 0;  // end of animated gif marker
   if (u) {
      *x = g.w;
      *y = g.h;

      // moved conversion to after successful load so that the same
      // can be done for multiple frames.
      if (req_comp && req_comp != 4)
         u = stbi__convert_format(u, 4, req_comp, g.w, g.h);
   } else if (g.out) {
      // if there was an error and we allocated an image buffer, free it!
      STBI_FREE(g.out);
   }

   // free buffers needed for multiple frame loading;
   STBI_FREE(g.history);
   STBI_FREE(g.background);

   return u;
}

static int stbi__gif_info(stbi__context *s, int *x, int *y, int *comp)
{
   return stbi__gif_info_raw(s,x,y,comp);
}
#endif

// *************************************************************************************************
// Radiance RGBE HDR loader
// originally by Nicolas Schulz
#ifndef STBI_NO_HDR
static int stbi__hdr_test_core(stbi__context *s, const char *signature)
{
   int i;
   for (i=0; signature[i]; ++i)
      if (stbi__get8(s) != signature[i])
          return 0;
   stbi__rewind(s);
   return 1;
}

static int stbi__hdr_test(stbi__context* s)
{
   int r = stbi__hdr_test_core(s, "#?RADIANCE\n");
   stbi__rewind(s);
   if(!r) {
       r = stbi__hdr_test_core(s, "#?RGBE\n");
       stbi__rewind(s);
   }
   return r;
}

#define STBI__HDR_BUFLEN  1024
static char *stbi__hdr_gettoken(stbi__context *z, char *buffer)
{
   int len=0;
   char c = '\0';

   c = (char) stbi__get8(z);

   while (!stbi__at_eof(z) && c != '\n') {
      buffer[len++] = c;
      if (len == STBI__HDR_BUFLEN-1) {
         // flush to end of line
         while (!stbi__at_eof(z) && stbi__get8(z) != '\n')
            ;
         break;
      }
      c = (char) stbi__get8(z);
   }

   buffer[len] = 0;
   return buffer;
}

static void stbi__hdr_convert(float *output, stbi_uc *input, int req_comp)
{
   if ( input[3] != 0 ) {
      float f1;
      // Exponent
      f1 = (float) ldexp(1.0f, input[3] - (int)(128 + 8));
      if (req_comp <= 2)
         output[0] = (input[0] + input[1] + input[2]) * f1 / 3;
      else {
         output[0] = input[0] * f1;
         output[1] = input[1] * f1;
         output[2] = input[2] * f1;
      }
      if (req_comp == 2) output[1] = 1;
      if (req_comp == 4) output[3] = 1;
   } else {
      switch (req_comp) {
         case 4: output[3] = 1; /* fallthrough */
         case 3: output[0] = output[1] = output[2] = 0;
                 break;
         case 2: output[1] = 1; /* fallthrough */
         case 1: output[0] = 0;
                 break;
      }
   }
}

static float *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   char buffer[STBI__HDR_BUFLEN];
   char *token;
   int valid = 0;
   int width, height;
   stbi_uc *scanline;
   float *hdr_data;
   int len;
   unsigned char count, value;
   int i, j, k, c1,c2, z;
   const char *headerToken;
   STBI_NOTUSED(ri);

   // Check identifier
   headerToken = stbi__hdr_gettoken(s,buffer);
   if (strcmp(headerToken, "#?RADIANCE") != 0 && strcmp(headerToken, "#?RGBE") != 0)
      return stbi__errpf("not HDR", "Corrupt HDR image");

   // Parse header
   for(;;) {
      token = stbi__hdr_gettoken(s,buffer);
      if (token[0] == 0) break;
      if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
   }

   if (!valid)    return stbi__errpf("unsupported format", "Unsupported HDR format");

   // Parse width and height
   // can't use sscanf() if we're not using stdio!
   token = stbi__hdr_gettoken(s,buffer);
   if (strncmp(token, "-Y ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR format");
   token += 3;
   height = (int) strtol(token, &token, 10);
   while (*token == ' ') ++token;
   if (strncmp(token, "+X ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR format");
   token += 3;
   width = (int) strtol(token, NULL, 10);

   if (height > STBI_MAX_DIMENSIONS) return stbi__errpf("too large","Very large image (corrupt?)");
   if (width > STBI_MAX_DIMENSIONS) return stbi__errpf("too large","Very large image (corrupt?)");

   *x = width;
   *y = height;

   if (comp) *comp = 3;
   if (req_comp == 0) req_comp = 3;

   if (!stbi__mad4sizes_valid(width, height, req_comp, sizeof(float), 0))
      return stbi__errpf("too large", "HDR image is too large");

   // Read data
   hdr_data = (float *) stbi__malloc_mad4(width, height, req_comp, sizeof(float), 0);
   if (!hdr_data)
      return stbi__errpf("outofmem", "Out of memory");

   // Load image data
   // image data is stored as some number of sca
   if ( width < 8 || width >= 32768) {
      // Read flat data
      for (j=0; j < height; ++j) {
         for (i=0; i < width; ++i) {
            stbi_uc rgbe[4];
           main_decode_loop:
            stbi__getn(s, rgbe, 4);
            stbi__hdr_convert(hdr_data + j * width * req_comp + i * req_comp, rgbe, req_comp);
         }
      }
   } else {
      // Read RLE-encoded data
      scanline = NULL;

      for (j = 0; j < height; ++j) {
         c1 = stbi__get8(s);
         c2 = stbi__get8(s);
         len = stbi__get8(s);
         if (c1 != 2 || c2 != 2 || (len & 0x80)) {
            // not run-length encoded, so we have to actually use THIS data as a decoded
            // pixel (note this can't be a valid pixel--one of RGB must be >= 128)
            stbi_uc rgbe[4];
            rgbe[0] = (stbi_uc) c1;
            rgbe[1] = (stbi_uc) c2;
            rgbe[2] = (stbi_uc) len;
            rgbe[3] = (stbi_uc) stbi__get8(s);
            stbi__hdr_convert(hdr_data, rgbe, req_comp);
            i = 1;
            j = 0;
            STBI_FREE(scanline);
            goto main_decode_loop; // yes, this makes no sense
         }
         len «= 8;
         len |= stbi__get8(s);
         if (len != width) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("invalid decoded scanline length", "corrupt HDR"); }
         if (scanline == NULL) {
            scanline = (stbi_uc *) stbi__malloc_mad2(width, 4, 0);
            if (!scanline) {
               STBI_FREE(hdr_data);
               return stbi__errpf("outofmem", "Out of memory");
            }
         }

         for (k = 0; k < 4; ++k) {
            int nleft;
            i = 0;
            while ((nleft = width - i) > 0) {
               count = stbi__get8(s);
               if (count > 128) {
                  // Run
                  value = stbi__get8(s);
                  count -= 128;
                  if ((count == 0) || (count > nleft)) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("corrupt", "bad RLE data in HDR"); }
                  for (z = 0; z < count; ++z)
                     scanline[i++ * 4 + k] = value;
               } else {
                  // Dump
                  if ((count == 0) || (count > nleft)) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("corrupt", "bad RLE data in HDR"); }
                  for (z = 0; z < count; ++z)
                     scanline[i++ * 4 + k] = stbi__get8(s);
               }
            }
         }
         for (i=0; i < width; ++i)
            stbi__hdr_convert(hdr_data+(j*width + i)*req_comp, scanline + i*4, req_comp);
      }
      if (scanline)
         STBI_FREE(scanline);
   }

   return hdr_data;
}

static int stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp)
{
   char buffer[STBI__HDR_BUFLEN];
   char *token;
   int valid = 0;
   int dummy;

   if (!x) x = &dummy;
   if (!y) y = &dummy;
   if (!comp) comp = &dummy;

   if (stbi__hdr_test(s) == 0) {
       stbi__rewind( s );
       return 0;
   }

   for(;;) {
      token = stbi__hdr_gettoken(s,buffer);
      if (token[0] == 0) break;
      if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
   }

   if (!valid) {
       stbi__rewind( s );
       return 0;
   }
   token = stbi__hdr_gettoken(s,buffer);
   if (strncmp(token, "-Y ", 3)) {
       stbi__rewind( s );
       return 0;
   }
   token += 3;
   *y = (int) strtol(token, &token, 10);
   while (*token == ' ') ++token;
   if (strncmp(token, "+X ", 3)) {
       stbi__rewind( s );
       return 0;
   }
   token += 3;
   *x = (int) strtol(token, NULL, 10);
   *comp = 3;
   return 1;
}
#endif // STBI_NO_HDR

#ifndef STBI_NO_BMP
static int stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp)
{
   void *p;
   stbi__bmp_data info;

   info.all_a = 255;
   p = stbi__bmp_parse_header(s, &info);
   if (p == NULL) {
      stbi__rewind( s );
      return 0;
   }
   if (x) *x = s->img_x;
   if (y) *y = s->img_y;
   if (comp) {
      if (info.bpp == 24 && info.ma == 0xff000000)
         *comp = 3;
      else
         *comp = info.ma ? 4 : 3;
   }
   return 1;
}
#endif
```

```
#ifndef STBI_NO_PSD
static int stbi__psd_info(stbi__context *s, int *x, int *y, int *comp)
{
   int channelCount, dummy, depth;
   if (!x) x = &dummy;
   if (!y) y = &dummy;
   if (!comp) comp = &dummy;
   if (stbi__get32be(s) != 0x38425053) {
       stbi__rewind( s );
       return 0;
   }
   if (stbi__get16be(s) != 1) {
       stbi__rewind( s );
       return 0;
   }
   stbi__skip(s, 6);
   channelCount = stbi__get16be(s);
   if (channelCount < 0 || channelCount > 16) {
       stbi__rewind( s );
       return 0;
   }
   *y = stbi__get32be(s);
   *x = stbi__get32be(s);
   depth = stbi__get16be(s);
   if (depth != 8 && depth != 16) {
       stbi__rewind( s );
       return 0;
   }
   if (stbi__get16be(s) != 3) {
       stbi__rewind( s );
       return 0;
   }
   *comp = 4;
   return 1;
}

static int stbi__psd_is16(stbi__context *s)
{
   int channelCount, depth;
   if (stbi__get32be(s) != 0x38425053) {
       stbi__rewind( s );
       return 0;
   }
   if (stbi__get16be(s) != 1) {
       stbi__rewind( s );
       return 0;
   }
   stbi__skip(s, 6);
   channelCount = stbi__get16be(s);
   if (channelCount < 0 || channelCount > 16) {
       stbi__rewind( s );
       return 0;
   }
   STBI_NOTUSED(stbi__get32be(s));
   STBI_NOTUSED(stbi__get32be(s));
   depth = stbi__get16be(s);
   if (depth != 16) {
       stbi__rewind( s );
       return 0;
   }
   return 1;
}
#endif

#ifndef STBI_NO_PIC
static int stbi__pic_info(stbi__context *s, int *x, int *y, int *comp)
{
   int act_comp=0,num_packets=0,chained,dummy;
   stbi__pic_packet packets[10];

   if (!x) x = &dummy;
   if (!y) y = &dummy;
   if (!comp) comp = &dummy;

   if (!stbi__pic_is4(s,"\x53\x80\xF6\x34")) {
      stbi__rewind(s);
      return 0;
   }

   stbi__skip(s, 88);

   *x = stbi__get16be(s);
   *y = stbi__get16be(s);
   if (stbi__at_eof(s)) {
      stbi__rewind( s);
      return 0;
   }
   if ( (*x) != 0 && (1 « 28) / (*x) < (*y)) {
      stbi__rewind( s );
      return 0;
   }

   stbi__skip(s, 8);

   do {
      stbi__pic_packet *packet;

      if (num_packets==sizeof(packets)/sizeof(packets[0]))
         return 0;

      packet = &packets[num_packets++];
      chained = stbi__get8(s);
      packet->size    = stbi__get8(s);
      packet->type    = stbi__get8(s);
      packet->channel = stbi__get8(s);
      act_comp |= packet->channel;

      if (stbi__at_eof(s)) {
          stbi__rewind( s );
          return 0;
      }
      if (packet->size != 8) {
          stbi__rewind( s );
          return 0;
      }
   } while (chained);

   *comp = (act_comp & 0x10 ? 4 : 3);

   return 1;
}
#endif

// ****************************************************************************************************
// Portable Gray Map and Portable Pixel Map loader
// by Ken Miller
//
// PGM: http://netpbm.sourceforge.net/doc/pgm.html
// PPM: http://netpbm.sourceforge.net/doc/ppm.html
//
// Known limitations:
//    Does not support comments in the header section
//    Does not support ASCII image data (formats P2 and P3)

#ifndef STBI_NO_PNM

static int      stbi__pnm_test(stbi__context *s)
{
   char p, t;
   p = (char) stbi__get8(s);
   t = (char) stbi__get8(s);
   if (p != ’P’ || (t != ’5’ && t != ’6’)) {
       stbi__rewind( s );
       return 0;
   }
   return 1;
}

static void *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
   stbi_uc *out;
   STBI_NOTUSED(ri);

   ri->bits_per_channel = stbi__pnm_info(s, (int *)&s->img_x, (int *)&s->img_y, (int *)&s->img_n);
   if (ri->bits_per_channel == 0)
      return 0;

   if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
   if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");

   *x = s->img_x;
   *y = s->img_y;
   if (comp) *comp = s->img_n;

   if (!stbi__mad4sizes_valid(s->img_n, s->img_x, s->img_y, ri->bits_per_channel / 8, 0))
      return stbi__errpuc("too large", "PNM too large");

   out = (stbi_uc *) stbi__malloc_mad4(s->img_n, s->img_x, s->img_y, ri->bits_per_channel / 8, 0);
   if (!out) return stbi__errpuc("outofmem", "Out of memory");
   if (!stbi__getn(s, out, s->img_n * s->img_x * s->img_y * (ri->bits_per_channel / 8))) {
      STBI_FREE(out);
      return stbi__errpuc("bad PNM", "PNM file truncated");
   }

   if (req_comp && req_comp != s->img_n) {
      if (ri->bits_per_channel == 16) {
         out = (stbi_uc *) stbi__convert_format16((stbi__uint16 *) out, s->img_n, req_comp, s->img_x, s->img_y);
      } else {
         out = stbi__convert_format(out, s->img_n, req_comp, s->img_x, s->img_y);
      }
      if (out == NULL) return out; // stbi__convert_format frees input on failure
   }
   return out;
}

static int      stbi__pnm_isspace(char c)
{
   return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';
}

static void     stbi__pnm_skip_whitespace(stbi__context *s, char *c)
{
   for (;;) {
      while (!stbi__at_eof(s) && stbi__pnm_isspace(*c))
         *c = (char) stbi__get8(s);

      if (stbi__at_eof(s) || *c != '#')
         break;

      while (!stbi__at_eof(s) && *c != '\n' && *c != '\r' )
         *c = (char) stbi__get8(s);
   }
}

static int      stbi__pnm_isdigit(char c)
{
   return c >= '0' && c <= '9';
}

static int      stbi__pnm_getinteger(stbi__context *s, char *c)
{
   int value = 0;

   while (!stbi__at_eof(s) && stbi__pnm_isdigit(*c)) {
      value = value*10 + (*c - '0');
      *c = (char) stbi__get8(s);
      if((value > 214748364) || (value == 214748364 && *c > '7'))
          return stbi__err("integer parse overflow", "Parsing an integer in the PPM header overflowed a 32-bit int");
   }

   return value;
}

static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp)
{
   int maxv, dummy;
   char c, p, t;

   if (!x) x = &dummy;
   if (!y) y = &dummy;
   if (!comp) comp = &dummy;

   stbi__rewind(s);

   // Get identifier
   p = (char) stbi__get8(s);
   t = (char) stbi__get8(s);
   if (p != 'P' || (t != '5' && t != '6')) {
       stbi__rewind(s);
       return 0;
   }

   *comp = (t == '6') ? 3 : 1;  // '5' is 1-component .pgm; '6' is 3-component .ppm

   c = (char) stbi__get8(s);
   stbi__pnm_skip_whitespace(s, &c);

   *x = stbi__pnm_getinteger(s, &c); // read width
   if(*x == 0)
       return stbi__err("invalid width", "PPM image header had zero or overflowing width");
   stbi__pnm_skip_whitespace(s, &c);

   *y = stbi__pnm_getinteger(s, &c); // read height
   if (*y == 0)
       return stbi__err("invalid width", "PPM image header had zero or overflowing width");
   stbi__pnm_skip_whitespace(s, &c);

   maxv = stbi__pnm_getinteger(s, &c);  // read max value
   if (maxv > 65535)
      return stbi__err("max value > 65535", "PPM image supports only 8-bit and 16-bit images");
   else if (maxv > 255)
      return 16;
   else
      return 8;
}

static int stbi__pnm_is16(stbi__context *s)
{
   if (stbi__pnm_info(s, NULL, NULL, NULL) == 16)
       return 1;
   return 0;
}
#endif

static int stbi__info_main(stbi__context *s, int *x, int *y, int *comp)
{
   #ifndef STBI_NO_JPEG
   if (stbi__jpeg_info(s, x, y, comp)) return 1;
   #endif

   #ifndef STBI_NO_PNG
   if (stbi__png_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_GIF
   if (stbi__gif_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_BMP
   if (stbi__bmp_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_PSD
   if (stbi__psd_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_PIC
   if (stbi__pic_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_PNM
   if (stbi__pnm_info(s, x, y, comp))  return 1;
   #endif

   #ifndef STBI_NO_HDR
   if (stbi__hdr_info(s, x, y, comp))  return 1;
   #endif

   // test tga last because it's a crappy test!
   #ifndef STBI_NO_TGA
   if (stbi__tga_info(s, x, y, comp))
       return 1;
   #endif
   return stbi__err("unknown image type", "Image not of any known type, or corrupt");
}

static int stbi__is_16_main(stbi__context *s)
{
   #ifndef STBI_NO_PNG
   if (stbi__png_is16(s))  return 1;
   #endif

   #ifndef STBI_NO_PSD
   if (stbi__psd_is16(s))  return 1;
   #endif

   #ifndef STBI_NO_PNM
   if (stbi__pnm_is16(s))  return 1;
   #endif
   return 0;
}

#ifndef STBI_NO_STDIO
STBIDEF int stbi_info(char const *filename, int *x, int *y, int *comp)
{
    FILE *f = stbi__fopen(filename, "rb");
    int result;
    if (!f) return stbi__err("can't fopen", "Unable to open file");
    result = stbi_info_from_file(f, x, y, comp);
    fclose(f);
    return result;
}

STBIDEF int stbi_info_from_file(FILE *f, int *x, int *y, int *comp)
```

```
{
   int r;
   stbi__context s;
   long pos = ftell(f);
   stbi__start_file(&s, f);
   r = stbi__info_main(&s,x,y,comp);
   fseek(f,pos,SEEK_SET);
   return r;
}

STBIDEF int stbi_is_16_bit(char const *filename)
{
    FILE *f = stbi__fopen(filename, "rb");
    int result;
    if (!f) return stbi__err("can't fopen", "Unable to open file");
    result = stbi_is_16_bit_from_file(f);
    fclose(f);
    return result;
}

STBIDEF int stbi_is_16_bit_from_file(FILE *f)
{
   int r;
   stbi__context s;
   long pos = ftell(f);
   stbi__start_file(&s, f);
   r = stbi__is_16_main(&s);
   fseek(f,pos,SEEK_SET);
   return r;
}
#endif // !STBI_NO_STDIO

STBIDEF int stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp)
{
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__info_main(&s,x,y,comp);
}

STBIDEF int stbi_info_from_callbacks(stbi_io_callbacks const *c, void *user, int *x, int *y, int *comp)
{
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *) c, user);
   return stbi__info_main(&s,x,y,comp);
}

STBIDEF int stbi_is_16_bit_from_memory(stbi_uc const *buffer, int len)
{
   stbi__context s;
   stbi__start_mem(&s,buffer,len);
   return stbi__is_16_main(&s);
}

STBIDEF int stbi_is_16_bit_from_callbacks(stbi_io_callbacks const *c, void *user)
{
   stbi__context s;
   stbi__start_callbacks(&s, (stbi_io_callbacks *) c, user);
   return stbi__is_16_main(&s);
}

#endif // STB_IMAGE_IMPLEMENTATION

/*
   revision history:
      2.20  (2019-02-07) support utf8 filenames in Windows; fix warnings and platform ifdefs
      2.19  (2018-02-11) fix warning
      2.18  (2018-01-30) fix warnings
      2.17  (2018-01-29) change sbti__shiftsigned to avoid clang -O2 bug
                         1-bit BMP
                         *_is_16_bit api
                         avoid warnings
      2.16  (2017-07-23) all functions have 16-bit variants;
                         STBI_NO_STDIO works again;
                         compilation fixes;
                         fix rounding in unpremultiply;
                         optimize vertical flip;
                         disable raw_len validation;
                         documentation fixes
      2.15  (2017-03-18) fix png-1,2,4 bug; now all Imagenet JPGs decode;
                         warning fixes; disable run-time SSE detection on gcc;
                         uniform handling of optional "return" values;
                         thread-safe initialization of zlib tables
      2.14  (2017-03-03) remove deprecated STBI_JPEG_OLD; fixes for Imagenet JPGs
      2.13  (2016-11-29) add 16-bit API, only supported for PNG right now
      2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
      2.11  (2016-04-02) allocate large structures on the stack
```

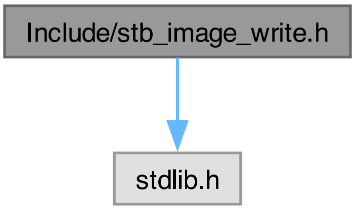
```
                         remove white matting for transparent PSD
                         fix reported channel count for PNG & BMP
                         re-enable SSE2 in non-gcc 64-bit
                         support RGB-formatted JPEG
                         read 16-bit PNGs (only as 8-bit)
      2.10  (2016-01-22) avoid warning introduced in 2.09 by STBI_REALLOC_SIZED
      2.09  (2016-01-16) allow comments in PNM files
                         16-bit-per-pixel TGA (not bit-per-component)
                         info() for TGA could break due to .hdr handling
                         info() for BMP to shares code instead of sloppy parse
                         can use STBI_REALLOC_SIZED if allocator doesn't support realloc
                         code cleanup
      2.08  (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
      2.07  (2015-09-13) fix compiler warnings
                         partial animated GIF support
                         limited 16-bpc PSD support
                         #ifdef unused functions
                         bug with < 92 byte PIC,PNM,HDR,TGA
      2.06  (2015-04-19) fix bug where PSD returns wrong '*comp' value
      2.05  (2015-04-19) fix bug in progressive JPEG handling, fix warning
      2.04  (2015-04-15) try to re-enable SIMD on MinGW 64-bit
      2.03  (2015-04-12) extra corruption checking (mmozeiko)
                         stbi_set_flip_vertically_on_load (nguillemot)
                         fix NEON support; fix mingw support
      2.02  (2015-01-19) fix incorrect assert, fix warning
      2.01  (2015-01-17) fix various warnings; suppress SIMD on gcc 32-bit without -msse2
      2.00b (2014-12-25) fix STBI_MALLOC in progressive JPEG
      2.00  (2014-12-25) optimize JPG, including x86 SSE2 & NEON SIMD (ryg)
                         progressive JPEG (stb)
                         PGM/PPM support (Ken Miller)
                         STBI_MALLOC,STBI_REALLOC,STBI_FREE
                         GIF bugfix -- seemingly never worked
                         STBI_NO_*, STBI_ONLY_*
      1.48  (2014-12-14) fix incorrectly-named assert()
      1.47  (2014-12-14) 1/2/4-bit PNG support, both direct and paletted (Omar Cornut & stb)
                         optimize PNG (ryg)
                         fix bug in interlaced PNG with user-specified channel count (stb)
      1.46  (2014-08-26)
              fix broken tRNS chunk (colorkey-style transparency) in non-paletted PNG
      1.45  (2014-08-16)
              fix MSVC-ARM internal compiler error by wrapping malloc
      1.44  (2014-08-07)
              various warning fixes from Ronny Chevalier
      1.43  (2014-07-15)
              fix MSVC-only compiler problem in code changed in 1.42
      1.42  (2014-07-09)
              don't define _CRT_SECURE_NO_WARNINGS (affects user code)
              fixes to stbi__cleanup_jpeg path
              added STBI_ASSERT to avoid requiring assert.h
      1.41  (2014-06-25)
              fix search&replace from 1.36 that messed up comments/error messages
      1.40  (2014-06-22)
              fix gcc struct-initialization warning
      1.39  (2014-06-15)
              fix to TGA optimization when req_comp != number of components in TGA;
              fix to GIF loading because BMP wasn't rewinding (whoops, no GIFs in my test suite)
              add support for BMP version 5 (more ignored fields)
      1.38  (2014-06-06)
              suppress MSVC warnings on integer casts truncating values
              fix accidental rename of 'skip' field of I/O
      1.37  (2014-06-04)
              remove duplicate typedef
      1.36  (2014-06-03)
              convert to header file single-file library
              if de-iphone isn't set, load iphone images color-swapped instead of returning NULL
      1.35  (2014-05-27)
              various warnings
              fix broken STBI_SIMD path
              fix bug where stbi_load_from_file no longer left file pointer in correct place
              fix broken non-easy path for 32-bit BMP (possibly never used)
              TGA optimization by Arseny Kapoulkine
      1.34  (unknown)
              use STBI_NOTUSED in stbi__resample_row_generic(), fix one more leak in tga failure case
      1.33  (2011-07-14)
              make stbi_is_hdr work in STBI_NO_HDR (as specified), minor compiler-friendly improvements
      1.32  (2011-07-13)
              support for "info" function for all supported filetypes (SpartanJ)
      1.31  (2011-06-20)
              a few more leak fixes, bug in PNG handling (SpartanJ)
      1.30  (2011-06-11)
              added ability to load files via callbacks to accomidate custom input streams (Ben Wenger)
              removed deprecated format-specific test/load functions
              removed support for installable file formats (stbi_loader) -- would have been broken for IO callbacks anyway
              error cases in bmp and tga give messages and don't leak (Raymond Barbiero, grisha)
              fix inefficiency in decoding 32-bit BMP (David Woo)
      1.29  (2010-08-16)
              various warning fixes from Aurelien Pocheville
      1.28  (2010-08-01)
              fix bug in GIF palette transparency (SpartanJ)
      1.27  (2010-08-01)
              cast-to-stbi_uc to fix warnings
      1.26  (2010-07-24)
              fix bug in file buffering for PNG reported by SpartanJ
      1.25  (2010-07-17)
              refix trans_data warning (Won Chun)
      1.24  (2010-07-12)
              perf improvements reading from files on platforms with lock-heavy fgetc()
              minor perf improvements for jpeg
              deprecated type-specific functions so we’ll get feedback if they’re needed
              attempt to fix trans_data warning (Won Chun)
      1.23    fixed bug in iPhone support
      1.22  (2010-07-10)
              removed image *writing* support
              stbi_info support from Jetro Lauha
              GIF support from Jean-Marc Lienher
              iPhone PNG-extensions from James Brown
              warning-fixes from Nicolas Schulz and Janez Zemva (i.stbi__err. Janez (U+017D)emva)
      1.21    fix use of ’stbi_uc’ in header (reported by jon blow)
      1.20    added support for Softimage PIC, by Tom Seddon
      1.19    bug in interlaced PNG corruption check (found by ryg)
      1.18  (2008-08-02)
              fix a threading bug (local mutable static)
      1.17    support interlaced PNG
      1.16    major bugfix - stbi__convert_format converted one too many pixels
      1.15    initialize some fields for thread safety
      1.14    fix threadsafe conversion bug
              header-file-only version (#define STBI_HEADER_FILE_ONLY before including)
      1.13    threadsafe
      1.12    const qualifiers in the API
      1.11    Support installable IDCT, colorspace conversion routines
      1.10    Fixes for 64-bit (don’t use "unsigned long")
              optimized upsampling by Fabian "ryg" Giesen
      1.09    Fix format-conversion for PSD code (bad global variables!)
      1.08    Thatcher Ulrich’s PSD code integrated by Nicolas Schulz
      1.07    attempt to fix C++ warning/errors again
      1.06    attempt to fix C++ warning/errors again
      1.05    fix TGA loading to return correct *comp and use good luminance calc
      1.04    default float alpha is 1, not 255; use ’void *’ for stbi_image_free
      1.03    bugfixes to STBI_NO_STDIO, STBI_NO_HDR
      1.02    support for (subset of) HDR files, float interface for preferred access to them
      1.01    fix bug: possible bug in handling right-side up bmps... not sure
              fix bug: the stbi__bmp_load() and stbi__tga_load() functions didn’t work at all
      1.00    interface to zlib that skips zlib header
      0.99    correct handling of alpha in palette
      0.98    TGA loader by lonesock; dynamically add loaders (untested)
      0.97    jpeg errors on too large a file; also catch another malloc failure
      0.96    fix detection of invalid v value - particleman@mollyrocket forum
      0.95    during header scan, seek to markers in case of padding
      0.94    STBI_NO_STDIO to disable stdio usage; rename all #defines the same
      0.93    handle jpegtran output; verbose errors
      0.92    read 4,8,16,24,32-bit BMP files of several formats
      0.91    output 24-bit Windows 3.0 BMP files
      0.90    fix a few more warnings; bump version number to approach 1.0
      0.61    bugfixes due to Marc LeBlanc, Christopher Lloyd
      0.60    fix compiling as c++
      0.59    fix warnings: merge Dave Moore’s -Wall fixes
      0.58    fix bug: zlib uncompressed mode len/nlen was wrong endian
      0.57    fix bug: jpg last huffman symbol before marker was >9 bits but less than 16 available
      0.56    fix bug: zlib uncompressed mode len vs. nlen
      0.55    fix bug: restart_interval not initialized to 0
      0.54    allow NULL for ’int *comp’
      0.53    fix bug in png 3->4; speedup png decoding
      0.52    png handles req_comp=3,4 directly; minor cleanup; jpeg comments
      0.51    obey req_comp requests, 1-component jpegs return as 1-component,
              on ’test’ only check type, not whether we support this variant
      0.50  (2006-11-19)
              first released version
*/


/*
------------------------------------------------------------------------------
This software is available under 2 licenses -- choose whichever you prefer.
------------------------------------------------------------------------------
ALTERNATIVE A - MIT License
Copyright (c) 2017 Sean Barrett
Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
```

```
so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
------------------------------------------------------------------------------
ALTERNATIVE B - Public Domain (www.unlicense.org)
This is free and unencumbered software released into the public domain.
Anyone is free to copy, modify, publish, use, compile, sell, or distribute this
software, either in source code form or as a compiled binary, for any purpose,
commercial or non-commercial, and by any means.
In jurisdictions that recognize copyright laws, the author or authors of this
software dedicate any and all copyright interest in the software to the public
domain. We make this dedication for the benefit of the public at large and to
the detriment of our heirs and successors. We intend this dedication to be an
overt act of relinquishment in perpetuity of all present and future rights to
this software under copyright law.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
------------------------------------------------------------------------------
*/
```

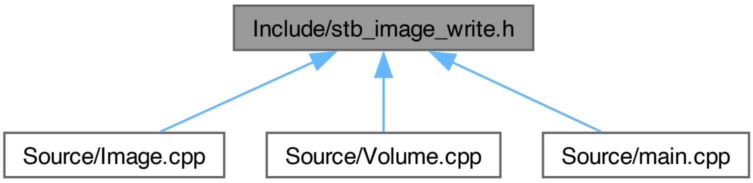
9.46 Include/stb_image_write.h File Reference

```
#include <stdlib.h>
```

Include dependency graph for stb_image_write.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define STBIWDEF extern

Typedefs

- typedef void stbi_write_func(void ∗context, void ∗data, int size)

Functions

- STBIWDEF int stbi_write_png (char const ∗filename, int w, int h, int comp, const void ∗data, int stride_in_↩ bytes)
- STBIWDEF int stbi_write_bmp (char const ∗filename, int w, int h, int comp, const void ∗data)
- STBIWDEF int stbi_write_tga (char const ∗filename, int w, int h, int comp, const void ∗data)
- STBIWDEF int stbi_write_hdr (char const ∗filename, int w, int h, int comp, const float ∗data)
- STBIWDEF int stbi_write_jpg (char const ∗filename, int x, int y, int comp, const void ∗data, int quality)
- STBIWDEF int stbi_write_png_to_func (stbi_write_func ∗func, void ∗context, int w, int h, int comp, const void ∗data, int stride_in_bytes)
- STBIWDEF int stbi_write_bmp_to_func (stbi_write_func ∗func, void ∗context, int w, int h, int comp, const void ∗data)
- STBIWDEF int stbi_write_tga_to_func (stbi_write_func ∗func, void ∗context, int w, int h, int comp, const void ∗data)
- STBIWDEF int stbi_write_hdr_to_func (stbi_write_func ∗func, void ∗context, int w, int h, int comp, const float ∗data)
- STBIWDEF int stbi_write_jpg_to_func (stbi_write_func ∗func, void ∗context, int x, int y, int comp, const void ∗data, int quality)
- STBIWDEF void stbi_flip_vertically_on_write (int flip_boolean)

Variables

- STBIWDEF int stbi_write_tga_with_rle
- STBIWDEF int stbi_write_png_compression_level
- STBIWDEF int stbi_write_force_png_filter

9.46.1 Macro Definition Documentation

9.46.1.1 STBIWDEF

```
#define STBIWDEF extern
```

9.46.2 Typedef Documentation

9.46.2.1 stbi_write_func

```
typedef void stbi_write_func(void *context, void *data, int size)
```

9.46.3 Function Documentation

9.46.3.1 stbi_flip_vertically_on_write()

```
STBIWDEF void stbi_flip_vertically_on_write (
             int flip_boolean )
```

9.46.3.2 stbi_write_bmp()

```
STBIWDEF int stbi_write_bmp (
             char const * filename,
             int w,
             int h,
             int comp,
             const void * data )
```

9.46.3.3 stbi_write_bmp_to_func()

```
STBIWDEF int stbi_write_bmp_to_func (
             stbi_write_func * func,
             void * context,
             int w,
             int h,
             int comp,
             const void * data )
```

9.46.3.4 stbi_write_hdr()

```
STBIWDEF int stbi_write_hdr (
             char const * filename,
             int w,
             int h,
             int comp,
             const float * data )
```

9.46.3.5 stbi_write_hdr_to_func()

```
STBIWDEF int stbi_write_hdr_to_func (
             stbi_write_func * func,
             void * context,
             int w,
             int h,
             int comp,
             const float * data )
```

9.46.3.6 stbi_write_jpg()

```
STBIWDEF int stbi_write_jpg (
             char const * filename,
             int x,
             int y,
             int comp,
             const void * data,
             int quality )
```

9.46.3.7 stbi_write_jpg_to_func()

```
STBIWDEF int stbi_write_jpg_to_func (
             stbi_write_func * func,
             void * context,
             int x,
             int y,
             int comp,
             const void * data,
             int quality )
```

9.46.3.8 stbi_write_png()

```
STBIWDEF int stbi_write_png (
             char const * filename,
             int w,
             int h,
             int comp,
             const void * data,
             int stride_in_bytes )
```

9.46.3.9 stbi_write_png_to_func()

```
STBIWDEF int stbi_write_png_to_func (
             stbi_write_func * func,
             void * context,
             int w,
             int h,
             int comp,
             const void * data,
             int stride_in_bytes )
```

9.46.3.10 stbi_write_tga()

```
STBIWDEF int stbi_write_tga (
             char const * filename,
             int w,
             int h,
             int comp,
             const void * data )
```

9.46.3.11 stbi_write_tga_to_func()

```
STBIWDEF int stbi_write_tga_to_func (
             stbi_write_func * func,
             void * context,
             int w,
             int h,
             int comp,
             const void * data )
```

9.46.4 Variable Documentation

9.46.4.1 stbi_write_force_png_filter

```
STBIWDEF int stbi_write_force_png_filter
```

9.46.4.2 stbi_write_png_compression_level

```
STBIWDEF int stbi_write_png_compression_level
```

9.46.4.3 stbi_write_tga_with_rle

```
STBIWDEF int stbi_write_tga_with_rle
```

9.47 stb_image_write.h

Go to the documentation of this file.

```
/* stb_image_write - v1.16 - public domain - http://nothings.org/stb
   writes out PNG/BMP/TGA/JPEG/HDR images to C stdio - Sean Barrett 2010-2015
                                     no warranty implied; use at your own risk

   Before #including,

       #define STB_IMAGE_WRITE_IMPLEMENTATION

   in the file that you want to have the implementation.

   Will probably not work correctly with strict-aliasing optimizations.

ABOUT:

   This header file is a library for writing images to C stdio or a callback.

   The PNG output is not optimal; it is 20-50% larger than the file
   written by a decent optimizing implementation; though providing a custom
   zlib compress function (see STBIW_ZLIB_COMPRESS) can mitigate that.
   This library is designed for source code compactness and simplicity,
   not optimal image file size or run-time performance.

BUILDING:

   You can #define STBIW_ASSERT(x) before the #include to avoid using assert.h.
   You can #define STBIW_MALLOC(), STBIW_REALLOC(), and STBIW_FREE() to replace
   malloc,realloc,free.
   You can #define STBIW_MOVE() to replace memmove()
   You can #define STBIW_ZLIB_COMPRESS to use a custom zlib-style compress function
   for PNG compression (instead of the builtin one), it must have the following signature:
   unsigned char * my_compress(unsigned char *data, int data_len, int *out_len, int quality);
   The returned data will be freed with STBIW_FREE() (free() by default),
   so it must be heap allocated with STBIW_MALLOC() (malloc() by default),

UNICODE:

   If compiling for Windows and you wish to use Unicode filenames, compile
   with
       #define STBIW_WINDOWS_UTF8
   and pass utf8-encoded filenames. Call stbiw_convert_wchar_to_utf8 to convert
   Windows wchar_t filenames to utf8.

USAGE:

   There are five functions, one for each image file format:

     int stbi_write_png(char const *filename, int w, int h, int comp, const void *data, int stride_in_bytes);
     int stbi_write_bmp(char const *filename, int w, int h, int comp, const void *data);
     int stbi_write_tga(char const *filename, int w, int h, int comp, const void *data);
     int stbi_write_jpg(char const *filename, int w, int h, int comp, const void *data, int quality);
     int stbi_write_hdr(char const *filename, int w, int h, int comp, const float *data);

     void stbi_flip_vertically_on_write(int flag); // flag is non-zero to flip data vertically

   There are also five equivalent functions that use an arbitrary write function. You are
   expected to open/close your file-equivalent before and after calling these:

     int stbi_write_png_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data, int stride_in_bytes);
     int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data);
     int stbi_write_tga_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data);
     int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const float *data);
     int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void *data, int quality);

   where the callback is:
      void stbi_write_func(void *context, void *data, int size);

   You can configure it with these global variables:
      int stbi_write_tga_with_rle;             // defaults to true; set to 0 to disable RLE
      int stbi_write_png_compression_level;    // defaults to 8; set to higher for more compression
      int stbi_write_force_png_filter;         // defaults to -1; set to 0..5 to force a filter mode


   You can define STBI_WRITE_NO_STDIO to disable the file variant of these
   functions, so the library will not use stdio.h at all. However, this will
   also disable HDR writing, because it requires stdio for formatted output.

   Each function returns 0 on failure and non-0 on success.

   The functions create an image file defined by the parameters. The image
   is a rectangle of pixels stored from left-to-right, top-to-bottom.
   Each pixel contains 'comp' channels of data stored interleaved with 8-bits
   per channel, in the following order: 1=Y, 2=YA, 3=RGB, 4=RGBA. (Y is
   monochrome color.) The rectangle is 'w' pixels wide and 'h' pixels tall.
   The *data pointer points to the first byte of the top-left-most pixel.
   For PNG, "stride_in_bytes" is the distance in bytes from the first byte of
   a row of pixels to the first byte of the next row of pixels.

   PNG creates output files with the same number of components as the input.
   The BMP format expands Y to RGB in the file format and does not
   output alpha.

   PNG supports writing rectangles of data even when the bytes storing rows of
   data are not consecutive in memory (e.g. sub-rectangles of a larger image),
   by supplying the stride between the beginning of adjacent rows. The other
   formats do not. (Thus you cannot write a native-format BMP through the BMP
   writer, both because it is in BGR order and because it may have padding
   at the end of the line.)

   PNG allows you to set the deflate compression level by setting the global
   variable 'stbi_write_png_compression_level' (it defaults to 8).

   HDR expects linear float data. Since the format is always 32-bit rgb(e)
   data, alpha (if provided) is discarded, and for monochrome data it is
   replicated across all three channels.

   TGA supports RLE or non-RLE compressed data. To use non-RLE-compressed
   data, set the global variable 'stbi_write_tga_with_rle' to 0.

   JPEG does ignore alpha channels in input data; quality is between 1 and 100.
   Higher quality looks better but results in a bigger image.
   JPEG baseline (no JPEG progressive).

CREDITS:


   Sean Barrett           -    PNG/BMP/TGA
   Baldur Karlsson        -    HDR
   Jean-Sebastien Guay    -    TGA monochrome
   Tim Kelsey             -    misc enhancements
   Alan Hickman           -    TGA RLE
   Emmanuel Julien        -    initial file IO callback implementation
   Jon Olick              -    original jo_jpeg.cpp code
   Daniel Gibson          -    integrate JPEG, allow external zlib
   Aarni Koskela          -    allow choosing PNG filter

   bugfixes:
      github:Chribba
      Guillaume Chereau
      github:jry2
      github:romigrou
      Sergio Gonzalez
      Jonas Karlsson
      Filip Wasil
      Thatcher Ulrich
      github:poppolopoppo
      Patrick Boettcher
      github:xeekworx
      Cap Petschulat
      Simon Rodriguez
      Ivan Tikhonov
      github:ignotion
      Adam Schackart
      Andrew Kensler

LICENSE

  See end of file for license information.

*/

#ifndef INCLUDE_STB_IMAGE_WRITE_H
#define INCLUDE_STB_IMAGE_WRITE_H

#include <stdlib.h>

// if STB_IMAGE_WRITE_STATIC causes problems, try defining STBIWDEF to 'inline' or 'static inline'
#ifndef STBIWDEF
#ifdef STB_IMAGE_WRITE_STATIC
#define STBIWDEF  static
#else
#ifdef __cplusplus
#define STBIWDEF  extern "C"
#else
#define STBIWDEF  extern
#endif
#endif
#endif

#ifndef STB_IMAGE_WRITE_STATIC  // C++ forbids static forward declarations
STBIWDEF int stbi_write_tga_with_rle;
STBIWDEF int stbi_write_png_compression_level;
STBIWDEF int stbi_write_force_png_filter;
#endif

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_png(char const *filename, int w, int h, int comp, const void  *data, int stride_in_bytes);
STBIWDEF int stbi_write_bmp(char const *filename, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_tga(char const *filename, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_hdr(char const *filename, int w, int h, int comp, const float *data);
STBIWDEF int stbi_write_jpg(char const *filename, int x, int y, int comp, const void  *data, int quality);

#ifdef STBIW_WINDOWS_UTF8
STBIWDEF int stbiw_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input);
#endif
#endif

typedef void stbi_write_func(void *context, void *data, int size);

STBIWDEF int stbi_write_png_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data, int stride_in_bytes);
STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const void  *data);
STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const float *data);
STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void  *data, int quality);

STBIWDEF void stbi_flip_vertically_on_write(int flip_boolean);

#endif//INCLUDE_STB_IMAGE_WRITE_H

#ifdef STB_IMAGE_WRITE_IMPLEMENTATION

#ifdef _WIN32
   #ifndef _CRT_SECURE_NO_WARNINGS
   #define _CRT_SECURE_NO_WARNINGS
   #endif
   #ifndef _CRT_NONSTDC_NO_DEPRECATE
   #define _CRT_NONSTDC_NO_DEPRECATE
   #endif
#endif

#ifndef STBI_WRITE_NO_STDIO
#include <stdio.h>
#endif // STBI_WRITE_NO_STDIO

#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#if defined(STBIW_MALLOC) && defined(STBIW_FREE) && (defined(STBIW_REALLOC) || defined(STBIW_REALLOC_SIZED))
// ok
#elif !defined(STBIW_MALLOC) && !defined(STBIW_FREE) && !defined(STBIW_REALLOC) && !defined(STBIW_REALLOC_SIZED)
// ok
#else
#error "Must define all or none of STBIW_MALLOC, STBIW_FREE, and STBIW_REALLOC (or STBIW_REALLOC_SIZED)."
#endif

#ifndef STBIW_MALLOC
#define STBIW_MALLOC(sz)        malloc(sz)
#define STBIW_REALLOC(p,newsz)  realloc(p,newsz)
#define STBIW_FREE(p)           free(p)
#endif

#ifndef STBIW_REALLOC_SIZED
#define STBIW_REALLOC_SIZED(p,oldsz,newsz) STBIW_REALLOC(p,newsz)
#endif


#ifndef STBIW_MEMMOVE
#define STBIW_MEMMOVE(a,b,sz) memmove(a,b,sz)
#endif


#ifndef STBIW_ASSERT
#include <assert.h>
#define STBIW_ASSERT(x) assert(x)
#endif

#define STBIW_UCHAR(x) (unsigned char) ((x) & 0xff)

#ifdef STB_IMAGE_WRITE_STATIC
static int stbi_write_png_compression_level = 8;
static int stbi_write_tga_with_rle = 1;
static int stbi_write_force_png_filter = -1;
#else
int stbi_write_png_compression_level = 8;
int stbi_write_tga_with_rle = 1;
int stbi_write_force_png_filter = -1;
#endif

static int stbi__flip_vertically_on_write = 0;

STBIWDEF void stbi_flip_vertically_on_write(int flag)
{
   stbi__flip_vertically_on_write = flag;
}

typedef struct
{
   stbi_write_func *func;
   void *context;
   unsigned char buffer[64];
   int buf_used;
} stbi__write_context;

// initialize a callback-based context
static void stbi__start_write_callbacks(stbi__write_context *s, stbi_write_func *c, void *context)
{
   s->func    = c;
   s->context = context;
}

#ifndef STBI_WRITE_NO_STDIO

static void stbi__stdio_write(void *context, void *data, int size)
{
   fwrite(data,1,size,(FILE*) context);
}

#if defined(_WIN32) && defined(STBIW_WINDOWS_UTF8)
#ifdef __cplusplus
#define STBIW_EXTERN extern "C"
#else
#define STBIW_EXTERN extern
#endif
STBIW_EXTERN __declspec(dllimport) int __stdcall MultiByteToWideChar(unsigned int cp, unsigned long flags, const char *str, int cbmb, wchar_t *widestr, int cchwide);
STBIW_EXTERN __declspec(dllimport) int __stdcall WideCharToMultiByte(unsigned int cp, unsigned long flags, const wchar_t *widestr, int cchwide, char *str, int cbmb, const char *defchar, int *used_default);

STBIWDEF int stbiw_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input)
{
   return WideCharToMultiByte(65001 /* UTF8 */, 0, input, -1, buffer, (int) bufferlen, NULL, NULL);
}
#endif

static FILE *stbiw__fopen(char const *filename, char const *mode)
{
   FILE *f;
#if defined(_WIN32) && defined(STBIW_WINDOWS_UTF8)
   wchar_t wMode[64];
   wchar_t wFilename[1024];
   if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, filename, -1, wFilename, sizeof(wFilename)/sizeof(*wFilename)))
      return 0;

   if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, mode, -1, wMode, sizeof(wMode)/sizeof(*wMode)))
      return 0;

#if defined(_MSC_VER) && _MSC_VER >= 1400
   if (0 != _wfopen_s(&f, wFilename, wMode))
      f = 0;
#else
   f = _wfopen(wFilename, wMode);
#endif

#elif defined(_MSC_VER) && _MSC_VER >= 1400
   if (0 != fopen_s(&f, filename, mode))
      f=0;
#else
   f = fopen(filename, mode);
#endif
   return f;
}

static int stbi__start_write_file(stbi__write_context *s, const char *filename)
{
   FILE *f = stbiw__fopen(filename, "wb");
   stbi__start_write_callbacks(s, stbi__stdio_write, (void *) f);
   return f != NULL;
}

static void stbi__end_write_file(stbi__write_context *s)
{
   fclose((FILE *)s->context);
}

#endif // !STBI_WRITE_NO_STDIO

typedef unsigned int stbiw_uint32;
typedef int stb_image_write_test[sizeof(stbiw_uint32)==4 ? 1 : -1];

static void stbiw__writefv(stbi__write_context *s, const char *fmt, va_list v)
{
   while (*fmt) {
      switch (*fmt++) {
         case ’ ’: break;
         case ’1’: { unsigned char x = STBIW_UCHAR(va_arg(v, int));
                     s->func(s->context,&x,1);
                     break; }
         case ’2’: { int x = va_arg(v,int);
                     unsigned char b[2];
                     b[0] = STBIW_UCHAR(x);
                     b[1] = STBIW_UCHAR(x»8);
                     s->func(s->context,b,2);
                     break; }
         case '4': { stbiw_uint32 x = va_arg(v,int);
                     unsigned char b[4];
                     b[0]=STBIW_UCHAR(x);
                     b[1]=STBIW_UCHAR(x»8);
                     b[2]=STBIW_UCHAR(x»16);
                     b[3]=STBIW_UCHAR(x»24);
                     s->func(s->context,b,4);
                     break; }
         default:
            STBIW_ASSERT(0);
            return;
      }
   }
}

static void stbiw__writef(stbi__write_context *s, const char *fmt, ...)
{
   va_list v;
   va_start(v, fmt);
   stbiw__writefv(s, fmt, v);
   va_end(v);
}

static void stbiw__write_flush(stbi__write_context *s)
{
   if (s->buf_used) {
      s->func(s->context, &s->buffer, s->buf_used);
      s->buf_used = 0;
   }
}

static void stbiw__putc(stbi__write_context *s, unsigned char c)
{
   s->func(s->context, &c, 1);
}

static void stbiw__write1(stbi__write_context *s, unsigned char a)
{
   if ((size_t)s->buf_used + 1 > sizeof(s->buffer))
      stbiw__write_flush(s);
   s->buffer[s->buf_used++] = a;
}

static void stbiw__write3(stbi__write_context *s, unsigned char a, unsigned char b, unsigned char c)
{
   int n;
   if ((size_t)s->buf_used + 3 > sizeof(s->buffer))
      stbiw__write_flush(s);
   n = s->buf_used;
   s->buf_used = n+3;
   s->buffer[n+0] = a;
   s->buffer[n+1] = b;
   s->buffer[n+2] = c;
}

static void stbiw__write_pixel(stbi__write_context *s, int rgb_dir, int comp, int write_alpha, int expand_mono, unsigned char *d)
{
   unsigned char bg[3] = { 255, 0, 255}, px[3];
   int k;

   if (write_alpha < 0)
      stbiw__write1(s, d[comp - 1]);

   switch (comp) {
      case 2: // 2 pixels = mono + alpha, alpha is written separately, so same as 1-channel case
      case 1:
         if (expand_mono)
            stbiw__write3(s, d[0], d[0], d[0]); // monochrome bmp
         else
            stbiw__write1(s, d[0]);  // monochrome TGA
         break;
      case 4:
         if (!write_alpha) {
            // composite against pink background
            for (k = 0; k < 3; ++k)
               px[k] = bg[k] + ((d[k] - bg[k]) * d[3]) / 255;
            stbiw__write3(s, px[1 - rgb_dir], px[1], px[1 + rgb_dir]);
            break;
         }
         /* FALLTHROUGH */
      case 3:
         stbiw__write3(s, d[1 - rgb_dir], d[1], d[1 + rgb_dir]);
         break;
   }
   if (write_alpha > 0)
```

```
      stbiw__write1(s, d[comp - 1]);
}

static void stbiw__write_pixels(stbi__write_context *s, int rgb_dir, int vdir, int x, int y, int comp, void *data, int write_alpha, int scanline_pad, int expand_mono)
{
   stbiw_uint32 zero = 0;
   int i,j, j_end;

   if (y <= 0)
      return;

   if (stbi__flip_vertically_on_write)
      vdir *= -1;

   if (vdir < 0) {
      j_end = -1; j = y-1;
   } else {
      j_end =  y; j = 0;
   }

   for (; j != j_end; j += vdir) {
      for (i=0; i < x; ++i) {
         unsigned char *d = (unsigned char *) data + (j*x+i)*comp;
         stbiw__write_pixel(s, rgb_dir, comp, write_alpha, expand_mono, d);
      }
      stbiw__write_flush(s);
      s->func(s->context, &zero, scanline_pad);
   }
}

static int stbiw__outfile(stbi__write_context *s, int rgb_dir, int vdir, int x, int y, int comp, int expand_mono, void *data, int alpha, int pad, const char *fmt, ...)
{
   if (y < 0 || x < 0) {
      return 0;
   } else {
      va_list v;
      va_start(v, fmt);
      stbiw__writefv(s, fmt, v);
      va_end(v);
      stbiw__write_pixels(s,rgb_dir,vdir,x,y,comp,data,alpha,pad, expand_mono);
      return 1;
   }
}

static int stbi_write_bmp_core(stbi__write_context *s, int x, int y, int comp, const void *data)
{
   if (comp != 4) {
      // write RGB bitmap
      int pad = (-x*3) & 3;
      return stbiw__outfile(s,-1,-1,x,y,comp,1,(void *) data,0,pad,
              "11 4 22 4" "4 44 22 444444",
              'B', 'M', 14+40+(x*3+pad)*y, 0,0, 14+40,  // file header
               40, x,y, 1,24, 0,0,0,0,0,0);             // bitmap header
   } else {
      // RGBA bitmaps need a v4 header
      // use BI_BITFIELDS mode with 32bpp and alpha mask
      // (straight BI_RGB with alpha mask doesn't work in most readers)
      return stbiw__outfile(s,-1,-1,x,y,comp,1,(void *)data,1,0,
         "11 4 22 4" "4 44 22 444444 4444 4 444 444 444 444",
         'B', 'M', 14+108+x*y*4, 0, 0, 14+108, // file header
         108, x,y, 1,32, 3,0,0,0,0,0, 0xff0000,0xff00,0xff,0xff000000u, 0, 0,0,0, 0,0,0, 0,0,0, 0,0,0); // bitmap V4 header
   }
}

STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void *data)
{
   stbi__write_context s = { 0 };
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_bmp_core(&s, x, y, comp, data);
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_bmp(char const *filename, int x, int y, int comp, const void *data)
{
   stbi__write_context s = { 0 };
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_bmp_core(&s, x, y, comp, data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif
```

```c

static int stbi_write_tga_core(stbi__write_context *s, int x, int y, int comp, void *data)
{
   int has_alpha = (comp == 2 || comp == 4);
   int colorbytes = has_alpha ? comp-1 : comp;
   int format = colorbytes < 2 ? 3 : 2; // 3 color channels (RGB/RGBA) = 2, 1 color channel (Y/YA) = 3

   if (y < 0 || x < 0)
      return 0;

   if (!stbi_write_tga_with_rle) {
      return stbiw__outfile(s, -1, -1, x, y, comp, 0, (void *) data, has_alpha, 0,
         "111 221 2222 11", 0, 0, format, 0, 0, 0, 0, 0, x, y, (colorbytes + has_alpha) * 8, has_alpha * 8);
   } else {
      int i,j,k;
      int jend, jdir;

      stbiw__writef(s, "111 221 2222 11", 0,0,format+8, 0,0,0, 0,0,x,y, (colorbytes + has_alpha) * 8, has_alpha * 8);

      if (stbi__flip_vertically_on_write) {
         j = 0;
         jend = y;
         jdir = 1;
      } else {
         j = y-1;
         jend = -1;
         jdir = -1;
      }
      for (; j != jend; j += jdir) {
         unsigned char *row = (unsigned char *) data + j * x * comp;
         int len;

         for (i = 0; i < x; i += len) {
            unsigned char *begin = row + i * comp;
            int diff = 1;
            len = 1;

            if (i < x - 1) {
               ++len;
               diff = memcmp(begin, row + (i + 1) * comp, comp);
               if (diff) {
                  const unsigned char *prev = begin;
                  for (k = i + 2; k < x && len < 128; ++k) {
                     if (memcmp(prev, row + k * comp, comp)) {
                        prev += comp;
                        ++len;
                     } else {
                        --len;
                        break;
                     }
                  }
               } else {
                  for (k = i + 2; k < x && len < 128; ++k) {
                     if (!memcmp(begin, row + k * comp, comp)) {
                        ++len;
                     } else {
                        break;
                     }
                  }
               }
            }

            if (diff) {
               unsigned char header = STBIW_UCHAR(len - 1);
               stbiw__write1(s, header);
               for (k = 0; k < len; ++k) {
                  stbiw__write_pixel(s, -1, comp, has_alpha, 0, begin + k * comp);
               }
            } else {
               unsigned char header = STBIW_UCHAR(len - 129);
               stbiw__write1(s, header);
               stbiw__write_pixel(s, -1, comp, has_alpha, 0, begin);
            }
         }
      }
      stbiw__write_flush(s);
   }
   return 1;
}

STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void *data)
{
   stbi__write_context s = { 0 };
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_tga_core(&s, x, y, comp, (void *) data);
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_tga(char const *filename, int x, int y, int comp, const void *data)
{
   stbi__write_context s = { 0 };
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_tga_core(&s, x, y, comp, (void *) data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif

// *************************************************************************************************
// Radiance RGBE HDR writer
// by Baldur Karlsson

#define stbiw__max(a, b)  ((a) > (b) ? (a) : (b))

#ifndef STBI_WRITE_NO_STDIO

static void stbiw__linear_to_rgbe(unsigned char *rgbe, float *linear)
{
   int exponent;
   float maxcomp = stbiw__max(linear[0], stbiw__max(linear[1], linear[2]));

   if (maxcomp < 1e-32f) {
      rgbe[0] = rgbe[1] = rgbe[2] = rgbe[3] = 0;
   } else {
      float normalize = (float) frexp(maxcomp, &exponent) * 256.0f/maxcomp;

      rgbe[0] = (unsigned char)(linear[0] * normalize);
      rgbe[1] = (unsigned char)(linear[1] * normalize);
      rgbe[2] = (unsigned char)(linear[2] * normalize);
      rgbe[3] = (unsigned char)(exponent + 128);
   }
}

static void stbiw__write_run_data(stbi__write_context *s, int length, unsigned char databyte)
{
   unsigned char lengthbyte = STBIW_UCHAR(length+128);
   STBIW_ASSERT(length+128 <= 255);
   s->func(s->context, &lengthbyte, 1);
   s->func(s->context, &databyte, 1);
}

static void stbiw__write_dump_data(stbi__write_context *s, int length, unsigned char *data)
{
   unsigned char lengthbyte = STBIW_UCHAR(length);
   STBIW_ASSERT(length <= 128); // inconsistent with spec but consistent with official code
   s->func(s->context, &lengthbyte, 1);
   s->func(s->context, data, length);
}

static void stbiw__write_hdr_scanline(stbi__write_context *s, int width, int ncomp, unsigned char *scratch, float *scanline)
{
   unsigned char scanlineheader[4] = { 2, 2, 0, 0 };
   unsigned char rgbe[4];
   float linear[3];
   int x;

   scanlineheader[2] = (width&0xff00)»8;
   scanlineheader[3] = (width&0x00ff);

   /* skip RLE for images too small or large */
   if (width < 8 || width >= 32768) {
      for (x=0; x < width; x++) {
         switch (ncomp) {
            case 4: /* fallthrough */
            case 3: linear[2] = scanline[x*ncomp + 2];
                    linear[1] = scanline[x*ncomp + 1];
                    linear[0] = scanline[x*ncomp + 0];
                    break;
            default:
                    linear[0] = linear[1] = linear[2] = scanline[x*ncomp + 0];
                    break;
         }
         stbiw__linear_to_rgbe(rgbe, linear);
         s->func(s->context, rgbe, 4);
      }
   } else {
      int c,r;
      /* encode into scratch buffer */
      for (x=0; x < width; x++) {
         switch(ncomp) {
            case 4: /* fallthrough */
            case 3: linear[2] = scanline[x*ncomp + 2];
                    linear[1] = scanline[x*ncomp + 1];
                    linear[0] = scanline[x*ncomp + 0];
                    break;
            default:
                    linear[0] = linear[1] = linear[2] = scanline[x*ncomp + 0];
                    break;
         }
         stbiw__linear_to_rgbe(rgbe, linear);
         scratch[x + width*0] = rgbe[0];
         scratch[x + width*1] = rgbe[1];
         scratch[x + width*2] = rgbe[2];
         scratch[x + width*3] = rgbe[3];
      }

      s->func(s->context, scanlineheader, 4);

      /* RLE each component separately */
      for (c=0; c < 4; c++) {
         unsigned char *comp = &scratch[width*c];

         x = 0;
         while (x < width) {
            // find first run
            r = x;
            while (r+2 < width) {
               if (comp[r] == comp[r+1] && comp[r] == comp[r+2])
                  break;
               ++r;
            }
            if (r+2 >= width)
               r = width;
            // dump up to first run
            while (x < r) {
               int len = r-x;
               if (len > 128) len = 128;
               stbiw__write_dump_data(s, len, &comp[x]);
               x += len;
            }
            // if there's a run, output it
            if (r+2 < width) { // same test as what we break out of in search loop, so only true if we break'd
               // find next byte after run
               while (r < width && comp[r] == comp[x])
                  ++r;
               // output run up to r
               while (x < r) {
                  int len = r-x;
                  if (len > 127) len = 127;
                  stbiw__write_run_data(s, len, comp[x]);
                  x += len;
               }
            }
         }
      }
   }
}

static int stbi_write_hdr_core(stbi__write_context *s, int x, int y, int comp, float *data)
{
   if (y <= 0 || x <= 0 || data == NULL)
      return 0;
   else {
      // Each component is stored separately. Allocate scratch space for full output scanline.
      unsigned char *scratch = (unsigned char *) STBIW_MALLOC(x*4);
      int i, len;
      char buffer[128];
      char header[] = "#?RADIANCE\n# Written by stb_image_write.h\nFORMAT=32-bit_rle_rgbe\n";
      s->func(s->context, header, sizeof(header)-1);

#ifdef __STDC_LIB_EXT1__
      len = sprintf_s(buffer, sizeof(buffer), "EXPOSURE=          1.0000000000000\n\n-Y %d +X %d\n", y, x);
#else
      len = sprintf(buffer, "EXPOSURE=          1.0000000000000\n\n-Y %d +X %d\n", y, x);
#endif
      s->func(s->context, buffer, len);

      for(i=0; i < y; i++)
         stbiw__write_hdr_scanline(s, x, comp, scratch, data + comp*x*(stbi__flip_vertically_on_write ? y-1-i : i));
      STBIW_FREE(scratch);
      return 1;
   }
```

```
}

STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const float *data)
{
   stbi__write_context s = { 0 };
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_hdr_core(&s, x, y, comp, (float *) data);
}

STBIWDEF int stbi_write_hdr(char const *filename, int x, int y, int comp, const float *data)
{
   stbi__write_context s = { 0 };
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_hdr_core(&s, x, y, comp, (float *) data);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif // STBI_WRITE_NO_STDIO


//
// PNG writer
//

#ifndef STBIW_ZLIB_COMPRESS
// stretchy buffer; stbiw__sbpush() == vector<>::push_back() -- stbiw__sbcount() == vector<>::size()
#define stbiw__sbraw(a) ((int *) (void *) (a) - 2)
#define stbiw__sbm(a)   stbiw__sbraw(a)[0]
#define stbiw__sbn(a)   stbiw__sbraw(a)[1]

#define stbiw__sbneedgrow(a,n)  ((a)==0 || stbiw__sbn(a)+n >= stbiw__sbm(a))
#define stbiw__sbmaybegrow(a,n) (stbiw__sbneedgrow(a,(n)) ? stbiw__sbgrow(a,n) : 0)
#define stbiw__sbgrow(a,n)  stbiw__sbgrowf((void **) &(a), (n), sizeof(*(a)))

#define stbiw__sbpush(a, v)      (stbiw__sbmaybegrow(a,1), (a)[stbiw__sbn(a)++] = (v))
#define stbiw__sbcount(a)        ((a) ? stbiw__sbn(a) : 0)
#define stbiw__sbfree(a)         ((a) ? STBIW_FREE(stbiw__sbraw(a)),0 : 0)

static void *stbiw__sbgrowf(void **arr, int increment, int itemsize)
{
   int m = *arr ? 2*stbiw__sbm(*arr)+increment : increment+1;
   void *p = STBIW_REALLOC_SIZED(*arr ? stbiw__sbraw(*arr) : 0, *arr ? (stbiw__sbm(*arr)*itemsize + sizeof(int)*2) : 0, itemsize * m + sizeof(int)*2);
   STBIW_ASSERT(p);
   if (p) {
      if (!*arr) ((int *) p)[1] = 0;
      *arr = (void *) ((int *) p + 2);
      stbiw__sbm(*arr) = m;
   }
   return *arr;
}

static unsigned char *stbiw__zlib_flushf(unsigned char *data, unsigned int *bitbuffer, int *bitcount)
{
   while (*bitcount >= 8) {
      stbiw__sbpush(data, STBIW_UCHAR(*bitbuffer));
      *bitbuffer »= 8;
      *bitcount -= 8;
   }
   return data;
}

static int stbiw__zlib_bitrev(int code, int codebits)
{
   int res=0;
   while (codebits--) {
      res = (res « 1) | (code & 1);
      code »= 1;
   }
   return res;
}

static unsigned int stbiw__zlib_countm(unsigned char *a, unsigned char *b, int limit)
{
   int i;
   for (i=0; i < limit && i < 258; ++i)
      if (a[i] != b[i]) break;
   return i;
}

static unsigned int stbiw__zhash(unsigned char *data)
{
   stbiw_uint32 hash = data[0] + (data[1] « 8) + (data[2] « 16);
   hash ^= hash « 3;
   hash += hash » 5;
   hash ^= hash « 4;
   hash += hash » 17;
   hash ^= hash « 25;
   hash += hash » 6;
   return hash;
}

#define stbiw__zlib_flush() (out = stbiw__zlib_flushf(out, &bitbuf, &bitcount))
#define stbiw__zlib_add(code,codebits) \
      (bitbuf |= (code) « bitcount, bitcount += (codebits), stbiw__zlib_flush())
#define stbiw__zlib_huffa(b,c)  stbiw__zlib_add(stbiw__zlib_bitrev(b,c),c)
// default huffman tables
#define stbiw__zlib_huff1(n)  stbiw__zlib_huffa(0x30 + (n), 8)
#define stbiw__zlib_huff2(n)  stbiw__zlib_huffa(0x190 + (n)-144, 9)
#define stbiw__zlib_huff3(n)  stbiw__zlib_huffa(0 + (n)-256,7)
#define stbiw__zlib_huff4(n)  stbiw__zlib_huffa(0xc0 + (n)-280,8)
#define stbiw__zlib_huff(n)  ((n) <= 143 ? stbiw__zlib_huff1(n) : (n) <= 255 ? stbiw__zlib_huff2(n) : (n) <= 279 ? stbiw__zlib_huff3(n) : stbiw__zlib_huff4(n))
#define stbiw__zlib_huffb(n) ((n) <= 143 ? stbiw__zlib_huff1(n) : stbiw__zlib_huff2(n))

#define stbiw__ZHASH   16384

#endif // STBIW_ZLIB_COMPRESS

STBIWDEF unsigned char * stbi_zlib_compress(unsigned char *data, int data_len, int *out_len, int quality)
{
#ifdef STBIW_ZLIB_COMPRESS
   // user provided a zlib compress implementation, use that
   return STBIW_ZLIB_COMPRESS(data, data_len, out_len, quality);
#else // use builtin
   static unsigned short lengthc[] = { 3,4,5,6,7,8,9,10,11,13,15,17,19,23,27,31,35,43,51,59,67,83,99,115,131,163,195,227,258, 259 };
   static unsigned char  lengtheb[]= { 0,0,0,0,0,0,0,0,  0,  1,  1,  1,  1,  2,  2,  2,  2,  3,  3,  3,  3,  4,  4,  4,  4,  5,  5,  5,  5,  0 };
   static unsigned short distc[]   = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577, 32768 };
   static unsigned char  disteb[]  = { 0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13 };
   unsigned int bitbuf=0;
   int i,j, bitcount=0;
   unsigned char *out = NULL;
   unsigned char ***hash_table = (unsigned char***) STBIW_MALLOC(stbiw__ZHASH * sizeof(unsigned char**));
   if (hash_table == NULL)
      return NULL;
   if (quality < 5) quality = 5;

   stbiw__sbpush(out, 0x78);   // DEFLATE 32K window
   stbiw__sbpush(out, 0x5e);   // FLEVEL = 1
   stbiw__zlib_add(1,1);  // BFINAL = 1
   stbiw__zlib_add(1,2);  // BTYPE = 1 -- fixed huffman

   for (i=0; i < stbiw__ZHASH; ++i)
      hash_table[i] = NULL;

   i=0;
   while (i < data_len-3) {
      // hash next 3 bytes of data to be compressed
      int h = stbiw__zhash(data+i)&(stbiw__ZHASH-1), best=3;
      unsigned char *bestloc = 0;
      unsigned char **hlist = hash_table[h];
      int n = stbiw__sbcount(hlist);
      for (j=0; j < n; ++j) {
         if (hlist[j]-data > i-32768) { // if entry lies within window
            int d = stbiw__zlib_countm(hlist[j], data+i, data_len-i);
            if (d >= best) { best=d; bestloc=hlist[j]; }
         }
      }
      // when hash table entry is too long, delete half the entries
      if (hash_table[h] && stbiw__sbn(hash_table[h]) == 2*quality) {
         STBIW_MEMMOVE(hash_table[h], hash_table[h]+quality, sizeof(hash_table[h][0])*quality);
         stbiw__sbn(hash_table[h]) = quality;
      }
      stbiw__sbpush(hash_table[h],data+i);

      if (bestloc) {
         // "lazy matching" - check match at *next* byte, and if it's better, do cur byte as literal
         h = stbiw__zhash(data+i+1)&(stbiw__ZHASH-1);
         hlist = hash_table[h];
         n = stbiw__sbcount(hlist);
         for (j=0; j < n; ++j) {
            if (hlist[j]-data > i-32767) {
               int e = stbiw__zlib_countm(hlist[j], data+i+1, data_len-i-1);
               if (e > best) { // if next match is better, bail on current match
```

```
                  bestloc = NULL;
                  break;
               }
            }
         }
      }

      if (bestloc) {
         int d = (int) (data+i - bestloc); // distance back
         STBIW_ASSERT(d <= 32767 && best <= 258);
         for (j=0; best > lengthc[j+1]-1; ++j);
         stbiw__zlib_huff(j+257);
         if (lengtheb[j]) stbiw__zlib_add(best - lengthc[j], lengtheb[j]);
         for (j=0; d > distc[j+1]-1; ++j);
         stbiw__zlib_add(stbiw__zlib_bitrev(j,5),5);
         if (disteb[j]) stbiw__zlib_add(d - distc[j], disteb[j]);
         i += best;
      } else {
         stbiw__zlib_huffb(data[i]);
         ++i;
      }
   }
   // write out final bytes
   for (;i < data_len; ++i)
      stbiw__zlib_huffb(data[i]);
   stbiw__zlib_huff(256); // end of block
   // pad with 0 bits to byte boundary
   while (bitcount)
      stbiw__zlib_add(0,1);

   for (i=0; i < stbiw__ZHASH; ++i)
      (void) stbiw__sbfree(hash_table[i]);
   STBIW_FREE(hash_table);

   // store uncompressed instead if compression was worse
   if (stbiw__sbn(out) > data_len + 2 + ((data_len+32766)/32767)*5) {
      stbiw__sbn(out) = 2;  // truncate to DEFLATE 32K window and FLEVEL = 1
      for (j = 0; j < data_len;) {
         int blocklen = data_len - j;
         if (blocklen > 32767) blocklen = 32767;
         stbiw__sbpush(out, data_len - j == blocklen); // BFINAL = ?, BTYPE = 0 -- no compression
         stbiw__sbpush(out, STBIW_UCHAR(blocklen)); // LEN
         stbiw__sbpush(out, STBIW_UCHAR(blocklen » 8));
         stbiw__sbpush(out, STBIW_UCHAR(~blocklen)); // NLEN
         stbiw__sbpush(out, STBIW_UCHAR(~blocklen » 8));
         memcpy(out+stbiw__sbn(out), data+j, blocklen);
         stbiw__sbn(out) += blocklen;
         j += blocklen;
      }
   }

   {
      // compute adler32 on input
      unsigned int s1=1, s2=0;
      int blocklen = (int) (data_len % 5552);
      j=0;
      while (j < data_len) {
         for (i=0; i < blocklen; ++i) { s1 += data[j+i]; s2 += s1; }
         s1 %= 65521; s2 %= 65521;
         j += blocklen;
         blocklen = 5552;
      }
      stbiw__sbpush(out, STBIW_UCHAR(s2 » 8));
      stbiw__sbpush(out, STBIW_UCHAR(s2));
      stbiw__sbpush(out, STBIW_UCHAR(s1 » 8));
      stbiw__sbpush(out, STBIW_UCHAR(s1));
   }
   *out_len = stbiw__sbn(out);
   // make returned pointer freeable
   STBIW_MEMMOVE(stbiw__sbraw(out), out, *out_len);
   return (unsigned char *) stbiw__sbraw(out);
#endif // STBIW_ZLIB_COMPRESS
}

static unsigned int stbiw__crc32(unsigned char *buffer, int len)
{
#ifdef STBIW_CRC32
    return STBIW_CRC32(buffer, len);
#else
   static unsigned int crc_table[256] =
   {
      0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA, 0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
      0x0eDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988, 0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
      0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE, 0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,
      0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC, 0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,
      0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172, 0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
      0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940, 0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,
      0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116, 0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,
      0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924, 0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,
      0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A, 0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,
      0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818, 0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
      0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E, 0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,
      0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C, 0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,
      0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2, 0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,
      0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0, 0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,
      0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086, 0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
      0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4, 0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,
      0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A, 0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,
      0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8, 0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,
      0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE, 0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,
      0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC, 0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
      0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252, 0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,
      0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60, 0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,
      0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236, 0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,
      0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04, 0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,
      0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A, 0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
      0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38, 0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,
      0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E, 0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,
      0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C, 0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,
      0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2, 0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,
      0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0, 0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
      0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6, 0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,
      0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94, 0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D
   };

   unsigned int crc = ~0u;
   int i;
   for (i=0; i < len; ++i)
      crc = (crc » 8) ^ crc_table[buffer[i] ^ (crc & 0xff)];
   return ~crc;
#endif
}

#define stbiw__wpng4(o,a,b,c,d) ((o)[0]=STBIW_UCHAR(a),(o)[1]=STBIW_UCHAR(b),(o)[2]=STBIW_UCHAR(c),(o)[3]=STBIW_UCHAR(d),(o)+=4)
#define stbiw__wp32(data,v) stbiw__wpng4(data, (v)»24,(v)»16,(v)»8,(v));
#define stbiw__wptag(data,s) stbiw__wpng4(data, s[0],s[1],s[2],s[3])

static void stbiw__wpcrc(unsigned char **data, int len)
{
   unsigned int crc = stbiw__crc32(*data - len - 4, len+4);
   stbiw__wp32(*data, crc);
}

static unsigned char stbiw__paeth(int a, int b, int c)
{
   int p = a + b - c, pa = abs(p-a), pb = abs(p-b), pc = abs(p-c);
   if (pa <= pb && pa <= pc) return STBIW_UCHAR(a);
   if (pb <= pc) return STBIW_UCHAR(b);
   return STBIW_UCHAR(c);
}

// @OPTIMIZE: provide an option that always forces left-predict or paeth predict
static void stbiw__encode_png_line(unsigned char *pixels, int stride_bytes, int width, int height, int y, int n, int filter_type, signed char *line_buffer)
{
   static int mapping[] = { 0,1,2,3,4 };
   static int firstmap[] = { 0,1,0,5,6 };
   int *mymap = (y != 0) ? mapping : firstmap;
   int i;
   int type = mymap[filter_type];
   unsigned char *z = pixels + stride_bytes * (stbi__flip_vertically_on_write ? height-1-y : y);
   int signed_stride = stbi__flip_vertically_on_write ? -stride_bytes : stride_bytes;

   if (type==0) {
      memcpy(line_buffer, z, width*n);
      return;
   }

   // first loop isn’t optimized since it’s just one pixel
   for (i = 0; i < n; ++i) {
      switch (type) {
         case 1: line_buffer[i] = z[i]; break;
         case 2: line_buffer[i] = z[i] - z[i-signed_stride]; break;
         case 3: line_buffer[i] = z[i] - (z[i-signed_stride]»1); break;
         case 4: line_buffer[i] = (signed char) (z[i] - stbiw__paeth(0,z[i-signed_stride],0)); break;
         case 5: line_buffer[i] = z[i]; break;
         case 6: line_buffer[i] = z[i]; break;
      }
   }
   switch (type) {
      case 1: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - z[i-n]; break;
      case 2: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - z[i-signed_stride]; break;
      case 3: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - ((z[i-n] + z[i-signed_stride])»1);
```

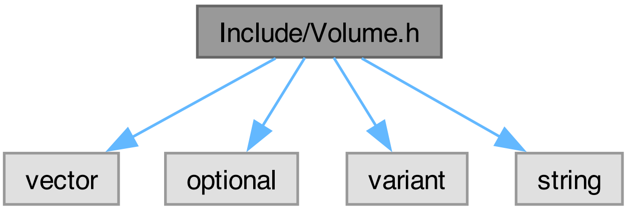
```
break;
      case 4: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - stbiw__paeth(z[i-n], z[i-signed_stride], z[i-signed_stride-n]); break;
      case 5: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - (z[i-n]»1); break;
      case 6: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - stbiw__paeth(z[i-n], 0,0); break;
   }
}

STBIWDEF unsigned char *stbi_write_png_to_mem(const unsigned char *pixels, int stride_bytes, int x, int y, int n, int *out_len)
{
   int force_filter = stbi_write_force_png_filter;
   int ctype[5] = { -1, 0, 4, 2, 6 };
   unsigned char sig[8] = { 137,80,78,71,13,10,26,10 };
   unsigned char *out,*o, *filt, *zlib;
   signed char *line_buffer;
   int j,zlen;

   if (stride_bytes == 0)
      stride_bytes = x * n;

   if (force_filter >= 5) {
      force_filter = -1;
   }

   filt = (unsigned char *) STBIW_MALLOC((x*n+1) * y); if (!filt) return 0;
   line_buffer = (signed char *) STBIW_MALLOC(x * n); if (!line_buffer) { STBIW_FREE(filt); return 0; }
   for (j=0; j < y; ++j) {
      int filter_type;
      if (force_filter > -1) {
         filter_type = force_filter;
         stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, force_filter, line_buffer);
      } else { // Estimate the best filter by running through all of them:
         int best_filter = 0, best_filter_val = 0x7fffffff, est, i;
         for (filter_type = 0; filter_type < 5; filter_type++) {
            stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, filter_type, line_buffer);

            // Estimate the entropy of the line using this filter; the less, the better.
            est = 0;
            for (i = 0; i < x*n; ++i) {
               est += abs((signed char) line_buffer[i]);
            }
            if (est < best_filter_val) {
               best_filter_val = est;
               best_filter = filter_type;
            }
         }
         if (filter_type != best_filter) {  // If the last iteration already got us the best filter, don’t redo it
            stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, best_filter, line_buffer);
            filter_type = best_filter;
         }
      }
      // when we get here, filter_type contains the filter type, and line_buffer contains the data
      filt[j*(x*n+1)] = (unsigned char) filter_type;
      STBIW_MEMMOVE(filt+j*(x*n+1)+1, line_buffer, x*n);
   }
   STBIW_FREE(line_buffer);
   zlib = stbi_zlib_compress(filt, y*( x*n+1), &zlen, stbi_write_png_compression_level);
   STBIW_FREE(filt);
   if (!zlib) return 0;

   // each tag requires 12 bytes of overhead
   out = (unsigned char *) STBIW_MALLOC(8 + 12+13 + 12+zlen + 12);
   if (!out) return 0;
   *out_len = 8 + 12+13 + 12+zlen + 12;

   o=out;
   STBIW_MEMMOVE(o,sig,8); o+= 8;
   stbiw__wp32(o, 13); // header length
   stbiw__wptag(o, "IHDR");
   stbiw__wp32(o, x);
   stbiw__wp32(o, y);
   *o++ = 8;
   *o++ = STBIW_UCHAR(ctype[n]);
   *o++ = 0;
   *o++ = 0;
   *o++ = 0;
   stbiw__wpcrc(&o,13);

   stbiw__wp32(o, zlen);
   stbiw__wptag(o, "IDAT");
   STBIW_MEMMOVE(o, zlib, zlen);
   o += zlen;
   STBIW_FREE(zlib);
   stbiw__wpcrc(&o, zlen);

   stbiw__wp32(o,0);
   stbiw__wptag(o, "IEND");
   stbiw__wpcrc(&o,0);

   STBIW_ASSERT(o == out + *out_len);

   return out;
}

#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_png(char const *filename, int x, int y, int comp, const void *data, int stride_bytes)
{
   FILE *f;
   int len;
   unsigned char *png = stbi_write_png_to_mem((const unsigned char *) data, stride_bytes, x, y, comp, &len);
   if (png == NULL) return 0;

   f = stbiw__fopen(filename, "wb");
   if (!f) { STBIW_FREE(png); return 0; }
   fwrite(png, 1, len, f);
   fclose(f);
   STBIW_FREE(png);
   return 1;
}
#endif

STBIWDEF int stbi_write_png_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void *data, int stride_bytes)
{
   int len;
   unsigned char *png = stbi_write_png_to_mem((const unsigned char *) data, stride_bytes, x, y, comp, &len);
   if (png == NULL) return 0;
   func(context, png, len);
   STBIW_FREE(png);
   return 1;
}


/* ***************************************************************************
 *
 * JPEG writer
 *
 * This is based on Jon Olick's jo_jpeg.cpp:
 * public domain Simple, Minimalistic JPEG writer - http://www.jonolick.com/code.html
 */

static const unsigned char stbiw__jpg_ZigZag[] = { 0,1,5,6,14,15,27,28,2,4,7,13,16,26,29,42,3,8,12,17,25,30,41,43,9,11,18,
24,31,40,44,53,10,19,23,32,39,45,52,54,20,22,33,38,46,51,55,60,21,34,37,47,50,56,59,61,35,36,48,49,57,58,62,63 };

static void stbiw__jpg_writeBits(stbi__write_context *s, int *bitBufP, int *bitCntP, const unsigned short *bs) {
   int bitBuf = *bitBufP, bitCnt = *bitCntP;
   bitCnt += bs[1];
   bitBuf |= bs[0] « (24 - bitCnt);
   while(bitCnt >= 8) {
      unsigned char c = (bitBuf » 16) & 255;
      stbiw__putc(s, c);
      if(c == 255) {
         stbiw__putc(s, 0);
      }
      bitBuf «= 8;
      bitCnt -= 8;
   }
   *bitBufP = bitBuf;
   *bitCntP = bitCnt;
}

static void stbiw__jpg_DCT(float *d0p, float *d1p, float *d2p, float *d3p, float *d4p, float *d5p, float *d6p, float *d7p) {
   float d0 = *d0p, d1 = *d1p, d2 = *d2p, d3 = *d3p, d4 = *d4p, d5 = *d5p, d6 = *d6p, d7 = *d7p;
   float z1, z2, z3, z4, z5, z11, z13;

   float tmp0 = d0 + d7;
   float tmp7 = d0 - d7;
   float tmp1 = d1 + d6;
   float tmp6 = d1 - d6;
   float tmp2 = d2 + d5;
   float tmp5 = d2 - d5;
   float tmp3 = d3 + d4;
   float tmp4 = d3 - d4;

   // Even part
   float tmp10 = tmp0 + tmp3;   // phase 2
   float tmp13 = tmp0 - tmp3;
   float tmp11 = tmp1 + tmp2;
   float tmp12 = tmp1 - tmp2;

   d0 = tmp10 + tmp11;       // phase 3
   d4 = tmp10 - tmp11;

   z1 = (tmp12 + tmp13) * 0.707106781f; // c4
   d2 = tmp13 + z1;       // phase 5
   d6 = tmp13 - z1;

   // Odd part
   tmp10 = tmp4 + tmp5;       // phase 2
   tmp11 = tmp5 + tmp6;
   tmp12 = tmp6 + tmp7;

   // The rotator is modified from fig 4-8 to avoid extra negations.
   z5 = (tmp10 - tmp12) * 0.382683433f; // c6
   z2 = tmp10 * 0.541196100f + z5; // c2-c6
   z4 = tmp12 * 1.306562965f + z5; // c2+c6
   z3 = tmp11 * 0.707106781f; // c4

   z11 = tmp7 + z3;      // phase 5
   z13 = tmp7 - z3;

   *d5p = z13 + z2;         // phase 6
   *d3p = z13 - z2;
   *d1p = z11 + z4;
   *d7p = z11 - z4;

   *d0p = d0;  *d2p = d2;  *d4p = d4;  *d6p = d6;
}

static void stbiw__jpg_calcBits(int val, unsigned short bits[2]) {
   int tmp1 = val < 0 ? -val : val;
   val = val < 0 ? val-1 : val;
   bits[1] = 1;
   while(tmp1 »= 1) {
      ++bits[1];
   }
   bits[0] = val & ((1«bits[1])-1);
}

static int stbiw__jpg_processDU(stbi__write_context *s, int *bitBuf, int *bitCnt, float *CDU, int
du_stride, float *fdtbl, int DC, const unsigned short HTDC[256][2], const unsigned short HTAC[256][2])
{
   const unsigned short EOB[2] = { HTAC[0x00][0], HTAC[0x00][1] };
   const unsigned short M16zeroes[2] = { HTAC[0xF0][0], HTAC[0xF0][1] };
   int dataOff, i, j, n, diff, end0pos, x, y;
   int DU[64];

   // DCT rows
   for(dataOff=0, n=du_stride*8; dataOff<n; dataOff+=du_stride) {
      stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+1], &CDU[dataOff+2], &CDU[dataOff+3],
&CDU[dataOff+4], &CDU[dataOff+5], &CDU[dataOff+6], &CDU[dataOff+7]);
   }
   // DCT columns
   for(dataOff=0; dataOff<8; ++dataOff) {
      stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+du_stride], &CDU[dataOff+du_stride*2],
&CDU[dataOff+du_stride*3], &CDU[dataOff+du_stride*4],
                     &CDU[dataOff+du_stride*5], &CDU[dataOff+du_stride*6], &CDU[dataOff+du_stride*7]);
   }
   // Quantize/descale/zigzag the coefficients
   for(y = 0, j=0; y < 8; ++y) {
      for(x = 0; x < 8; ++x,++j) {
         float v;
         i = y*du_stride+x;
         v = CDU[i]*fdtbl[j];
         // DU[stbiw__jpg_ZigZag[j]] = (int)(v < 0 ? ceilf(v - 0.5f) : floorf(v + 0.5f));
         // ceilf() and floorf() are C99, not C89, but I /think/ they're not needed here anyway?
         DU[stbiw__jpg_ZigZag[j]] = (int)(v < 0 ? v - 0.5f : v + 0.5f);
      }
   }

   // Encode DC
   diff = DU[0] - DC;
   if (diff == 0) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTDC[0]);
   } else {
      unsigned short bits[2];
      stbiw__jpg_calcBits(diff, bits);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTDC[bits[1]]);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
   }
   // Encode ACs
   end0pos = 63;
   for(; (end0pos>0)&&(DU[end0pos]==0); --end0pos) {
   }
   // end0pos = first element in reverse order !=0
   if(end0pos == 0) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
      return DU[0];
   }
   for(i = 1; i <= end0pos; ++i) {
      int startpos = i;
      int nrzeroes;
      unsigned short bits[2];
      for (; DU[i]==0 && i<=end0pos; ++i) {
      }
      nrzeroes = i-startpos;
      if ( nrzeroes >= 16 ) {
         int lng = nrzeroes»4;
         int nrmarker;
         for (nrmarker=1; nrmarker <= lng; ++nrmarker)
            stbiw__jpg_writeBits(s, bitBuf, bitCnt, M16zeroes);
         nrzeroes &= 15;
      }
      stbiw__jpg_calcBits(DU[i], bits);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTAC[(nrzeroes«4)+bits[1]]);
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
   }
   if(end0pos != 63) {
      stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
   }
   return DU[0];
}

static int stbi_write_jpg_core(stbi__write_context *s, int width, int height, int comp, const void* data, int quality) {
   // Constants that don't pollute global namespace
   static const unsigned char std_dc_luminance_nrcodes[] = {0,0,1,5,1,1,1,1,1,1,0,0,0,0,0,0,0};
   static const unsigned char std_dc_luminance_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};
   static const unsigned char std_ac_luminance_nrcodes[] = {0,0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,0x7d};
   static const unsigned char std_ac_luminance_values[] = {
0x01,0x02,0x03,0x00,0x04,0x11,0x05,0x12,0x21,0x31,0x41,0x06,0x13,0x51,0x61,0x07,0x22,0x71,0x14,0x32,0x81,0x91,0xa1,0x08,
0x23,0x42,0xb1,0xc1,0x15,0x52,0xd1,0xf0,0x24,0x33,0x62,0x72,0x82,0x09,0x0a,0x16,0x17,0x18,0x19,0x1a,0x25,0x26,0x27,0x28,
0x29,0x2a,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,0x59,
0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x83,0x84,0x85,0x86,0x87,0x88,0x89,
0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,0xb5,0xb6,
0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xe1,0xe2,
      0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
   };
   static const unsigned char std_dc_chrominance_nrcodes[] = {0,0,3,1,1,1,1,1,1,1,1,1,0,0,0,0,0};
   static const unsigned char std_dc_chrominance_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};
   static const unsigned char std_ac_chrominance_nrcodes[] = {0,0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,0x77};
   static const unsigned char std_ac_chrominance_values[] = {
0x00,0x01,0x02,0x03,0x11,0x04,0x05,0x21,0x31,0x06,0x12,0x41,0x51,0x07,0x61,0x71,0x13,0x22,0x32,0x81,0x08,0x14,0x42,0x91,
0xa1,0xb1,0xc1,0x09,0x23,0x33,0x52,0xf0,0x15,0x62,0x72,0xd1,0x0a,0x16,0x24,0x34,0xe1,0x25,0xf1,0x17,0x18,0x19,0x1a,0x26,
0x27,0x28,0x29,0x2a,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,
0x59,0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x82,0x83,0x84,0x85,0x86,0x87,
0x88,0x89,0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,
0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,
      0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
   };
   // Huffman tables
   static const unsigned short YDC_HT[256][2] = { {0,2},{2,3},{3,3},{4,3},{5,3},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9}};
   static const unsigned short UVDC_HT[256][2] = { {0,2},{1,2},{2,2},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9},{1022,10},{2046,11}};
   static const unsigned short YAC_HT[256][2] = {
{10,4},{0,2},{1,2},{4,3},{11,4},{26,5},{120,7},{248,8},{1014,10},{65410,16},{65411,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,
{12,4},{27,5},{121,7},{502,9},{2038,11},{65412,16},{65413,16},{65414,16},{65415,16},{65416,16},{0,0},{0,0},{0,0},{0,0},
{28,5},{249,8},{1015,10},{4084,12},{65417,16},{65418,16},{65419,16},{65420,16},{65421,16},{65422,16},{0,0},{0,0},{0,0},
{58,6},{503,9},{4085,12},{65423,16},{65424,16},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{0,0},{0,0},{0,0},
{59,6},{1016,10},{65430,16},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{0,0},{0,0},{0,
{122,7},{2039,11},{65438,16},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{0,0},{0,0},{
{123,7},{4086,12},{65446,16},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{0,0},{0,0},{
{250,8},{4087,12},{65454,16},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{0,0},{0,0},{
{504,9},{32704,15},{65462,16},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{0,0},{0,0},
{505,9},{65470,16},{65471,16},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{0,0},{0,0},
{506,9},{65479,16},{65480,16},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{0,0},{0,0},
{1017,10},{65488,16},{65489,16},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{0,0},{0,0
{1018,10},{65497,16},{65498,16},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{0,0},{0,0
{2040,11},{65506,16},{65507,16},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{0,0},{0,0
{65515,16},{65516,16},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{0,0},{0,
{2041,11},{65525,16},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
   };
   static const unsigned short UVAC_HT[256][2] = {
{0,2},{1,2},{4,3},{10,4},{24,5},{25,5},{56,6},{120,7},{500,9},{1014,10},{4084,12},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},
{11,4},{57,6},{246,8},{501,9},{2038,11},{4085,12},{65416,16},{65417,16},{65418,16},{65419,16},{0,0},{0,0},{0,0},{0,0},{
{26,5},{247,8},{1015,10},{4086,12},{32706,15},{65420,16},{65421,16},{65422,16},{65423,16},{65424,16},{0,0},{0,0},{0,0},
{27,5},{248,8},{1016,10},{4087,12},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{65430,16},{0,0},{0,0},{0,0},
{58,6},{502,9},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{65438,16},{0,0},{0,0},{0,0
{59,6},{1017,10},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{65446,16},{0,0},{0,0},{0,
{121,7},{2039,11},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{65454,16},{0,0},{0,0},{
{122,7},{2040,11},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{65462,16},{0,0},{0,0},{
{249,8},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{65470,16},{65471,16},{0,0},{0,0},
{503,9},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{65479,16},{65480,16},{0,0},{0,0},
{504,9},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{65488,16},{65489,16},{0,0},{0,0},
{505,9},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{65497,16},{65498,16},{0,0},{0,0},
{506,9},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{65506,16},{65507,16},{0,0},{0,0},
{2041,11},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{65515,16},{65516,16},{0,0},{0,0
{16352,14},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{65525,16},{0,0},{0,
{1018,10},{32707,15},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
   };
   static const int YQT[] =
{16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,24,40,57,69,56,14,17,22,29,51,87,80,62,18,22,
37,56,68,109,103,77,24,35,55,64,81,104,113,92,49,64,78,87,103,121,120,101,72,92,95,98,112,100,103,99};
   static const int UVQT[] =
{17,18,24,47,99,99,99,99,18,21,26,66,99,99,99,99,24,26,56,99,99,99,99,99,47,66,99,99,99,99,99,99,
99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99};
   static const float aasf[] = { 1.0f * 2.828427125f, 1.387039845f * 2.828427125f, 1.306562965f *
2.828427125f, 1.175875602f * 2.828427125f,
                                 1.0f * 2.828427125f, 0.785694958f * 2.828427125f, 0.541196100f *
2.828427125f, 0.275899379f * 2.828427125f };

   int row, col, i, k, subsample;
   float fdtbl_Y[64], fdtbl_UV[64];
   unsigned char YTable[64], UVTable[64];

   if(!data || !width || !height || comp > 4 || comp < 1) {
      return 0;
   }

   quality = quality ? quality : 90;
   subsample = quality <= 90 ? 1 : 0;
   quality = quality < 1 ? 1 : quality > 100 ? 100 : quality;
   quality = quality < 50 ? 5000 / quality : 200 - quality * 2;

   for(i = 0; i < 64; ++i) {
      int uvti, yti = (YQT[i]*quality+50)/100;
      YTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (yti < 1 ? 1 : yti > 255 ? 255 : yti);
      uvti = (UVQT[i]*quality+50)/100;
      UVTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (uvti < 1 ? 1 : uvti > 255 ? 255 : uvti);
   }

   for(row = 0, k = 0; row < 8; ++row) {
      for(col = 0; col < 8; ++col, ++k) {
         fdtbl_Y[k]  = 1 / (YTable [stbiw__jpg_ZigZag[k]] * aasf[row] * aasf[col]);
         fdtbl_UV[k] = 1 / (UVTable[stbiw__jpg_ZigZag[k]] * aasf[row] * aasf[col]);
      }
   }

   // Write Headers
   {
      static const unsigned char head0[] = { 0xFF,0xD8,0xFF,0xE0,0,0x10,’J’,’F’,’I’,’F’,0,1,1,0,0,1,0,1,0,0,0xFF,0xDB,0,0x84,0 };
      static const unsigned char head2[] = { 0xFF,0xDA,0,0xC,3,1,0,2,0x11,3,0x11,0,0x3F,0 };
      const unsigned char head1[] = { 0xFF,0xC0,0,0x11,8,(unsigned char)(height»8),STBIW_UCHAR(height),(unsigned char)(width»8),STBIW_UCHAR(width),
                                      3,1,(unsigned char)(subsample?0x22:0x11),0,2,0x11,1,3,0x11,1,0xFF,0xC4,0x01,0xA2,0 };
      s->func(s->context, (void*)head0, sizeof(head0));
      s->func(s->context, (void*)YTable, sizeof(YTable));
      stbiw__putc(s, 1);
      s->func(s->context, UVTable, sizeof(UVTable));
      s->func(s->context, (void*)head1, sizeof(head1));
      s->func(s->context, (void*)(std_dc_luminance_nrcodes+1), sizeof(std_dc_luminance_nrcodes)-1);
      s->func(s->context, (void*)std_dc_luminance_values, sizeof(std_dc_luminance_values));
      stbiw__putc(s, 0x10); // HTYACinfo
      s->func(s->context, (void*)(std_ac_luminance_nrcodes+1), sizeof(std_ac_luminance_nrcodes)-1);
      s->func(s->context, (void*)std_ac_luminance_values, sizeof(std_ac_luminance_values));
      stbiw__putc(s, 1); // HTUDCinfo
      s->func(s->context, (void*)(std_dc_chrominance_nrcodes+1), sizeof(std_dc_chrominance_nrcodes)-1);
      s->func(s->context, (void*)std_dc_chrominance_values, sizeof(std_dc_chrominance_values));
      stbiw__putc(s, 0x11); // HTUACinfo
      s->func(s->context, (void*)(std_ac_chrominance_nrcodes+1), sizeof(std_ac_chrominance_nrcodes)-1);
      s->func(s->context, (void*)std_ac_chrominance_values, sizeof(std_ac_chrominance_values));
      s->func(s->context, (void*)head2, sizeof(head2));
   }

   // Encode 8x8 macroblocks
   {
      static const unsigned short fillBits[] = {0x7F, 7};
      int DCY=0, DCU=0, DCV=0;
      int bitBuf=0, bitCnt=0;
      // comp == 2 is grey+alpha (alpha is ignored)
      int ofsG = comp > 2 ? 1 : 0, ofsB = comp > 2 ? 2 : 0;
      const unsigned char *dataR = (const unsigned char *)data;
      const unsigned char *dataG = dataR + ofsG;
      const unsigned char *dataB = dataR + ofsB;
      int x, y, pos;
      if(subsample) {
         for(y = 0; y < height; y += 16) {
            for(x = 0; x < width; x += 16) {
               float Y[256], U[256], V[256];
               for(row = y, pos = 0; row < y+16; ++row) {
                  // row >= height => use last input row
                  int clamped_row = (row < height) ? row : height - 1;
                  int base_p = (stbi__flip_vertically_on_write ? (height-1-clamped_row) : clamped_row)*width*comp;
                  for(col = x; col < x+16; ++col, ++pos) {
                     // if col >= width => use pixel from last input column
                     int p = base_p + ((col < width) ? col : (width-1))*comp;
                     float r = dataR[p], g = dataG[p], b = dataB[p];
                     Y[pos]= +0.29900f*r + 0.58700f*g + 0.11400f*b - 128;
                     U[pos]= -0.16874f*r - 0.33126f*g + 0.50000f*b;
                     V[pos]= +0.50000f*r - 0.41869f*g - 0.08131f*b;
                  }
               }
               DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+0,   16, fdtbl_Y, DCY, YDC_HT, YAC_HT);
               DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+8,   16, fdtbl_Y, DCY, YDC_HT, YAC_HT);
               DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+128, 16, fdtbl_Y, DCY, YDC_HT, YAC_HT);
               DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+136, 16, fdtbl_Y, DCY, YDC_HT, YAC_HT);

               // subsample U,V
               {
                  float subU[64], subV[64];
                  int yy, xx;
                  for(yy = 0, pos = 0; yy < 8; ++yy) {
                     for(xx = 0; xx < 8; ++xx, ++pos) {
                        int j = yy*32+xx*2;
                        subU[pos] = (U[j+0] + U[j+1] + U[j+16] + U[j+17]) * 0.25f;
                        subV[pos] = (V[j+0] + V[j+1] + V[j+16] + V[j+17]) * 0.25f;
                     }
                  }
                  DCU = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, subU, 8, fdtbl_UV, DCU, UVDC_HT, UVAC_HT);
                  DCV = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, subV, 8, fdtbl_UV, DCV, UVDC_HT, UVAC_HT);
               }
            }
         }
      } else {
         for(y = 0; y < height; y += 8) {
            for(x = 0; x < width; x += 8) {
               float Y[64], U[64], V[64];
               for(row = y, pos = 0; row < y+8; ++row) {
                  // row >= height => use last input row
                  int clamped_row = (row < height) ? row : height - 1;
                  int base_p = (stbi__flip_vertically_on_write ? (height-1-clamped_row) : clamped_row)*width*comp;
                  for(col = x; col < x+8; ++col, ++pos) {
                     // if col >= width => use pixel from last input column
                     int p = base_p + ((col < width) ? col : (width-1))*comp;
                     float r = dataR[p], g = dataG[p], b = dataB[p];
                     Y[pos]= +0.29900f*r + 0.58700f*g + 0.11400f*b - 128;
                     U[pos]= -0.16874f*r - 0.33126f*g + 0.50000f*b;
                     V[pos]= +0.50000f*r - 0.41869f*g - 0.08131f*b;
                  }
               }

               DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y, 8, fdtbl_Y,  DCY, YDC_HT, YAC_HT);
               DCU = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, U, 8, fdtbl_UV, DCU, UVDC_HT, UVAC_HT);
               DCV = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, V, 8, fdtbl_UV, DCV, UVDC_HT, UVAC_HT);
            }
         }
      }

      // Do the bit alignment of the EOI marker
      stbiw__jpg_writeBits(s, &bitBuf, &bitCnt, fillBits);
   }

   // EOI
   stbiw__putc(s, 0xFF);
   stbiw__putc(s, 0xD9);

   return 1;
}

STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const void *data, int quality)
{
   stbi__write_context s = { 0 };
   stbi__start_write_callbacks(&s, func, context);
   return stbi_write_jpg_core(&s, x, y, comp, (void *) data, quality);
}


#ifndef STBI_WRITE_NO_STDIO
STBIWDEF int stbi_write_jpg(char const *filename, int x, int y, int comp, const void *data, int quality)
{
   stbi__write_context s = { 0 };
   if (stbi__start_write_file(&s,filename)) {
      int r = stbi_write_jpg_core(&s, x, y, comp, data, quality);
      stbi__end_write_file(&s);
      return r;
   } else
      return 0;
}
#endif

#endif // STB_IMAGE_WRITE_IMPLEMENTATION

/* Revision history
      1.16  (2021-07-11)
             make Deflate code emit uncompressed blocks when it would otherwise expand
             support writing BMPs with alpha channel
      1.15  (2020-07-13) unknown
      1.14  (2020-02-02) updated JPEG writer to downsample chroma channels
      1.13
      1.12
      1.11  (2019-08-11)

      1.10  (2019-02-07)
             support utf8 filenames in Windows; fix warnings and platform ifdefs
      1.09  (2018-02-11)
             fix typo in zlib quality API, improve STB_I_W_STATIC in C++
      1.08  (2018-01-29)
             add stbi__flip_vertically_on_write, external zlib, zlib quality, choose PNG filter
      1.07  (2017-07-24)
             doc fix
      1.06 (2017-07-23)
             writing JPEG (using Jon Olick's code)
      1.05   ???
      1.04 (2017-03-03)
             monochrome BMP expansion
      1.03   ???
      1.02 (2016-04-02)
             avoid allocating large structures on the stack
      1.01 (2016-01-16)
             STBIW_REALLOC_SIZED: support allocators with no realloc support
             avoid race-condition in crc initialization
             minor compile issues
      1.00 (2015-09-14)
             installable file IO function
      0.99 (2015-09-13)
             warning fixes; TGA rle support
      0.98 (2015-04-08)
             added STBIW_MALLOC, STBIW_ASSERT etc
      0.97 (2015-01-18)
             fixed HDR asserts, rewrote HDR rle logic
      0.96 (2015-01-17)
             add HDR output
             fix monochrome BMP
      0.95 (2014-08-17)
             add monochrome TGA output
      0.94 (2014-05-31)
             rename private functions to avoid conflicts with stb_image.h
      0.93 (2014-05-27)
             warning fixes
      0.92 (2010-08-01)
             casts to unsigned char to fix warnings
      0.91 (2010-07-17)
             first public release
      0.90   first internal release
*/

/*
------------------------------------------------------------------------------
This software is available under 2 licenses -- choose whichever you prefer.
------------------------------------------------------------------------------
ALTERNATIVE A - MIT License
Copyright (c) 2017 Sean Barrett
Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
------------------------------------------------------------------------------
ALTERNATIVE B - Public Domain (www.unlicense.org)
This is free and unencumbered software released into the public domain.
Anyone is free to copy, modify, publish, use, compile, sell, or distribute this
software, either in source code form or as a compiled binary, for any purpose,
commercial or non-commercial, and by any means.
In jurisdictions that recognize copyright laws, the author or authors of this
software dedicate any and all copyright interest in the software to the public
domain. We make this dedication for the benefit of the public at large and to
the detriment of our heirs and successors. We intend this dedication to be an
overt act of relinquishment in perpetuity of all present and future rights to
this software under copyright law.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
------------------------------------------------------------------------------
*/
```

9.48 Include/Volume.h File Reference

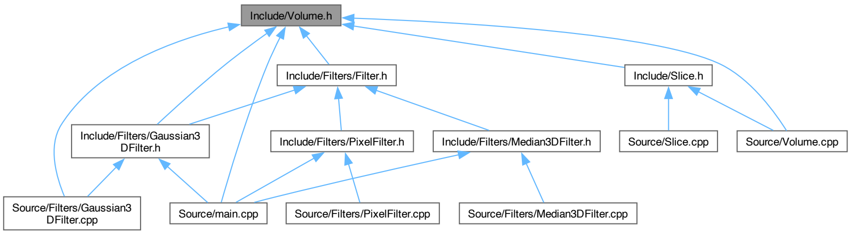
Manages 3D volumetric data for processing and analysis.

```
#include <vector>
#include <optional>
#include <variant>
#include <string>
```

Include dependency graph for Volume.h:



This graph shows which files directly or indirectly include this file:


Include/Volume.h
Include/Filters/Filter.h
Include/Slice.h
Include/Filters/Gaussian3
DFilter.h
Include/Filters/PixelFilter.h
Include/Filters/Median3DFilter.h
Source/Slice.cpp
Source/Volume.cpp
Source/Filters/Gaussian3
DFilter.cpp
Source/main.cpp
Source/Filters/PixelFilter.cpp
Source/Filters/Median3DFilter.cpp

Classes

- class Volume

Macros

- #define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H

9.48.1 Detailed Description

Manages 3D volumetric data for processing and analysis.

The Volume class is designed to encapsulate 3D volumetric data, providing functionalities for loading, saving, and manipulating this data. It supports loading volume data from individual files or an entire directory and saving processed data with flexible options. This class plays a crucial role in applications involving 3D data visualization, medical imaging analysis, and scientific research, where handling and analyzing volumetric data efficiently is essential. By preventing copy construction and assignment, it ensures that volume data management is both safe and efficient. The Volume class is a foundational component of the data manipulation toolkit developed by the Advanced Programming Group, aimed at facilitating sophisticated 3D data processing and analysis tasks.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.48.2 Macro Definition Documentation

9.48.2.1 ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H

```
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H
```

9.49 Volume.h

Go to the documentation of this file.

```

#pragma once

#ifndef ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H
#define ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H

#include <vector>
#include <optional>
#include <variant>
#include <string>

class Volume {
private:
    int width, height, depth; // Size of the volume
    unsigned char *data; // Volume data

    Volume(const Volume &) = delete;

    Volume &operator=(const Volume &) = delete;

public:
    Volume();

    Volume(int width, int height, int depth);

    Volume(int width, int height, int depth, unsigned char *data);

    ~Volume();

    int getWidth() const;

    int getHeight() const;

    int getDepth() const;

    unsigned char *getData() const;

    unsigned char getVoxel(int x, int y, int z) const;

    void setWidth(int width);

    void setHeight(int height);

    void setDepth(int depth);

    void updateData(const std::vector<unsigned char> &newData);

    bool loadFromFiles(const std::vector<std::string> &paths);

    bool loadFromDirectory(const std::string &directoryPath);

    void save(const std::string &path, const std::string &plane) const;

    void save(const std::string &path, const std::string &plane, int sliceIndex) const;

    void save(const std::string &path, const std::string &plane, std::string projector) const;

    void
    save(const std::string &path, const std::string &plane, const std::string &projector, int begin,
int end) const;

};


#endif //ADVANCED_PROGRAMMING_GROUP_RADIX_SORT_VOLUME_H
```

9.50 REFERENCE.md File Reference

9.51 Output/README.md File Reference

9.52 README.md File Reference

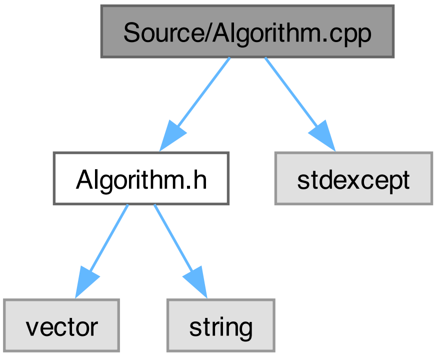
9.53 Scans/README.md File Reference

9.54 Source/Algorithm.cpp File Reference

Provides implementations of sorting and selection algorithms.

```
#include "Algorithm.h"
#include <stdexcept>
```

Include dependency graph for Algorithm.cpp:



9.54.1 Detailed Description

Provides implementations of sorting and selection algorithms.

This source file contains the implementation of several algorithms including Quickselect and Quicksort, tailored for both numerical data and strings. The Quickselect algorithm is designed to efficiently find the k-th smallest element in an unordered list, while the Quicksort algorithm sorts elements of an array or a vector in ascending order. Both algorithms employ a partitioning approach to organize data around a pivot value, facilitating rapid sorting and retrieval operations. These algorithms are optimized for performance and flexibility, making them suitable for a wide range of applications requiring efficient data organization and retrieval.

Date

Created on March 19, 2024

Authors

Advanced Programming Group Radix Sort:

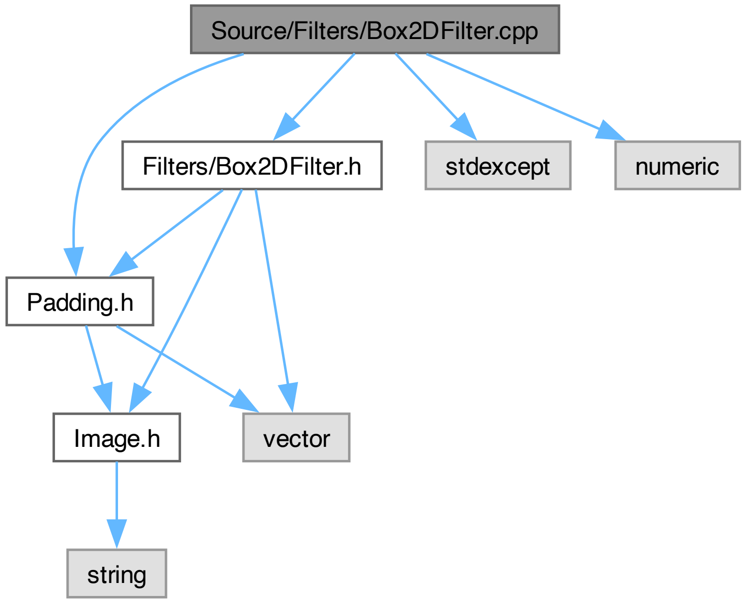
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.55 Source/Filters/Box2DFilter.cpp File Reference

Implementation of the Box2DFilter class for image processing.

```
#include "Filters/Box2DFilter.h"
#include "Filters/Padding.h"
#include <stdexcept>
#include <numeric>
```

Include dependency graph for Box2DFilter.cpp:



9.55.1 Detailed Description

Implementation of the Box2DFilter class for image processing.

This file provides the implementation of the Box2DFilter class. The Box2DFilter applies a simple yet effective 2D box filtering operation to images. It is designed to perform spatial averaging, which can be particularly useful for blurring or smoothing images. The implementation supports custom kernel sizes (must be odd) and handles edges through various padding strategies defined in the PaddingType enum. A noteworthy feature planned for future implementation is the use of the integral image technique to significantly speed up the computation. This contribution is part of the tools developed by the Advanced Programming Group for advanced image manipulation and processing.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

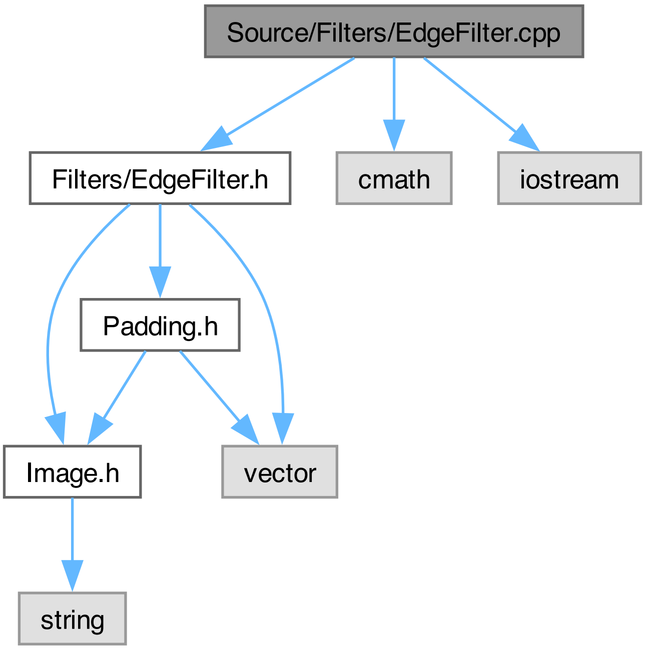
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.56 Source/Filters/EdgeFilter.cpp File Reference

Implementation of the EdgeFilter class for edge detection in images.

```
#include "Filters/EdgeFilter.h"
#include <cmath>
#include <iostream>
```

Include dependency graph for EdgeFilter.cpp:



9.56.1 Detailed Description

Implementation of the EdgeFilter class for edge detection in images.

This file defines the methods of the EdgeFilter class, including various edge detection algorithms such as Sobel, Prewitt, Scharr, and Roberts. Edge detection is performed on grayscale images, where the input image is checked for its color channels to ensure it's suitable for edge detection. Each edge detection method applies a specific kernel to highlight edges in the image by calculating the gradient magnitude at each pixel. This implementation allows for flexible edge detection through the choice of algorithm and padding method, catering to diverse image processing needs. The EdgeFilter class is part of the Advanced Programming Group's efforts to provide robust tools for image analysis and manipulation.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

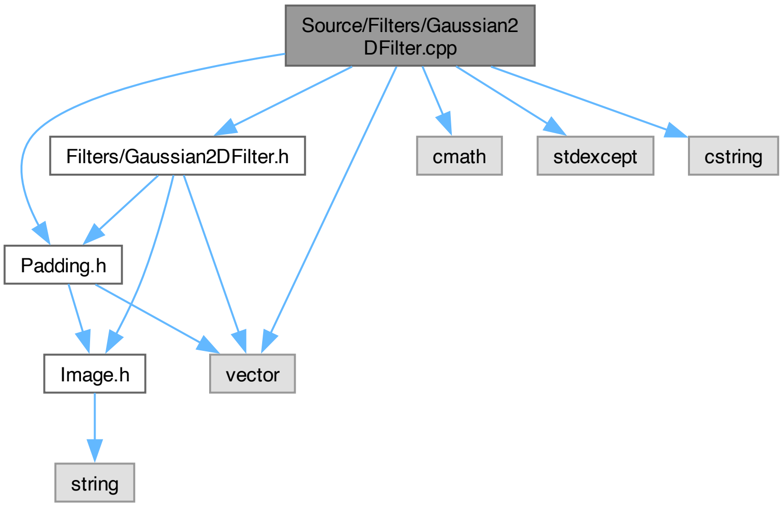
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.57 Source/Filters/Gaussian2DFilter.cpp File Reference

Implementation of the Gaussian2DFilter class for applying a Gaussian blur to images.

```
#include "Filters/Gaussian2DFilter.h"
#include "Filters/Padding.h"
#include <cmath>
#include <vector>
#include <stdexcept>
#include <cstring>
```

Include dependency graph for Gaussian2DFilter.cpp:


Source/Filters/Gaussian2DFilter.cpp
Filters/Gaussian2DFilter.h
cmath
stdexcept
cstring
Padding.h
Image.h
vector
string

9.57.1 Detailed Description

Implementation of the Gaussian2DFilter class for applying a Gaussian blur to images.

This file contains the implementation of the Gaussian2DFilter class, which is designed to apply a Gaussian blur to images. The Gaussian blur is performed by convolving the image with a Gaussian kernel. The class allows for custom kernel sizes and sigma values, providing flexibility in the strength and extent of the blur effect. The Gaussian kernel is generated dynamically based on the provided sigma and kernel size, ensuring that the kernel is properly normalized. The class also supports different padding types to handle image borders. This implementation is part of the Advanced Programming Group's efforts to develop comprehensive tools for image manipulation and processing, enhancing image quality and preparing images for further analysis or display.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

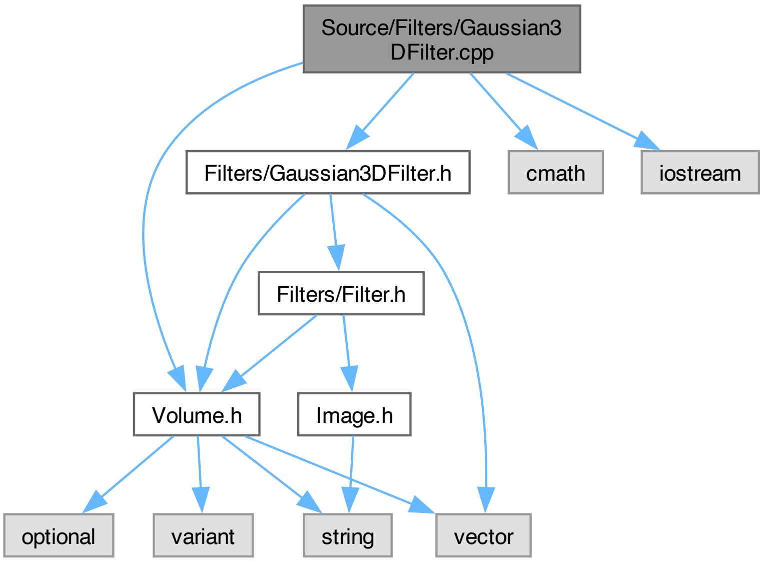
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.58 Source/Filters/Gaussian3DFilter.cpp File Reference

Implements a 3D Gaussian filter for volumetric data smoothing.

```
#include "Filters/Gaussian3DFilter.h"
#include "Volume.h"
#include <cmath>
#include <iostream>
```

Include dependency graph for Gaussian3DFilter.cpp:



9.58.1 Detailed Description

Implements a 3D Gaussian filter for volumetric data smoothing.

The Gaussian3DFilter class applies a Gaussian smoothing operation to 3D volume data. Designed to reduce noise and smooth transitions without significantly blurring the edges, this filter replaces each voxel's value with a weighted average of its neighbors' values, where the weights are determined by a Gaussian distribution. The class allows for customization of the standard deviation (sigma) and the kernel size, enabling fine control over the extent of smoothing. Efficient convolution operations along each axis (X, Y, and Z) ensure that the filter is applied thoroughly across the entire volume. This class is essential for preprocessing in applications such as medical imaging, where enhancing the clarity of features within volumetric data is crucial.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

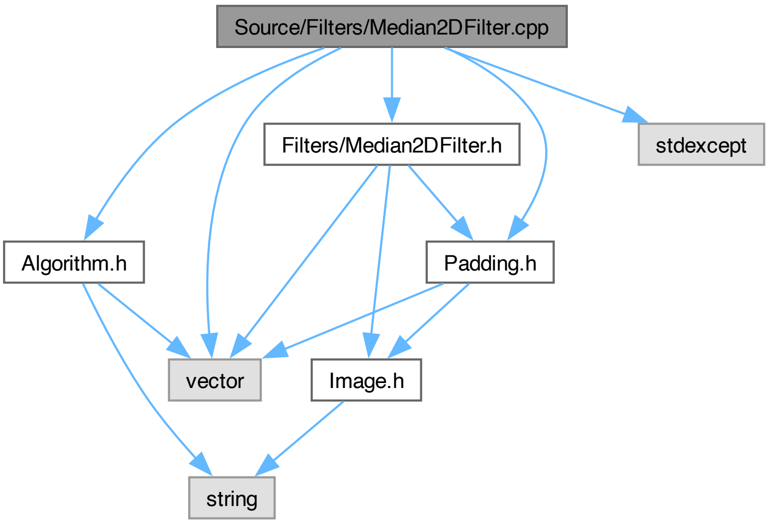
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.59 Source/Filters/Median2DFilter.cpp File Reference

Implementation of the Median2DFilter class for applying median filtering to images.

```
#include "Filters/Median2DFilter.h"
#include "Filters/Padding.h"
#include "Algorithm.h"
#include <vector>
#include <stdexcept>
```

Include dependency graph for Median2DFilter.cpp:



9.59.1 Detailed Description

Implementation of the Median2DFilter class for applying median filtering to images.

This file contains the implementation of the Median2DFilter class, which applies a median filter to images for noise reduction. Median filtering is a non-linear process useful in reducing salt-and-pepper noise while preserving edges in the image. This class supports custom kernel sizes and incorporates various padding strategies to handle image borders effectively. The median value is computed using a quick select algorithm to find the middle value in the intensity distribution of the pixels within the kernel window. This approach ensures that the filtering process is both efficient and effective, making it suitable for real-time image processing applications. Part of the tools developed by the Advanced Programming Group, this implementation aims to provide a robust solution for enhancing image quality.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

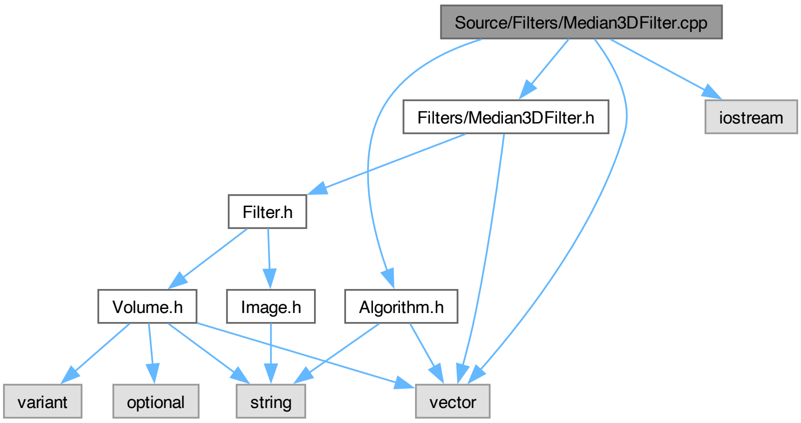
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.60 Source/Filters/Median3DFilter.cpp File Reference

Implements a 3D median filter for volumetric data noise reduction.

```
#include "Filters/Median3DFilter.h"
#include "Algorithm.h"
#include <iostream>
#include <vector>
```

Include dependency graph for Median3DFilter.cpp:



9.60.1 Detailed Description

Implements a 3D median filter for volumetric data noise reduction.

The Median3DFilter class applies a median filtering operation to 3D volume data, aiming to reduce noise while preserving edges. This filter replaces each voxel's value with the median value within a specified neighborhood around that voxel, effectively smoothing the volume data and enhancing the visibility of structural details. The class supports customizable kernel sizes and efficiently computes the median values using a quick select algorithm to handle large datasets. This approach is particularly beneficial in applications like medical imaging and scientific visualization, where maintaining the integrity of structural boundaries in the presence of noise is critical. The Median3DFilter is an essential component of the volumetric data processing toolkit developed by the Advanced Programming Group.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

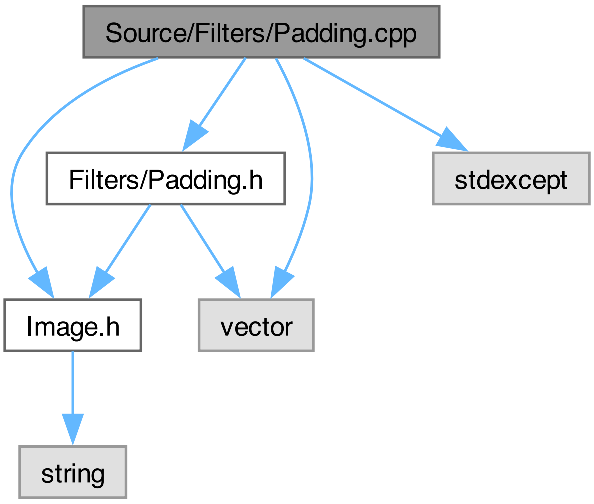
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.61 Source/Filters/Padding.cpp File Reference

Implementation of padding strategies for image processing.

```
#include "Filters/Padding.h"
#include "Image.h"
#include <stdexcept>
#include <vector>
```

Include dependency graph for Padding.cpp:



9.61.1 Detailed Description

Implementation of padding strategies for image processing.

This file contains the implementation of various padding strategies used in image processing operations, particularly in the context of applying filters such as median, Gaussian, and edge detection filters. The `getPixelWindow` function provided in this file facilitates the retrieval of pixel values from a specified window around a target pixel, applying padding as necessary according to the chosen padding strategy. Supported padding types include zero padding, edge replication, and reflect padding, each suitable for different image processing needs. This functionality is essential for handling image borders when applying filters that require neighborhood information. Part of the tools developed by the Advanced Programming Group, this implementation aims to enhance the flexibility and effectiveness of image manipulation tasks.

Date

Created on March 21, 2024

Authors

Advanced Programming Group Radix Sort:

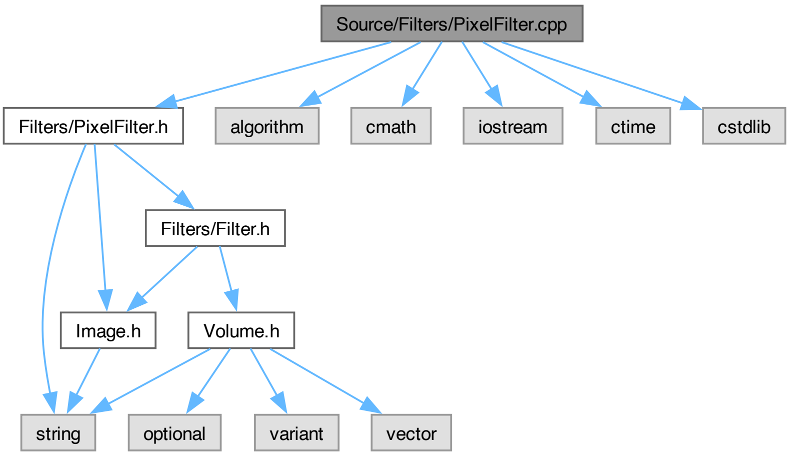
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.62 Source/Filters/PixelFilter.cpp File Reference

Implementation of the PixelFilter class for various pixel-level image filtering operations.

```
#include "Filters/PixelFilter.h"
#include <algorithm>
#include <cmath>
#include <iostream>
#include <ctime>
#include <cstdlib>
```

Include dependency graph for PixelFilter.cpp:



9.62.1 Detailed Description

Implementation of the PixelFilter class for various pixel-level image filtering operations.

This file defines the implementation of the PixelFilter class, which supports a range of image processing operations including grayscale conversion, brightness adjustment, histogram equalization, thresholding, and the addition of salt-and-pepper noise. These operations can be applied to images in different color spaces such as RGB, HSL, and HSV, depending on the filter type. The class is designed to be flexible, allowing for optional parameters and ensuring that input values are within expected ranges for each filter type. Part of the Advanced Programming Group's toolkit, this implementation aims to provide a comprehensive solution for image enhancement and manipulation, facilitating both basic and advanced image processing tasks.

Date

Created on March 20, 2024

Authors

Advanced Programming Group Radix Sort:

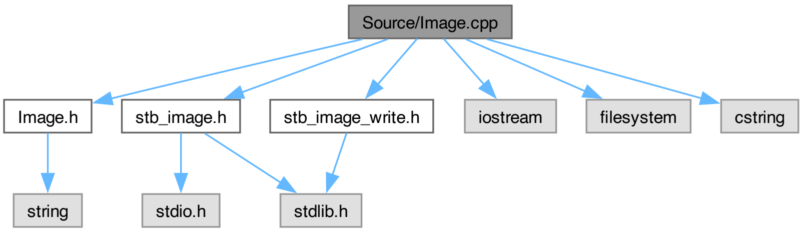
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.63 Source/Image.cpp File Reference

Implements basic image processing functionalities.

```
#include "Image.h"
#include "stb_image.h"
#include "stb_image_write.h"
#include <iostream>
#include <filesystem>
#include <cstring>
```

Include dependency graph for Image.cpp:



Macros

- #define STB_IMAGE_WRITE_IMPLEMENTATION
- #define STB_IMAGE_IMPLEMENTATION

9.63.1 Detailed Description

Implements basic image processing functionalities.

The Image class provides fundamental functionalities for image processing, including loading, saving, and converting images to grayscale. Leveraging the stb_image and stb_image_write libraries, this class supports a wide range of image formats for both input and output operations. The convertToGrayscale method utilizes the luminance method to transform color images into grayscale, reflecting human perception of color brightness. This class is designed to serve as a foundation for more complex image processing tasks, offering efficient and straightforward manipulation of image data.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.63.2 Macro Definition Documentation

9.63.2.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

9.63.2.2 STB_IMAGE_WRITE_IMPLEMENTATION

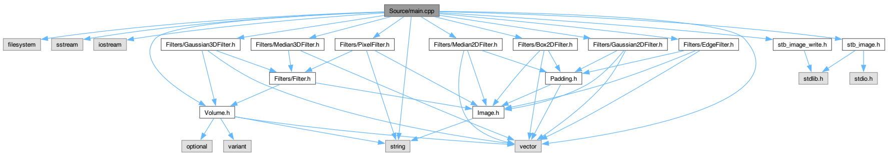
```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

9.64 Source/main.cpp File Reference

Main entry point for the Data Processing Program.

```
#include <filesystem>
#include <sstream>
#include <iostream>
#include <vector>
#include <string>
#include "Volume.h"
#include "Filters/Gaussian3DFilter.h"
#include "Filters/Median3DFilter.h"
#include "Filters/PixelFilter.h"
#include "Filters/Median2DFilter.h"
#include "Filters/Box2DFilter.h"
#include "Filters/Gaussian2DFilter.h"
#include "Filters/EdgeFilter.h"
#include "stb_image_write.h"
#include "stb_image.h"
```

Include dependency graph for main.cpp:



Macros

- #define STB_IMAGE_WRITE_IMPLEMENTATION
- #define STB_IMAGE_IMPLEMENTATION

Functions

- int main ()

9.64.1 Detailed Description

Main entry point for the Data Processing Program.

This program provides an interface for processing both 2D and 3D image data through a variety of filters. Users can choose to process 2D images with operations such as brightness adjustment, grayscale conversion, histogram equalization, thresholding, and adding salt-and-pepper noise. For 3D volume data, the program offers Gaussian and Median filtering to enhance or manipulate the data. The program leverages libraries such as STB for image reading and writing, and utilizes the filesystem library for directory operations. Users interact with the program via the console, selecting options to load, process, and save image data in various formats. This main file orchestrates the flow of operations based on user input, leveraging classes designed for specific image processing tasks.

Date

Created on 20/03/2024.

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.64.2 Macro Definition Documentation

9.64.2.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

9.64.2.2 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

9.64.3 Function Documentation

9.64.3.1 main()

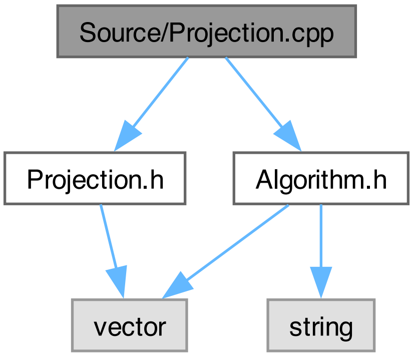
```
int main ( )
```

9.65 Source/Projection.cpp File Reference

Implements projection techniques for visualizing 3D volumetric data.

```
#include "Projection.h"
#include "Algorithm.h"
```

Include dependency graph for Projection.cpp:



9.65.1 Detailed Description

Implements projection techniques for visualizing 3D volumetric data.

The Projection class offers several methods to transform 3D volume data into 2D images through different types of intensity projections: Maximum Intensity Projection (MIP), Minimum Intensity Projection (MinIP), Average Intensity Projection (AIP), and Median Intensity Projection (MedIP). These methods facilitate the visualization of volumetric data by projecting certain statistical characteristics of the data along a chosen axis, typically the depth (z-axis), onto a 2D plane. Each projection technique highlights different aspects of the volume data, making them invaluable tools for analysis and interpretation in fields such as medical imaging, scientific visualization, and data analysis. The implementation is optimized to efficiently process large volumes of data, rendering meaningful 2D representations that encapsulate the essence of the 3D volume.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

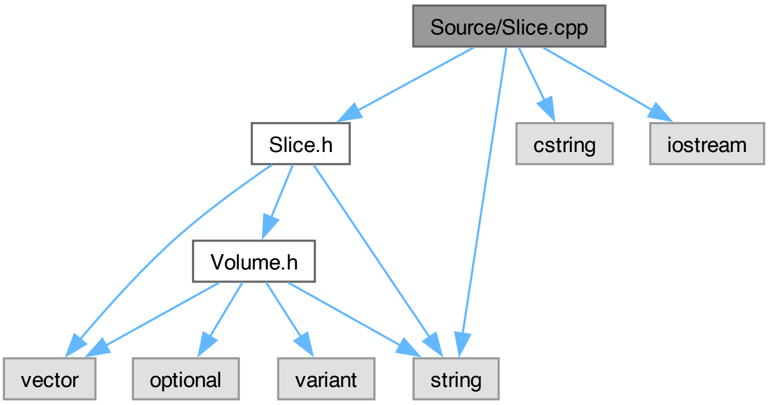
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.66 Source/Slice.cpp File Reference

Implements functionality to extract 2D slices from a 3D volume.

```
#include "Slice.h"
#include <cstring>
#include <iostream>
#include <string>
```

Include dependency graph for Slice.cpp:



9.66.1 Detailed Description

Implements functionality to extract 2D slices from a 3D volume.

The Slice class provides a method for extracting specific 2D slices from a 3D volumetric dataset, based on given dimensions, raw data, and the desired orientation. The class supports extraction along the 'x-y', 'x-z', and 'y-z' planes, accommodating a range of visualization and analysis needs. This flexibility allows users to examine the internal structure of the volume from different perspectives, facilitating a deeper understanding of the data. The method validates the requested slice index and plane orientation, ensuring the integrity of the returned slice. It serves as a crucial tool in domains where 3D volumetric data is prevalent, such as medical imaging, geospatial analysis, and scientific research, by simplifying the process of volumetric data exploration.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

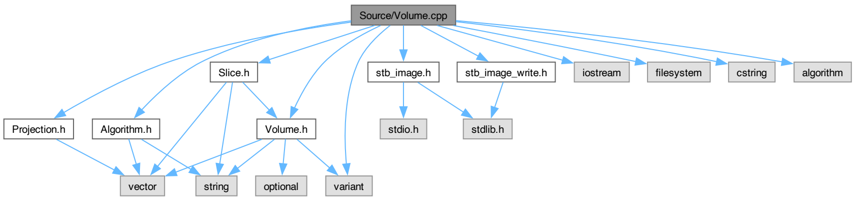
- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.67 Source/Volume.cpp File Reference

Manages and manipulates 3D volumetric data.

```
#include "Volume.h"
#include "Projection.h"
#include "Slice.h"
#include "Algorithm.h"
#include "stb_image.h"
#include "stb_image_write.h"
#include <iostream>
#include <filesystem>
#include <cstring>
#include <variant>
#include <algorithm>
```

Include dependency graph for Volume.cpp:



Macros

- #define STB_IMAGE_IMPLEMENTATION
- #define STB_IMAGE_WRITE_IMPLEMENTATION

9.67.1 Detailed Description

Manages and manipulates 3D volumetric data.

The Volume class encapsulates 3D volumetric data, providing functionalities to load data from multiple image files or a directory, update internal data, and save specific slices or projections to files. It supports extracting slices along 'x-y', 'x-z', or 'y-z' planes and computing various intensity projections such as Maximum, Minimum, Average, and Median Intensity Projections. This functionality is essential for analyzing 3D data in medical imaging, scientific visualization, and other fields requiring volumetric data manipulation. The class ensures efficient handling of large datasets and integrates with external libraries like stb_image for reading and writing image files, making it a versatile tool for 3D data processing.

Date

Created on March 18, 2024

Authors

Advanced Programming Group Radix Sort:

- Benjamin Duncan (edsml-bd1023)
- Boyang Hu (edsml-bh223)
- Chawk Chamoun (edsml-cc8915)
- Mingsheng Cai (acse-sc4623)
- Moyu Zhang (acse-mz223)
- Ryan Benney (acse-rgb123)

9.67.2 Macro Definition Documentation

9.67.2.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

9.67.2.2 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

Index