Test Markdown File

Showing heradoc's Features

Me
Supervisor: My Supervisor
Advisor: My Advisor

2021-02-13
My Publisher

1 image

1.1 Crate pub image

- image::error
- image::buffer
- image::math
- image::imageops
- image::io
- image::flat
- image::codecs
- image::bmp
- image::dds
- image::dxt
- image::farbfeld
- image::gif
- image::hdr
- image::ico
- image::jpeg
- image::png
- image::pnm
- image::tga
- image::tiff
- image::webp
- image::ColorType
- image::ExtendedColorType
- image::Luma
- image::LumaA
- image::Rgb
- image::Rgba
- image::Bgr
- image::Bgra
-
-
- image::AnimationDecoder

- image::GenericImage
- image::GenericImageView
- image::ImageDecoder
- image::ImageDecoderExt
- image::ImageEncoder
- image::ImageFormat
- image::ImageOutputFormat
- image::Progress
- image::Pixels
- image::SubImage
- image::GrayAlphaImage
- image::GrayImage
- image::ImageBuffer
- image::RgbImage
- image::RgbaImage
-
- image::EncodableLayout
- image::Primitive
- image::Pixel
- image::guess_format
- image::load
- image::load_from_memory
- image::load_from_memory_with_format
- image::open
- image::save_buffer
- image::save_buffer_with_format
- image::image_dimensions
- image::DynamicImage
- image::Delay
- image::Frame
- image::Frames

1.2 Module pub image::error

- image::error::ImageError
- image::error::UnsupportedError
- image::error::UnsupportedErrorKind
- image::error::EncodingError
- image::error::ParameterError
- image::error::ParameterErrorKind
- image::error::DecodingError
- image::error::LimitError
- image::error::LimitErrorKind
- image::error::ImageFormatHint
- image::error::ImageResult

1.3 Enum pub image::error::ImageError

```
pub enum image::error::ImageError {
    Decoding,
    Encoding,
    Parameter,
    Limits,
    Unsupported,
    IoError,
}
```

The generic error type for image operations.

This high level enum allows, by variant matching, a rough separation of concerns between underlying IO, the caller, format specifications, and the `image` implementation.

Variants

`Decoding` An error was encountered while decoding.

This means that the input data did not conform to the specification of some image format, or that no format could be determined, or that it did not match format specific requirements set by the caller.

`Encoding` An error was encountered while encoding.

The input image can not be encoded with the chosen format, for example because the specification has no representation for its color space or because a necessary conversion is ambiguous. In some cases it might also happen that the dimensions can not be used with the format.

`Parameter` An error was encountered in input arguments.

This is a catch-all case for strictly internal operations such as scaling, conversions, etc. that involve no external format specifications.

`Limits` Completing the operation would have required more resources than allowed.

Errors of this type are limits set by the user or environment, *not* inherent in a specific format or operation that was executed.

`Unsupported` An operation can not be completed by the chosen abstraction.

This means that it might be possible for the operation to succeed in general but

- it requires a disabled feature,
- the implementation does not yet exist, or
- no abstraction for a lower level could be found.

`IoError` An error occurred while interacting with the environment.

Implementations

`impl Send for ImageError`

`impl Sync for ImageError`

`impl Unpin for ImageError`

`impl !UnwindSafe for ImageError`

`impl !RefUnwindSafe for ImageError`

impl From<Error> for ImageError

fn from(err: io::Error) -> ImageError

impl From<Error> for ImageError

fn from(error: Error) -> ImageError

impl Debug for ImageError

fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Display for ImageError

fn fmt(self: &Self, fmt: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>

impl Error for ImageError

fn source(self: &Self) -> Option<&Error>

1.4 Struct pub image::error::UnsupportedError

```rust
pub struct UnsupportedError{
    // some fields omitted
}
```

The implementation for an operation was not provided.

See the variant enum.ImageError.html#variant.Unsupported for more documentation.

Implementations

impl UnsupportedError

pub fn from_format_and_kind(format: ImageFormatHint, kind: UnsupportedErrorKind) -> Self

Create an 'UnsupportedError' for an image with details on the unsupported feature.

If the operation was not connected to a particular image format then the hint may be 'Unknown'.

pub fn kind(self: &Self) -> UnsupportedErrorKind

Returns the corresponding 'UnsupportedErrorKind' of the error.

pub fn format_hint(self: &Self) -> ImageFormatHint

Returns the image format associated with this error.

impl Send for UnsupportedError

impl Sync for UnsupportedError

`impl Unpin for UnsupportedError`

`impl UnwindSafe for UnsupportedError`

`impl RefUnwindSafe for UnsupportedError`

`impl From<ImageFormatHint> for UnsupportedError`

`fn from(hint: ImageFormatHint) -> Self`

`impl Debug for UnsupportedError`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl Display for UnsupportedError`

`fn fmt(self: &Self, fmt: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>`

`impl Error for UnsupportedError`

1.5 Enum pub image::error::UnsupportedErrorKind

```rust
pub enum image::error::UnsupportedErrorKind {
    Color,
    Format,
    GenericFeature,
    // some variants omitted
}
```

Details what feature is not supported.

Variants

`Color` The required color type can not be handled.

`Format` An image format is not supported.

`GenericFeature` Some feature specified by string. This is discouraged and is likely to get deprecated (but not removed).

Implementations

`impl Send for UnsupportedErrorKind`

`impl Sync for UnsupportedErrorKind`

`impl Unpin for UnsupportedErrorKind`

`impl UnwindSafe for UnsupportedErrorKind`

`impl RefUnwindSafe for UnsupportedErrorKind`

`impl Clone for UnsupportedErrorKind`

```
 fn clone(self:  &Self) -> UnsupportedErrorKind
```

```
impl PartialEq<UnsupportedErrorKind> for UnsupportedErrorKind
```

```
 fn eq(self:  &Self, other:  &UnsupportedErrorKind) -> bool
```

```
 fn ne(self:  &Self, other:  &UnsupportedErrorKind) -> bool
```

```
impl Debug for UnsupportedErrorKind
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Hash for UnsupportedErrorKind
```

```
 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()
```

```
impl StructuralPartialEq for UnsupportedErrorKind
```

1.6 Struct pub image::error::EncodingError

```
pub struct EncodingError{
    // some fields omitted
}
```

An error was encountered while encoding an image.

This is used as an opaque representation for the `enum.ImageError.html#variant.Encoding` variant. See its documentation for more information.

Implementations

```
impl EncodingError
```

```
 pub fn new<impl Into<Box<dyn Error + Send + Sync»:  Into<Box<Error»>(format:  ImageFormatHint,
err:  impl Into<Box<Error») -> Self
```

Create an ‘EncodingError‘ that stems from an arbitrary error of an underlying encoder.

```
 pub fn from_format_hint(format:  ImageFormatHint) -> Self
```

Create an ‘EncodingError‘ for an image format.

The error will not contain any further information but is very easy to create.

```
 pub fn format_hint(self:  &Self) -> ImageFormatHint
```

Return the image format associated with this error.

```
impl Send for EncodingError
```

```
impl Sync for EncodingError
```

```
impl Unpin for EncodingError
impl !UnwindSafe for EncodingError
```

```
impl !RefUnwindSafe for EncodingError
```

```
impl Debug for EncodingError
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Display for EncodingError
```

```
 fn fmt(self:  &Self, fmt:  &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>
```

```
impl Error for EncodingError
```

```
 fn source(self:  &Self) -> Option<&Error>
```

1.7 Struct pub image::error::ParameterError

```
pub struct ParameterError{
    // some fields omitted
}
```

An error was encountered in inputs arguments.

This is used as an opaque representation for the `enum.ImageError.html#variant.Parameter` variant. See its documentation for more information.

Implementations

```
impl ParameterError
```

```
 pub fn from_kind(kind:  ParameterErrorKind) -> Self
```

Construct a 'ParameterError' directly from a corresponding kind.

```
 pub fn kind(self:  &Self) -> ParameterErrorKind
```

Returns the corresponding 'ParameterErrorKind' of the error.

```
impl Send for ParameterError
```

```
impl Sync for ParameterError
```

```
impl Unpin for ParameterError
```

```
impl !UnwindSafe for ParameterError
```

```
impl !RefUnwindSafe for ParameterError
```

```
impl Debug for ParameterError
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

`impl Display for ParameterError`

`fn fmt(self: &Self, fmt: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>`

`impl Error for ParameterError`

`fn source(self: &Self) -> Option<&Error>`

1.8 Enum pub image::error::ParameterErrorKind

```
pub enum image::error::ParameterErrorKind {
    DimensionMismatch,
    FailedAlready,
    Generic,
    NoMoreData,
    // some variants omitted
}
```

Details how a parameter is malformed.

Variants

`DimensionMismatch` The dimensions passed are wrong.

`FailedAlready` Repeated an operation for which error that could not be cloned was emitted already.

`Generic` A string describing the parameter. This is discouraged and is likely to get deprecated (but not removed).

`NoMoreData` The end of the image has been reached.

Implementations

`impl Send for ParameterErrorKind`

`impl Sync for ParameterErrorKind`

`impl Unpin for ParameterErrorKind`

`impl UnwindSafe for ParameterErrorKind`

`impl RefUnwindSafe for ParameterErrorKind`

`impl Clone for ParameterErrorKind`

`fn clone(self: &Self) -> ParameterErrorKind`

`impl PartialEq<ParameterErrorKind> for ParameterErrorKind`

`fn eq(self: &Self, other: &ParameterErrorKind) -> bool`

`fn ne(self: &Self, other: &ParameterErrorKind) -> bool`

`impl Debug for ParameterErrorKind`

fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Hash for ParameterErrorKind

fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl StructuralPartialEq for ParameterErrorKind

1.9 Struct pub image::error::DecodingError

```rust
pub struct DecodingError{
    // some fields omitted
}
```

An error was encountered while decoding an image.

This is used as an opaque representation for the `enum.ImageError.html#variant.Decoding` variant. See its documentation for more information.

Implementations

impl DecodingError

pub fn new<impl Into<Box<dyn Error + Send + Sync»: Into<Box<Error»>(format: ImageFormatHint, err: impl Into<Box<Error») -> Self

Create a 'DecodingError' that stems from an arbitrary error of an underlying decoder.

pub fn from_format_hint(format: ImageFormatHint) -> Self

Create a 'DecodingError' for an image format.

The error will not contain any further information but is very easy to create.

pub fn format_hint(self: &Self) -> ImageFormatHint

Returns the image format associated with this error.

impl Send for DecodingError

impl Sync for DecodingError

impl Unpin for DecodingError

impl !UnwindSafe for DecodingError

impl !RefUnwindSafe for DecodingError

impl Debug for DecodingError

fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Display for DecodingError

```
fn fmt(self: &Self, fmt: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>
```

```
impl Error for DecodingError
```

```
fn source(self: &Self) -> Option<&Error>
```

1.10 Struct pub image::error::LimitError

```
pub struct LimitError{
    // some fields omitted
}
```

Completing the operation would have required more resources than allowed.

This is used as an opaque representation for the `enum.ImageError.html#variant.Limits` variant. See its documentation for more information.

Implementations

```
impl LimitError
```

```
pub fn from_kind(kind: LimitErrorKind) -> Self
```

Construct a generic ‘LimitError‘ directly from a corresponding kind.

```
pub fn kind(self: &Self) -> LimitErrorKind
```

Returns the corresponding ‘LimitErrorKind‘ of the error.

```
impl Send for LimitError
```

```
impl Sync for LimitError
```

```
impl Unpin for LimitError
```

```
impl UnwindSafe for LimitError
```

```
impl RefUnwindSafe for LimitError
```

```
impl Debug for LimitError
```

```
fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Display for LimitError
```

```
fn fmt(self: &Self, fmt: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>
```

```
impl Error for LimitError
```

1.11 Enum pub image::error::LimitErrorKind

```rust
#[allow(missing_copy_implementations)]
pub enum image::error::LimitErrorKind {
    DimensionError,
    InsufficientMemory,
    // some variants omitted
}
```

Indicates the limit that prevented an operation from completing.

Note that this enumeration is not exhaustive and may in the future be extended to provide more detailed information or to incorporate other resources types.

Variants

`DimensionError` The resulting image exceed dimension limits in either direction.

`InsufficientMemory` The operation would have performed an allocation larger than allowed.

Implementations

`impl Send for LimitErrorKind`

`impl Sync for LimitErrorKind`

`impl Unpin for LimitErrorKind`

`impl UnwindSafe for LimitErrorKind`

`impl RefUnwindSafe for LimitErrorKind`

`impl Clone for LimitErrorKind`

`fn clone(self: &Self) -> LimitErrorKind`

`impl Eq for LimitErrorKind`

`impl PartialEq<LimitErrorKind> for LimitErrorKind`

`fn eq(self: &Self, other: &LimitErrorKind) -> bool`

`fn ne(self: &Self, other: &LimitErrorKind) -> bool`

`impl Debug for LimitErrorKind`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl Hash for LimitErrorKind`

`fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()`

`impl StructuralPartialEq for LimitErrorKind`

`impl StructuralEq for LimitErrorKind`

1.12 Enum pub image::error::ImageFormatHint

```
pub enum image::error::ImageFormatHint {
    Exact,
    Name,
    PathExtension,
    Unknown,
    // some variants omitted
}
```

A best effort representation for image formats.

Variants

`Exact` The format is known exactly.

`Name` The format can be identified by a name.

`PathExtension` A common path extension for the format is known.

`Unknown` The format is not known or could not be determined.

Implementations

`impl Send for ImageFormatHint`

`impl Sync for ImageFormatHint`

`impl Unpin for ImageFormatHint`

`impl UnwindSafe for ImageFormatHint`

`impl RefUnwindSafe for ImageFormatHint`

`impl From<ImageFormat> for ImageFormatHint`

`fn from(format: ImageFormat) -> Self`

`impl From<&'_ Path> for ImageFormatHint`

`fn from(path: &std::path::Path) -> Self`

`impl From<ImageFormatHint> for UnsupportedError`

`fn from(hint: ImageFormatHint) -> Self`

`impl Clone for ImageFormatHint`

`fn clone(self: &Self) -> ImageFormatHint`

`impl PartialEq<ImageFormatHint> for ImageFormatHint`

`fn eq(self: &Self, other: &ImageFormatHint) -> bool`

`fn ne(self: &Self, other: &ImageFormatHint) -> bool`

```
impl Debug for ImageFormatHint
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Display for ImageFormatHint
```

```
 fn fmt(self:  &Self, fmt:  &mut fmt::Formatter<'_>) -> Result<(), fmt::Error>
```

```
impl Hash for ImageFormatHint
```

```
 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()
```

```
impl StructuralPartialEq for ImageFormatHint
```

1.13 Typedef pub image::error::ImageResult

```
pub type ImageResult = Result<T, ImageError>;
```

Result of an image decoding/encoding process

1.14 Module pub image::buffer

- image::buffer::ConvertBuffer
- image::buffer::EnumeratePixels
- image::buffer::EnumeratePixelsMut
- image::buffer::EnumerateRows
- image::buffer::EnumerateRowsMut
- image::buffer::Pixels
- image::buffer::PixelsMut
- image::buffer::Rows
- image::buffer::RowsMut

1.15 Trait pub image::buffer::ConvertBuffer

```
pub trait ConvertBuffer<T> {
}
```

Provides color conversions for whole image buffers.

1.16 Struct pub image::buffer::EnumeratePixels

```
pub struct EnumeratePixels<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Enumerate the pixels of an image.

Implementations

`impl<'a, P> where <P as Pixel>::Subpixel:  Sync Send for EnumeratePixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel:  Sync Sync for EnumeratePixels<'a, P>`

`impl<'a, P> Unpin for EnumeratePixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel:  RefUnwindSafe UnwindSafe for EnumeratePixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel:  RefUnwindSafe RefUnwindSafe for EnumeratePixels<'a, P>`

`impl<'a, P: Pixel + 'a> where <P as >::Subpixel:  'a ExactSizeIterator for EnumeratePixels<'a, P>`

` fn len(self:  &Self) -> usize`

`impl<'a, P: Pixel + 'a> where <P as >::Subpixel:  'a Iterator for EnumeratePixels<'a, P>`

` type Item = (u32,u32,&'a P);`

` fn next(self:  &mut Self) -> Option<(u32,u32,&'a P)>`

`impl<P: Pixel> Clone for EnumeratePixels<'_, P>`

` fn clone(self:  &Self) -> Self`

`impl<P: Pixel> where <P as >::Subpixel:  fmt::Debug Debug for EnumeratePixels<'_, P>`

` fn fmt(self:  &Self, f:  &mut fmt::Formatter<'_>) -> fmt::Result`

1.17 Struct pub image::buffer::EnumeratePixelsMut

```
pub struct EnumeratePixelsMut<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Enumerate the pixels of an image.

Implementations

`impl<'a, P> where <P as Pixel>::Subpixel:  Send Send for EnumeratePixelsMut<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel:  Sync Sync for EnumeratePixelsMut<'a, P>`

`impl<'a, P> Unpin for EnumeratePixelsMut<'a, P>`

`impl<'a, P> !UnwindSafe for EnumeratePixelsMut<'a, P>`

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for EnumeratePixelsMut<'a
P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for EnumeratePixelsMut<'a
P>

fn len(self: &Self) -> usize

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for EnumeratePixelsMut<'a,
P>

type Item = (u32,u32,&'a mut P);

fn next(self: &mut Self) -> Option<(u32,u32,&'a mut P)>

impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for EnumeratePixelsMut<'_, P>

fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result

1.18 Struct pub image::buffer::EnumerateRows

```
pub struct EnumerateRows<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Enumerate the rows of an image.

Implementations

impl<'a, P> where <P as Pixel>::Subpixel: Sync Send for EnumerateRows<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for EnumerateRows<'a, P>

impl<'a, P> Unpin for EnumerateRows<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe UnwindSafe for EnumerateRows<'a,
P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for EnumerateRows<'a,
P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for EnumerateRows<'a,
P>

fn len(self: &Self) -> usize

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for EnumerateRows<'a, P>

type Item = (u32,EnumeratePixels<'a, P>);

fn next(self: &mut Self) -> Option<(u32,EnumeratePixels<'a, P>)>

impl<P: Pixel> Clone for EnumerateRows<'_, P>

fn clone(self: &Self) -> Self

impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for EnumerateRows<'_, P>

fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result

1.19 Struct pub image::buffer::EnumerateRowsMut

```
pub struct EnumerateRowsMut<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Enumerate the rows of an image.

Implementations

impl<'a, P> where <P as Pixel>::Subpixel: Send Send for EnumerateRowsMut<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for EnumerateRowsMut<'a, P>

impl<'a, P> Unpin for EnumerateRowsMut<'a, P>

impl<'a, P> !UnwindSafe for EnumerateRowsMut<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for EnumerateRowsMut<'a,
P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for EnumerateRowsMut<'a,
P>

fn len(self: &Self) -> usize

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for EnumerateRowsMut<'a,
P>

type Item = (u32,EnumeratePixelsMut<'a, P>);

fn next(self: &mut Self) -> Option<(u32,EnumeratePixelsMut<'a, P>)>

impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for EnumerateRowsMut<'_, P>

fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result

1.20 Struct pub image::buffer::Pixels

```
pub struct Pixels<'a, P: Pixel + 'a>
where
    <P as >::Subpixel: 'a{
    // some fields omitted
}
```

Iterate over pixel refs.

Implementations

`impl<'a, P> where <P as Pixel>::Subpixel: Sync Send for Pixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for Pixels<'a, P>`

`impl<'a, P> Unpin for Pixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe UnwindSafe for Pixels<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for Pixels<'a, P>`

`impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a DoubleEndedIterator for Pixels<'a, P>`

`fn next_back(self: &mut Self) -> Option<&'a P>`

`impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for Pixels<'a, P>`

`fn len(self: &Self) -> usize`

`impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for Pixels<'a, P>`

`type Item = &'a P;`

`fn next(self: &mut Self) -> Option<&'a P>`

`impl<P: Pixel> Clone for Pixels<'_, P>`

`fn clone(self: &Self) -> Self`

`impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for Pixels<'_, P>`

`fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result`

1.21 Struct pub image::buffer::PixelsMut

```
pub struct PixelsMut<'a, P: Pixel + 'a>
where
    <P as >::Subpixel: 'a{
    // some fields omitted
}
```

Iterate over mutable pixel refs.

Implementations

`impl<'a, P> where <P as Pixel>::Subpixel: Send Send for PixelsMut<'a, P>`

`impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for PixelsMut<'a, P>`

`impl<'a, P> Unpin for PixelsMut<'a, P>`

impl<'a, P> !UnwindSafe for PixelsMut<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for PixelsMut<'a, P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a DoubleEndedIterator for PixelsMut<'a, P>

fn next_back(self: &mut Self) -> Option<&'a mut P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for PixelsMut<'a, P>

fn len(self: &Self) -> usize

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for PixelsMut<'a, P>

type Item = &'a mut P;

fn next(self: &mut Self) -> Option<&'a mut P>

impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for PixelsMut<'_, P>

fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result

1.22 Struct pub image::buffer::Rows

```
pub struct Rows<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Iterate over rows of an image

This iterator is created with `../struct.ImageBuffer.html#method.rows`. See its document for details.

Implementations

impl<'a, P> where <P as Pixel>::Subpixel: Sync Send for Rows<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for Rows<'a, P>

impl<'a, P> Unpin for Rows<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe UnwindSafe for Rows<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for Rows<'a, P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a DoubleEndedIterator for Rows<'a, P>

fn next_back(self: &mut Self) -> Option<Pixels<'a, P» impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for Rows<'a, P>

fn len(self: &Self) -> usize

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a Iterator for Rows<'a, P>

type Item = Pixels<'a, P>;

fn next(self: &mut Self) -> Option<Pixels<'a, P»

impl<P: Pixel> Clone for Rows<'_, P>

fn clone(self: &Self) -> Self

impl<P: Pixel> where <P as >::Subpixel: fmt::Debug Debug for Rows<'_, P>

fn fmt(self: &Self, f: &mut fmt::Formatter<'_>) -> fmt::Result

1.23 Struct pub image::buffer::RowsMut

```
pub struct RowsMut<'a, P: Pixel + 'a>
where
    <P as Pixel>::Subpixel: 'a{
    // some fields omitted
}
```

Iterate over mutable rows of an image

This iterator is created with `../struct.ImageBuffer.html#method.rows_mut`. See its document for details.

Implementations

impl<'a, P> where <P as Pixel>::Subpixel: Send Send for RowsMut<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: Sync Sync for RowsMut<'a, P>

impl<'a, P> Unpin for RowsMut<'a, P>

impl<'a, P> !UnwindSafe for RowsMut<'a, P>

impl<'a, P> where <P as Pixel>::Subpixel: RefUnwindSafe RefUnwindSafe for RowsMut<'a, P>

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a DoubleEndedIterator for RowsMut<'a, P>

fn next_back(self: &mut Self) -> Option<PixelsMut<'a, P»

impl<'a, P: Pixel + 'a> where <P as >::Subpixel: 'a ExactSizeIterator for RowsMut<'a, P>

fn len(self: &Self) -> usize

```
impl<'a, P: Pixel + 'a> where <P as >::Subpixel:  'a Iterator for RowsMut<'a, P>

 type Item = PixelsMut<'a, P>;

 fn next(self:  &mut Self) -> Option<PixelsMut<'a, P»

impl<P: Pixel> where <P as >::Subpixel:  fmt::Debug Debug for RowsMut<'_, P>

 fn fmt(self:  &Self, f:  &mut fmt::Formatter<'_>) -> fmt::Result
```

1.24 Module pub image::math

- image::math::nq
- image::math::utils
- image::math::Rect

1.25 Module pub image::math::nq

- image::math::nq::NeuQuant

1.26 Struct pub image::math::nq::NeuQuant

```rust
#[deprecated(note = "Use the `color_quant` crate instead")]
pub struct NeuQuant{
    // some fields omitted
}
```

Neural network color quantizer

1.26.1 Examples

```rust
use image::imageops::colorops::{index_colors, ColorMap};
use image::math::nq::NeuQuant;
use image::{ImageBuffer, Rgba, RgbaImage};

// Create simple color image with RGBA pixels.
let (w, h) = (2, 2);
let red: Rgba<u8> = [255, 0, 0, 255].into();
let green: Rgba<u8> = [0, 255, 0, 255].into();
let blue: Rgba<u8> = [0, 0, 255, 255].into();
let white: Rgba<u8> = [255, 255, 255, 255].into();
let mut color_image = RgbaImage::new(w, h);
color_image.put_pixel(0, 0, red);
color_image.put_pixel(1, 0, green);
color_image.put_pixel(0, 1, blue);
color_image.put_pixel(1, 1, white);

// Create a `NeuQuant` colormap that will build an approximate color palette that best matches
// the original image.
// Note, the NeuQuant algorithm is only designed to work with 6-8 bit output, so `colors`
// should be a power of 2 in the range [64, 256].
let pixels = color_image.clone().into_raw();
let cmap = NeuQuant::new(1, 256, &pixels);
// Map the original image through the color map to create an indexed image stored in a
// `GrayImage`.
let palletized = index_colors(&color_image, &cmap);
```

```
// Map indexed image back 'RgbaImage'.  Note the NeuQuant algorithm creates an approximation of
// the original colors, so even in this simple example the output is not pixel equivalent to
// the original.
let mapped = ImageBuffer::from_fn(w, h, |x, y| -> Rgba<u8> {
    let p = palletized.get_pixel(x, y);
    cmap.lookup(p.0[0] as usize)
        .expect("indexed color out-of-range")
        .into()
});
```

Implementations

`impl NeuQuant`

The implementation only calls the corresponding inner 'color_quant' methods.

These exist purely to keep a type separate from ['color_quant::NeuQuant'] and the interface stable for this major version. The type will be changed to a pure re-export in the next version or might be removed.

['color_quant::NeuQuant']: https://docs.rs/color_quant/1.1.0/color_quant/struct.NeuQuant.html

` pub fn new(samplefac:  i32, colors:  usize, pixels:  &[u8]) -> Self`

` pub fn init(self:  &mut Self, pixels:  &[u8])`

` pub fn map_pixel(self:  &Self, pixel:  &mut [u8])`

` pub fn index_of(self:  &Self, pixel:  &[u8]) -> usize`

` pub fn lookup(self:  &Self, idx:  usize) -> Option<[u8; 4]>`

`impl Send for NeuQuant`

`impl Sync for NeuQuant`

`impl Unpin for NeuQuant`

`impl UnwindSafe for NeuQuant`

`impl RefUnwindSafe for NeuQuant`

`impl ColorMap for nq::NeuQuant`

` type Color = Rgba<u8>;`

` fn index_of(self:  &Self, color:  &Rgba<u8>) -> usize`

` fn lookup(self:  &Self, idx:  usize) -> Option«Self as >::Color>`

` fn has_lookup(self:  &Self) -> bool`

Indicate NeuQuant implements 'lookup'.

` fn map_color(self:  &Self, color:  &mut Rgba<u8>)`

`impl From<NeuQuant> for NeuQuant`

` fn from(inner:  color_quant::NeuQuant) -> Self`

1.27 Module pub image::math::utils

- image::math::utils::clamp

1.28 Function pub image::math::utils::clamp

```
pub fn clamp(a: N, min: N, max: N) -> N
```

Cut value to be inside given range

```
use image::math::utils;

assert_eq!(utils::clamp(-5, 0, 10),  0);
assert_eq!(utils::clamp( 6, 0, 10),  6);
assert_eq!(utils::clamp(15, 0, 10), 10);
```

1.29 Struct pub image::math::Rect

```
pub struct Rect{
    pub x: u32,
    pub y: u32,
    pub width: u32,
    pub height: u32,
}
```

A Rectangle defined by its top left corner, width and height.

Fields

`x:` u32 The x coordinate of the top left corner.

`y:` u32 The y coordinate of the top left corner.

`width:` u32 The rectangle's width.

`height:` u32 The rectangle's height.

Implementations

`impl Send for Rect`

`impl Sync for Rect`

`impl Unpin for Rect`

`impl UnwindSafe for Rect`

`impl RefUnwindSafe for Rect`

`impl Clone for Rect`

`fn clone(self: &Self) -> Rect`

```
impl Copy for Rect

impl Eq for Rect

impl PartialEq<Rect> for Rect

 fn eq(self:  &Self, other:  &Rect) -> bool

 fn ne(self:  &Self, other:  &Rect) -> bool

impl Debug for Rect

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Hash for Rect

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl StructuralPartialEq for Rect

impl StructuralEq for Rect
```

1.30 Module pub image::imageops

- image::imageops::colorops
- image::imageops::FilterType
-
-
-
-
-
- image::imageops::flip_horizontal
- image::imageops::flip_horizontal_in_place
- image::imageops::flip_vertical
- image::imageops::flip_vertical_in_place
- image::imageops::rotate180
- image::imageops::rotate180_in_place
- image::imageops::rotate270
- image::imageops::rotate90
- image::imageops::rotate180_in
- image::imageops::rotate90_in
- image::imageops::rotate270_in
- image::imageops::flip_horizontal_in
- image::imageops::flip_vertical_in
- image::imageops::blur
- image::imageops::filter3x3
- image::imageops::resize
- image::imageops::thumbnail
- image::imageops::unsharpen
-
-
-

-
-
-
-
-
-
- image::imageops::crop
- image::imageops::crop_imm
- image::imageops::overlay_bounds
- image::imageops::overlay
- image::imageops::tile
- image::imageops::vertical_gradient
- image::imageops::horizontal_gradient
- image::imageops::replace

1.31 Module pub image::imageops::colorops

- image::imageops::colorops::grayscale
- image::imageops::colorops::invert
- image::imageops::colorops::contrast
- image::imageops::colorops::contrast_in_place
- image::imageops::colorops::brighten
- image::imageops::colorops::brighten_in_place
- image::imageops::colorops::huerotate
- image::imageops::colorops::huerotate_in_place
- image::imageops::colorops::ColorMap
- image::imageops::colorops::BiLevel
- image::imageops::colorops::dither
- image::imageops::colorops::index_colors

1.32 Function pub image::imageops::colorops::grayscale

```rust
pub fn grayscale(
    image: &I
)
  -> ImageBuffer<Luma<<<I as GenericImageView>::Pixel as Pixel>::Subpixel>, Vec<<<I as
      GenericImageView>::Pixel as Pixel>::Subpixel>>
```

Convert the supplied image to grayscale

1.33 Function pub image::imageops::colorops::invert

```rust
pub fn invert(image: &mut I)
```

Invert each pixel within the supplied image. This function operates in place.

1.34 Function pub image::imageops::colorops::contrast

```rust
pub fn contrast(image: &I, contrast: f32) -> ImageBuffer<P, Vec<S>>
```

Adjust the contrast of the supplied image. `contrast` is the amount to adjust the contrast by. Negative values decrease the contrast and positive values increase the contrast.

[See also `contrast_in_place`.][contrast_in_place]

1.35 Function pub image::imageops::colorops::contrast_in_place

```
pub fn contrast_in_place(image: &mut I, contrast: f32)
```

Adjust the contrast of the supplied image in place. `contrast` is the amount to adjust the contrast by. Negative values decrease the contrast and positive values increase the contrast.

[See also `contrast`.][contrast]

1.36 Function pub image::imageops::colorops::brighten

```
pub fn brighten(image: &I, value: i32) -> ImageBuffer<P, Vec<S>>
```

Brighten the supplied image. `value` is the amount to brighten each pixel by. Negative values decrease the brightness and positive values increase it.

[See also `brighten_in_place`.][brighten_in_place]

1.37 Function pub image::imageops::colorops::brighten_in_place

```
pub fn brighten_in_place(image: &mut I, value: i32)
```

Brighten the supplied image in place. `value` is the amount to brighten each pixel by. Negative values decrease the brightness and positive values increase it.

[See also `brighten`.][brighten]

1.38 Function pub image::imageops::colorops::huerotate

```
pub fn huerotate(image: &I, value: i32) -> ImageBuffer<P, Vec<S>>
```

Hue rotate the supplied image. `value` is the degrees to rotate each pixel by. 0 and 360 do nothing, the rest rotates by the given degree value. just like the css webkit filter hue-rotate(180)

[See also `huerotate_in_place`.][huerotate_in_place]

1.39 Function pub image::imageops::colorops::huerotate_in_place

```
pub fn huerotate_in_place(image: &mut I, value: i32)
```

Hue rotate the supplied image in place. `value` is the degrees to rotate each pixel by. 0 and 360 do nothing, the rest rotates by the given degree value. just like the css webkit filter hue-rotate(180)

[See also `huerotate`.][huerotate]

1.40 Trait pub image::imageops::colorops::ColorMap

```
pub trait ColorMap {
    type Color;
}
```

A color map

Associated items

`Color` The color type on which the map operates on

1.41 Struct pub image::imageops::colorops::BiLevel

```
pub struct BiLevel;
```

A bi-level color map

1.41.1 Examples

```
use image::imageops::colorops::{index_colors, BiLevel, ColorMap};
use image::{ImageBuffer, Luma};

let (w, h) = (16, 16);
// Create an image with a smooth horizontal gradient from black (0) to white (255).
let gray = ImageBuffer::from_fn(w, h, |x, y| -> Luma<u8> { [(255 * x / w) as u8].into() });
// Mapping the gray image through the 'BiLevel' filter should map gray pixels less than half
// intensity (127) to black (0), and anything greater to white (255).
let cmap = BiLevel;
let palletized = index_colors(&gray, &cmap);
let mapped = ImageBuffer::from_fn(w, h, |x, y| {
    let p = palletized.get_pixel(x, y);
    cmap.lookup(p.0[0] as usize)
        .expect("indexed color out-of-range")
});
// Create an black and white image of expected output.
let bw = ImageBuffer::from_fn(w, h, |x, y| -> Luma<u8> {
    if x <= (w / 2) {
        [0].into()
    } else {
        [255].into()
    }
});
assert_eq!(mapped, bw);
```

Implementations

`impl Send for BiLevel`

`impl Sync for BiLevel`

`impl Unpin for BiLevel`

`impl UnwindSafe for BiLevel`

`impl RefUnwindSafe for BiLevel`

`impl ColorMap for BiLevel`

```
type Color = Luma<u8>;
```

```
fn index_of(self:  &Self, color:  &Luma<u8>) -> usize
```

```
fn lookup(self:  &Self, idx:  usize) -> Option«Self as >::Color>
```

```
fn has_lookup(self:  &Self) -> bool
```

Indicate NeuQuant implements 'lookup'.

```
fn map_color(self:  &Self, color:  &mut Luma<u8>)
```

```
impl Clone for BiLevel
```

```
fn clone(self:  &Self) -> BiLevel
```

```
impl Copy for BiLevel
```

1.42 Function pub image::imageops::colorops::dither

```rust
pub fn dither(image: &mut ImageBuffer<Pix, Vec<u8>>, color_map: &Map)
```

Reduces the colors of the image using the supplied `color_map` while applying Floyd-Steinberg dithering to improve the visual conception

1.43 Function pub image::imageops::colorops::index_colors

```rust
pub fn index_colors(
    image: &ImageBuffer<Pix, Vec<u8>>, color_map: &Map
)
  -> ImageBuffer<Luma<u8>, Vec<u8>>
```

Reduces the colors using the supplied `color_map` and returns an image of the indices

1.44 Enum pub image::imageops::FilterType

```rust
pub enum image::imageops::FilterType {
    Nearest,
    Triangle,
    CatmullRom,
    Gaussian,
    Lanczos3,
}
```

Available Sampling Filters.

Examples

To test the different sampling filters on a real example, you can find two examples called `scaledown` and `scaleup` in the `examples` directory of the crate source code.

Here is a 3.58 MiB test image that has been scaled down to 300x225 px:

<!– NOTE: To test new test images locally, replace the GitHub path with '../../../docs/' –> <div style="display: flex; flex-wrap: wrap; align-items: flex-start;"> <div style="margin: 0 8px 8px 0;"> <img src="https://raw.githubusercontent.com/image-rs/image/master/examples/scaledown/scaledown-test-near.png" title="Nearest"><br> Nearest Neighbor </div> <div style="margin: 0 8px 8px 0;"> <img src="https://raw.githubusercontent.com/image-rs/image/master/examples/scaledown/scaledown-test-tri.png" title="Triangle"><br> Linear: Triangle </div> <div style="margin: 0 8px 8px 0;"> <img src="https://raw.githubusercontent.com/image-rs/image/master/examples/scaledown/scaledown-test-cmr.png" title="CatmullRom"><br> Cubic: Catmull-Rom </div> <div style="margin: 0 8px 8px 0;"> <img src="https://raw.githubusercontent.com/image-rs/image/master/examples/scaledown/scaledown-test-gauss.png" title="Gaussian"><br> Gaussian </div> <div style="margin: 0 8px 8px 0;"> <img src="https://raw.githubusercontent.com/image-rs/image/master/examples/scaledown/scaledown-test-lcz2.png" title="Lanczos3"><br> Lanczos with window 3 </div> </div>

Speed

Time required to create each of the examples above, tested on an Intel i7-4770 CPU with Rust 1.37 in release mode:

<table style="width: auto;"> <tr> <th>Nearest</th> <td>31 ms</td> </tr> <tr> <th>Triangle</th> <td>414 ms</td> </tr> <tr> <th>CatmullRom</th> <td>817 ms</td> </tr> <tr> <th>Gaussian</th> <td>1180 ms</td> </tr> <tr> <th>Lanczos3</th> <td>1170 ms</td> </tr> </table>

Variants

`Nearest` Nearest Neighbor

`Triangle` Linear Filter

`CatmullRom` Cubic Filter

`Gaussian` Gaussian Filter

`Lanczos3` Lanczos with window 3

Implementations

`impl Send for FilterType`

`impl Sync for FilterType`

`impl Unpin for FilterType`

`impl UnwindSafe for FilterType`

`impl RefUnwindSafe for FilterType`

`impl Clone for FilterType`

```
 fn clone(self:  &Self) -> FilterType
```

```
impl Copy for FilterType

impl PartialEq<FilterType> for FilterType

 fn eq(self: &Self, other: &FilterType) -> bool

impl Debug for FilterType

 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for FilterType
```

1.45 Re-export pub CatmullRom

```
pub use CatmullRom;
```

1.46 Re-export pub Gaussian

```
pub use Gaussian;
```

1.47 Re-export pub Lanczos3

```
pub use Lanczos3;
```

1.48 Re-export pub Nearest

```
pub use Nearest;
```

1.49 Re-export pub Triangle

```
pub use Triangle;
```

1.50 Function pub image::imageops::flip_horizontal

```
pub fn flip_horizontal(
    image: &I
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Flip an image horizontally

1.51 Function pub image::imageops::flip_horizontal_in_place

```
pub fn flip_horizontal_in_place(image: &mut I)
```

Flip an image horizontally in place.

1.52 Function pub image::imageops::flip_vertical

```
pub fn flip_vertical(
    image: &I
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Flip an image vertically

1.53 Function pub image::imageops::flip_vertical_in_place

```
pub fn flip_vertical_in_place(image: &mut I)
```

Flip an image vertically in place.

1.54 Function pub image::imageops::rotate180

```
pub fn rotate180(
    image: &I
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Rotate an image 180 degrees clockwise.

1.55 Function pub image::imageops::rotate180_in_place

```
pub fn rotate180_in_place(image: &mut I)
```

Rotate an image 180 degrees clockwise in place.

1.56 Function pub image::imageops::rotate270

```
pub fn rotate270(
    image: &I
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Rotate an image 270 degrees clockwise.

1.57 Function pub image::imageops::rotate90

```
pub fn rotate90(
    image: &I
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Rotate an image 90 degrees clockwise.

1.58 Function pub image::imageops::rotate180_in

```
pub fn rotate180_in(
    image: &I, destination: &mut ImageBuffer<<I as >::Pixel, Container>
)
  -> crate::ImageResult<()>
```

Rotate an image 180 degrees clockwise and put the result into the destination [`ImageBuffer`].

1.59 Function pub image::imageops::rotate90_in

```
pub fn rotate90_in(
    image: &I, destination: &mut ImageBuffer<<I as >::Pixel, Container>
)
  -> crate::ImageResult<()>
```

Rotate an image 90 degrees clockwise and put the result into the destination [`ImageBuffer`].

1.60 Function pub image::imageops::rotate270_in

```
pub fn rotate270_in(
    image: &I, destination: &mut ImageBuffer<<I as >::Pixel, Container>
)
  -> crate::ImageResult<()>
```

Rotate an image 270 degrees clockwise and put the result into the destination [`ImageBuffer`].

1.61 Function pub image::imageops::flip_horizontal_in

```
pub fn flip_horizontal_in(
    image: &I, destination: &mut ImageBuffer<<I as >::Pixel, Container>
)
  -> crate::ImageResult<()>
```

Flip an image horizontally and put the result into the destination [`ImageBuffer`].

1.62 Function pub image::imageops::flip_vertical_in

```
pub fn flip_vertical_in(
    image: &I, destination: &mut ImageBuffer<<I as >::Pixel, Container>
)
  -> crate::ImageResult<()>
```

Flip an image vertically and put the result into the destination [`ImageBuffer`].

1.63 Function pub image::imageops::blur

```
pub fn blur(
    image: &I, sigma: f32
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Performs a Gaussian blur on the supplied image. `sigma` is a measure of how much to blur by.

1.64 Function pub image::imageops::filter3x3

```
pub fn filter3x3(image: &I, kernel: &[f32]) -> ImageBuffer<P, Vec<S>>
```

Perform a 3x3 box filter on the supplied image. `kernel` is an array of the filter weights of length 9.

1.65 Function pub image::imageops::resize

```
pub fn resize(
    image: &I, nwidth: u32, nheight: u32, filter: FilterType
)
  -> ImageBuffer<<I as >::Pixel, Vec<<<I as >::Pixel as Pixel>::Subpixel>>
```

Resize the supplied image to the specified dimensions. `nwidth` and `nheight` are the new dimensions. `filter` is the sampling filter to use.

1.66 Function pub image::imageops::thumbnail

```
pub fn thumbnail(image: &I, new_width: u32, new_height: u32) -> ImageBuffer<P, Vec<S>>
```

Resize the supplied image to the specific dimensions.

For downscaling, this method uses a fast integer algorithm where each source pixel contributes to exactly one target pixel. May give aliasing artifacts if new size is close to old size.

In case the current width is smaller than the new width or similar for the height, another strategy is used instead. For each pixel in the output, a rectangular region of the input is determined, just as previously. But when no input pixel is part of this region, the nearest pixels are interpolated instead.

For speed reasons, all interpolation is performed linearly over the colour values. It will not take the pixel colour spaces into account.

1.67 Function pub image::imageops::unsharpen

```
pub fn unsharpen(image: &I, sigma: f32, threshold: i32) -> ImageBuffer<P, Vec<S>>
```

Performs an unsharpen mask on the supplied image. `sigma` is the amount to blur the image by. `threshold` is the threshold for the difference between

See `https://en.wikipedia.org/wiki/Unsharp_masking#Digital_unsharp_masking`

1.68 Re-export pub brighten

```
pub use brighten;
```

Color operations

1.69 Re-export pub contrast

```
pub use contrast;
```

Color operations

1.70 Re-export pub dither

```
pub use dither;
```

Color operations

1.71 Re-export pub grayscale

```
pub use grayscale;
```

Color operations

1.72 Re-export pub huerotate

```
pub use huerotate;
```

Color operations

1.73 Re-export pub index_colors

```
pub use index_colors;
```

Color operations

1.74 Re-export pub invert

```
pub use invert;
```

Color operations

1.75 Re-export pub BiLevel

```
pub use BiLevel;
```

Color operations

1.76 Re-export pub ColorMap

```
pub use ColorMap;
```

Color operations

1.77 Function pub image::imageops::crop

```
pub fn crop(image: &mut I, x: u32, y: u32, width: u32, height: u32) -> SubImage<&mut I>
```

Return a mutable view into an image The coordinates set the position of the top left corner of the crop.

1.78 Function pub image::imageops::crop_imm

```
pub fn crop_imm(image: &I, x: u32, y: u32, width: u32, height: u32) -> SubImage<&I>
```

Return an immutable view into an image The coordinates set the position of the top left corner of the crop.

1.79 Function pub image::imageops::overlay_bounds

```
pub fn overlay_bounds(
    (bottom_width, bottom_height): (u32,u32), (top_width, top_height): (u32,u32), x: u32, y: u32
)
  -> (u32,u32)
```

Calculate the region that can be copied from top to bottom.

Given image size of bottom and top image, and a point at which we want to place the top image onto the bottom image, how large can we be? Have to wary of the following issues:

- Top might be larger than bottom
- Overflows in the computation
- Coordinates could be completely out of bounds

The main idea is to make use of inequalities provided by the nature of `saturing_add` and `saturating_sub`. These intrinsically validate that all resulting coordinates will be in bounds for both images.

We want that all these coordinate accesses are safe:

1. `bottom.get_pixel(x + [0..x_range), y + [0..y_range))`
2. `top.get_pixel([0..x_range), [0..y_range))`

Proof that the function provides the necessary bounds for width. Note that all unaugmented math operations are to be read in standard arithmetic, not integer arithmetic. Since no direct integer arithmetic occurs in the implementation, this is unambiguous.

```
Three short notes/lemmata:
- Iff `(a - b) <= 0` then `a.saturating_sub(b) = 0`
- Iff `(a - b) >= 0` then `a.saturating_sub(b) = a - b`
- If  `a <= c` then `a.saturating_sub(b) <= c.saturating_sub(b)`

1.1 We show that if `bottom_width <= x`, then `x_range = 0` therefore `x + [0..x_range)` is empty.

x_range
 = (top_width.saturating_add(x).min(bottom_width)).saturating_sub(x)
<= bottom_width.saturating_sub(x)

bottom_width <= x
<==> bottom_width - x <= 0
<==> bottom_width.saturating_sub(x) = 0
 ==> x_range <= 0
 ==> x_range  = 0

1.2 If `x < bottom_width` then `x + x_range < bottom_width`

x + x_range
<= x + bottom_width.saturating_sub(x)
 = x + (bottom_width - x)
 = bottom_width

2. We show that `x_range <= top_width`

x_range
 = (top_width.saturating_add(x).min(bottom_width)).saturating_sub(x)
<= top_width.saturating_add(x).saturating_sub(x)
<= (top_wdith + x).saturating_sub(x)
 = top_width (due to `top_width >= 0` and `x >= 0`)
```

Proof is the same for height.

1.80 Function pub image::imageops::overlay

```rust
pub fn overlay(bottom: &mut I, top: &J, x: u32, y: u32)
```

Overlay an image at a given coordinate (x, y)

1.81 Function pub image::imageops::tile

```rust
pub fn tile(bottom: &mut I, top: &J)
```

Tile an image by repeating it multiple times

1.81.1 Examples

```rust
use image::{RgbaImage};

fn main() {
     let mut img = RgbaImage::new(1920, 1080);
     let tile = image::open("tile.png").unwrap();

     image::imageops::tile(&mut img, &tile);
     img.save("tiled_wallpaper.png").unwrap();
}
```

1.82 Function pub image::imageops::vertical_gradient

```
pub fn vertical_gradient(img: &mut I, start: &P, stop: &P)
```

Fill the image with a linear vertical gradient

This function assumes a linear color space.

1.82.1 Examples

```
use image::{Rgba, RgbaImage, Pixel};

fn main() {
    let mut img = RgbaImage::new(100, 100);
    let start = Rgba::from_slice(&[0, 128, 0, 0]);
    let end = Rgba::from_slice(&[255, 255, 255, 255]);

    image::imageops::vertical_gradient(&mut img, start, end);
    img.save("vertical_gradient.png").unwrap();
}
```

1.83 Function pub image::imageops::horizontal_gradient

```
pub fn horizontal_gradient(img: &mut I, start: &P, stop: &P)
```

Fill the image with a linear horizontal gradient

This function assumes a linear color space.

1.83.1 Examples

```
use image::{Rgba, RgbaImage, Pixel};

fn main() {
    let mut img = RgbaImage::new(100, 100);
    let start = Rgba::from_slice(&[0, 128, 0, 0]);
    let end = Rgba::from_slice(&[255, 255, 255, 255]);

    image::imageops::horizontal_gradient(&mut img, start, end);
    img.save("horizontal_gradient.png").unwrap();
}
```

1.84 Function pub image::imageops::replace

```
pub fn replace(bottom: &mut I, top: &J, x: u32, y: u32)
```

Replace the contents of an image at a given coordinate (x, y)

1.85 Module pub image::io

- image::io::Reader

1.86 Struct pub image::io::Reader

```
pub struct Reader<R: Read>{
    // some fields omitted
}
```

A multi-format image reader.

Wraps an input reader to facilitate automatic detection of an image's format, appropriate decoding method, and dispatches into the set of supported `../trait.ImageDecoder.html` implementations.

Usage

Opening a file, deducing the format based on the file path automatically, and trying to decode the image contained can be performed by constructing the reader and immediately consuming it.

```
# use image::ImageError;
# use image::io::Reader;
# fn main() -> Result<(), ImageError> {
let image = Reader::open("path/to/image.png")?
    .decode()?;
# Ok(()) }
```

It is also possible to make a guess based on the content. This is especially handy if the source is some blob in memory and you have constructed the reader in another way. Here is an example with a `pnm` black-and-white subformat that encodes its pixel matrix with ascii values.

```
# use image::ImageError;
# use image::io::Reader;
# fn main() -> Result<(), ImageError> {
use std::io::Cursor;
use image::ImageFormat;

let raw_data = b"P1 2 2\n\
    0 1\n\
    1 0\n";

let mut reader = Reader::new(Cursor::new(raw_data))
    .with_guessed_format()
    .expect("Cursor io never fails");
assert_eq!(reader.format(), Some(ImageFormat::Pnm));

let image = reader.decode()?;
# Ok(()) }
```

As a final fallback or if only a specific format must be used, the reader always allows manual specification of the supposed image format with **??**.

Implementations

`impl<R: Read> Reader<R>`

` pub fn new(reader: R) -> Self`

Create a new image reader without a preset format.

It is possible to guess the format based on the content of the read object with ['with_guessed_format'], or to set the format directly with ['set_format'].

['with_guessed_format']: #method.with_guessed_format ['set_format']: method.set_format

```
pub fn with_format(reader:  R, format:  ImageFormat) -> Self
```

Construct a reader with specified format.

```
pub fn format(self:  &Self) -> Option<ImageFormat>
```

Get the currently determined format.

```
pub fn set_format(self:  &mut Self, format:  ImageFormat)
```

Supply the format as which to interpret the read image.

```
pub fn clear_format(self:  &mut Self)
```

Remove the current information on the image format.

Note that many operations require format information to be present and will return e.g. an 'ImageError::Unsupported' when the image format has not been set.

```
pub fn into_inner(self:  Self) -> R
```

Unwrap the reader.

```
impl Reader<BufReader<File»
```

```
pub fn open<P> where P: AsRef<Path>(path:  P) -> io::Result<Self>
```

Open a file to read, format will be guessed from path.

This will not attempt any io operation on the opened file.

If you want to inspect the content for a better guess on the format, which does not depend on file extensions, follow this call with a call to ['with_guessed_format'].

['with_guessed_format']: #method.with_guessed_format

```
impl<R: BufRead + Seek> Reader<R>
```

```
pub fn with_guessed_format(mut self:  Self) -> io::Result<Self>
```

Make a format guess based on the content, replacing it on success.

Returns 'Ok' with the guess if no io error occurs. Additionally, replaces the current format if the guess was successful. If the guess was unable to determine a format then the current format of the reader is unchanged.

Returns an error if the underlying reader fails. The format is unchanged. The error is a 'std::io::Error' and not 'ImageError' since the only error case is an error when the underlying reader seeks.

When an error occurs, the reader may not have been properly reset and it is potentially hazardous to continue with more io.

## Usage

This supplements the path based type deduction from [‘open‘](Reader::open) with content based deduction. This is more common in Linux and UNIX operating systems and also helpful if the path can not be directly controlled.

‘‘‘no_run # use image::ImageError; # use image::io::Reader; # fn main() -> Result<(), ImageError> { let image = Reader::open(“image.unknown”)? .with_guessed_format()? .decode()?; # Ok(()) } ‘‘‘

```
pub fn into_dimensions(mut self: Self) -> ImageResult<(u32,u32)>
```

Read the image dimensions.

Uses the current format to construct the correct reader for the format.

If no format was determined, returns an ‘ImageError::Unsupported‘.

```
pub fn decode(mut self: Self) -> ImageResult<DynamicImage>
```

Read the image (replaces ‘load‘).

Uses the current format to construct the correct reader for the format.

If no format was determined, returns an ‘ImageError::Unsupported‘.

```
impl<R> where R: Send Send for Reader<R>
```

```
impl<R> where R: Sync Sync for Reader<R>
```

```
impl<R> where R: Unpin Unpin for Reader<R>
```

```
impl<R> where R: UnwindSafe UnwindSafe for Reader<R>
```

```
impl<R> where R: RefUnwindSafe RefUnwindSafe for Reader<R>
```

1.87 Module pub image::flat

- image::flat::FlatSamples
- image::flat::SampleLayout
- image::flat::View
- image::flat::ViewMut
- image::flat::Error
- image::flat::NormalForm

1.88 Struct pub image::flat::FlatSamples

```
pub struct FlatSamples<Buffer>{
    pub samples: Buffer,
    pub layout: SampleLayout,
    pub color_hint: Option<ColorType>,
}
```

A flat buffer over a (multi channel) image.

In contrast to `ImageBuffer`, this representation of a sample collection is much more lenient in the layout thereof. It also allows grouping by color planes instead of by pixel as long as the strides of each extent are constant. This struct itself has no invariants on the strides but not every possible configuration can be interpreted as a `../trait.GenericImageView.html` or `../trait.GenericImage.html`. The methods **??** and **??** construct the actual implementors of these traits and perform necessary checks. To manually perform this and other layout checks use **??** or **??**.

Instances can be constructed not only by hand. The buffer instances returned by library functions such as `../struct.ImageBuffer.html#method.as_flat_samples` guarantee that the conversion to a generic image or generic view succeeds. A very different constructor is **??**. It uses a single pixel as the backing storage for an arbitrarily sized read-only raster by mapping each pixel to the same samples by setting some strides to `0`.

Fields

`samples: Buffer` Underlying linear container holding sample values.

`layout: SampleLayout` A `repr(C)` description of the layout of buffer samples.

`color_hint: Option<ColorType>` Supplementary color information.

You may keep this as `None` in most cases. This is NOT checked in `View` or other converters. It is intended mainly as a way for types that convert to this buffer type to attach their otherwise static color information. A dynamic image representation could however use this to resolve representational ambiguities such as the order of RGB channels.

Implementations

```
impl<Buffer> FlatSamples<Buffer>
```

```
 pub fn strides_cwh(self: &Self) -> (usize,usize,usize)
```

Get the strides for indexing matrix-like '[(c, w, h)]'.

For a row-major layout with grouped samples, this tuple is strictly increasing.

```
 pub fn extents(self: &Self) -> (usize,usize,usize)
```

Get the dimensions '(channels, width, height)'.

The interface is optimized for use with 'strides_cwh' instead. The channel extent will be before width and height.

```
 pub fn bounds(self: &Self) -> (u8,u32,u32)
```

Tuple of bounds in the order of coordinate inputs.

This function should be used whenever working with image coordinates opposed to buffer coordinates. The only difference compared to 'extents' is the output type.

```
 pub fn as_ref<T> where Buffer: AsRef<[T]>(self: &Self) -> FlatSamples<&[T]>
```

Get a reference based version.

```
pub fn as_mut<T> where Buffer:  AsMut<[T]>(self:  &mut Self) -> FlatSamples<&mut [T]>
```

Get a mutable reference based version.

```
pub fn to_vec<T> where T: Clone, Buffer:  AsRef<[T]>(self:  &Self) -> FlatSamples<Vec<T>>
```

Copy the data into an owned vector.

```
pub fn get_sample<T> where Buffer:  AsRef<[T]>(self:  &Self, channel:  u8, x:  u32, y:
u32) -> Option<&T>
```

Get a reference to a single sample.

This more restrictive than the method based on 'std::ops::Index' but guarantees to properly check all bounds and not panic as long as 'Buffer::as_ref' does not do so.

"' # use image::{RgbImage}; let flat = RgbImage::new(480, 640).into_flat_samples();

// Get the blue channel at (10, 10). assert!(flat.get_sample(1, 10, 10).is_some());

// There is no alpha channel. assert!(flat.get_sample(3, 10, 10).is_none()); "'

For cases where a special buffer does not provide 'AsRef<[T]>', consider encapsulating bounds checks with 'min_length' in a type similar to 'View'. Then you may use 'in_bounds_index' as a small speedup over the index calculation of this method which relies on 'index_ignoring_bounds' since it can not have a-priori knowledge that the sample coordinate is in fact backed by any memory buffer.

```
pub fn get_mut_sample<T> where Buffer:  AsMut<[T]>(self:  &mut Self, channel:  u8, x:
u32, y:  u32) -> Option<&mut T>
```

Get a mutable reference to a single sample.

This more restrictive than the method based on 'std::ops::IndexMut' but guarantees to properly check all bounds and not panic as long as 'Buffer::as_ref' does not do so. Contrary to conversion to 'ViewMut', this does not require that samples are packed since it does not need to convert samples to a color representation.

WARNING: Note that of course samples may alias, so that the mutable reference returned here can in fact modify more than the coordinate in the argument.

"' # use image::{RgbImage}; let mut flat = RgbImage::new(480, 640).into_flat_samples();

// Assign some new color to the blue channel at (10, 10). *flat.get_mut_sample(1, 10, 10).unwrap() = 255;

// There is no alpha channel. assert!(flat.get_mut_sample(3, 10, 10).is_none()); "'

For cases where a special buffer does not provide 'AsRef<[T]>', consider encapsulating bounds checks with 'min_length' in a type similar to 'View'. Then you may use 'in_bounds_index' as a small speedup over the index calculation of this method which relies on 'index_ignoring_bounds' since it can not have a-priori knowledge that the sample coordinate is in fact backed by any memory buffer.

```
pub fn as_view<P> where P: Pixel, Buffer:  AsRef<[<P as >::Subpixel]>(self:  &Self) ->
Result<View<&[<P as >::Subpixel], P>, Error>
```

View this buffer as an image over some type of pixel.

This first ensures that all in-bounds coordinates refer to valid indices in the sample buffer. It also checks that the specified pixel format expects the same number of channels that are present in this buffer. Neither are larger nor a smaller number will be accepted. There is no automatic conversion.

```
 pub fn as_view_with_mut_samples<P> where P: Pixel, Buffer:  AsMut<[<P as >::Subpixel]>(self:
&mut Self) -> Result<View<&mut [<P as >::Subpixel], P>, Error>
```

View this buffer but keep mutability at a sample level.

This is similar to 'as_view' but subtly different from 'as_view_mut'. The resulting type can be used as a 'GenericImage' with the same prior invariants needed as for 'as_view'. It can not be used as a mutable 'GenericImage' but does not need channels to be packed in their pixel representation.

This first ensures that all in-bounds coordinates refer to valid indices in the sample buffer. It also checks that the specified pixel format expects the same number of channels that are present in this buffer. Neither are larger nor a smaller number will be accepted. There is no automatic conversion.

WARNING: Note that of course samples may alias, so that the mutable reference returned for one sample can in fact modify other samples as well. Sometimes exactly this is intended.

```
 pub fn as_view_mut<P> where P: Pixel, Buffer:  AsMut<[<P as >::Subpixel]>(self:  &mut
Self) -> Result<ViewMut<&mut [<P as >::Subpixel], P>, Error>
```

Interpret this buffer as a mutable image.

To succeed, the pixels in this buffer may not alias each other and the samples of each pixel must be packed (i.e. 'channel_stride' is '1'). The number of channels must be consistent with the channel count expected by the pixel format.

This is similar to an 'ImageBuffer' except it is a temporary view that is not normalized as strongly. To get an owning version, consider copying the data into an 'ImageBuffer'. This provides many more operations, is possibly faster (if not you may want to open an issue) is generally polished. You can also try to convert this buffer inline, see 'ImageBuffer::from_raw'.

```
 pub fn as_slice<T> where Buffer:  AsRef<[T]>(self:  &Self) -> &[T]
```

View the samples as a slice.

The slice is not limited to the region of the image and not all sample indices are valid indices into this buffer. See 'image_mut_slice' as an alternative.

```
 pub fn as_mut_slice<T> where Buffer:  AsMut<[T]>(self:  &mut Self) -> &mut [T]
```

View the samples as a slice.

The slice is not limited to the region of the image and not all sample indices are valid indices into this buffer. See 'image_mut_slice' as an alternative.

```
 pub fn image_slice<T> where Buffer:  AsRef<[T]>(self:  &Self) -> Option<&[T]>
```

Return the portion of the buffer that holds sample values.

This may fail when the coordinates in this image are either out-of-bounds of the underlying buffer or can not be represented. Note that the slice may have holes that do not correspond to any sample in the image represented by it.

```
pub fn image_mut_slice<T> where Buffer: AsMut<[T]>(self: &mut Self) -> Option<&mut [T]>
```

Mutable portion of the buffer that holds sample values.

```
pub fn try_into_buffer<P> where P: Pixel + 'static, <P as >::Subpixel: 'static, Buffer:
Deref<Target=[<P as >::Subpixel]>(self: Self) -> Result<ImageBuffer<P, Buffer>, (Error,Self)>
```

Move the data into an image buffer.

This does **not** convert the sample layout. The buffer needs to be in packed row-major form before calling this function. In case of an error, returns the buffer again so that it does not release any allocation.

```
pub fn min_length(self: &Self) -> Option<usize>
```

Get the minimum length of a buffer such that all in-bounds samples have valid indices.

This method will allow zero strides, allowing compact representations of monochrome images. To check that no aliasing occurs, try 'check_alias_invariants'. For compact images (no aliasing and no unindexed samples) this is 'width*height*channels'. But for both of the other cases, the reasoning is slightly more involved.

# Explanation

Note that there is a difference between 'min_length' and the index of the sample 'one-past-the-end'. This is due to strides that may be larger than the dimension below.

## Example with holes

Let's look at an example of a grayscale image with * 'width_stride = 1' * 'width = 2' * 'height_stride = 3' * 'height = 2'

"'text | x x | x x m | $ min_length m ^ ^ one-past-the-end $ "'

The difference is also extreme for empty images with large strides. The one-past-the-end sample index is still as large as the largest of these strides while 'min_length = 0'.

## Example with aliasing

The concept gets even more important when you allow samples to alias each other. Here we have the buffer of a small grayscale image where this is the case, this time we will first show the buffer and then the individual rows below.

* 'width_stride = 1' * 'width = 3' * 'height_stride = 2' * 'height = 2'

"'text 1 2 3 4 5 m |1 2 3| row one |3 4 5| row two ^ m min_length ^ ??? one-past-the-end "'

This time 'one-past-the-end' is not even simply the largest stride times the extent of its dimension. That still points inside the image because 'height*height_stride = 4' but also 'index_of(1, 2) = 4'.

```
pub fn fits(self: &Self, len: usize) -> bool
```

Check if a buffer of length ‘len‘ is large enough.

```
pub fn has_aliased_samples(self: &Self) -> bool
```

If there are any samples aliasing each other.

If this is not the case, it would always be safe to allow mutable access to two different samples at the same time. Otherwise, this operation would need additional checks. When one dimension overflows ‘usize‘ with its stride we also consider this aliasing.

```
pub fn is_normal(self: &Self, form: NormalForm) -> bool
```

Check if a buffer fulfills the requirements of a normal form.

Certain conversions have preconditions on the structure of the sample buffer that are not captured (by design) by the type system. These are then checked before the conversion. Such checks can all be done in constant time and will not inspect the buffer content. You can perform these checks yourself when the conversion is not required at this moment but maybe still performed later.

```
pub fn in_bounds(self: &Self, channel: u8, x: u32, y: u32) -> bool
```

Check that the pixel and the channel index are in bounds.

An in-bound coordinate does not yet guarantee that the corresponding calculation of a buffer index does not overflow. However, if such a buffer large enough to hold all samples actually exists in memory, this porperty of course follows.

```
pub fn index(self: &Self, channel: u8, x: u32, y: u32) -> Option<usize>
```

Resolve the index of a particular sample.

‘None‘ if the index is outside the bounds or does not fit into a ‘usize‘.

```
pub fn index_ignoring_bounds(self: &Self, channel: usize, x: usize, y: usize) -> Option<usi
```

Get the theoretical position of sample (x, y, channel).

The 'check' is for overflow during index calculation, not that it is contained in the image. Two samples may return the same index, even when one of them is out of bounds. This happens when all strides are ‘0‘, i.e. the image is an arbitrarily large monochrome image.

```
pub fn in_bounds_index(self: &Self, channel: u8, x: u32, y: u32) -> usize
```

Get an index provided it is inbouds.

Assumes that the image is backed by some sufficiently large buffer. Then computation can not overflow as we could represent the maximum coordinate. Since overflow is defined either way, this method can not be unsafe.

```
pub fn shrink_to(self: &mut Self, channels: u8, width: u32, height: u32)
```

Shrink the image to the minimum of current and given extents.

This does not modify the strides, so that the resulting sample buffer may have holes created by the shrinking operation. Shrinking could also lead to an non-aliasing image when samples had aliased each other before.

```
impl<'buf, Subpixel> FlatSamples<&'buf [Subpixel]>
```

```
 pub fn with_monocolor<P> where P: Pixel<Subpixel=Subpixel>, Subpixel:  Primitive(pixel:
&'buf P, width:  u32, height:  u32) -> Self
```

Create a monocolor image from a single pixel.

This can be used as a very cheap source of a 'GenericImageView' with an arbitrary number of pixels of a single color, without any dynamic allocation.

## Examples

"' # fn paint_something<T>(_: T) {} use image::{flat::FlatSamples, GenericImage, RgbImage, Rgb};

let background = Rgb([20, 20, 20]); let bg = FlatSamples::with_monocolor(&background, 200, 200);;

let mut image = RgbImage::new(200, 200); paint_something(&mut image);

// Reset the canvas image.copy_from(&bg.as_view().unwrap(), 0, 0); "'

```
impl<Buffer> where Buffer:  Send Send for FlatSamples<Buffer>
```

```
impl<Buffer> where Buffer:  Sync Sync for FlatSamples<Buffer>
```

```
impl<Buffer> where Buffer:  Unpin Unpin for FlatSamples<Buffer>
```

```
impl<Buffer> where Buffer:  UnwindSafe UnwindSafe for FlatSamples<Buffer>
```

```
impl<Buffer> where Buffer:  RefUnwindSafe RefUnwindSafe for FlatSamples<Buffer>
```

```
impl<Buffer:  $crate::clone::Clone> Clone for FlatSamples<Buffer>
```

```
 fn clone(self:  &Self) -> FlatSamples<Buffer>
```

```
impl<Buffer:  $crate::fmt::Debug> Debug for FlatSamples<Buffer>
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl<Buffer> where Buffer:  Index<usize> Index<(u8,u32,u32)> for FlatSamples<Buffer>
```

```
 type Output = <Buffer as >::Output;
```

```
 fn index(self:  &Self, (c, x, y):  (u8,u32,u32)) -> &<Self as >::Output
```

Return a reference to a single sample at specified coordinates.

# Panics

When the coordinates are out of bounds or the index calculation fails.

```
impl<Buffer> where Buffer:  IndexMut<usize> IndexMut<(u8,u32,u32)> for FlatSamples<Buffer>
```

```
 fn index_mut(self:  &mut Self, (c, x, y):  (u8,u32,u32)) -> &mut <Self as >::Output
```

Return a mutable reference to a single sample at specified coordinates.

# Panics

When the coordinates are out of bounds or the index calculation fails.

1.89 Struct pub image::flat::SampleLayout

```rust
#[repr(C)]
pub struct SampleLayout{
    pub channels: u8,
    pub channel_stride: usize,
    pub width: u32,
    pub width_stride: usize,
    pub height: u32,
    pub height_stride: usize,
}
```

A ffi compatible description of a sample buffer.

Fields

`channels: u8` The number of channels in the color representation of the image.

`channel_stride: usize` Add this to an index to get to the sample in the next channel.

`width: u32` The width of the represented image.

`width_stride: usize` Add this to an index to get to the next sample in x-direction.

`height: u32` The height of the represented image.

`height_stride: usize` Add this to an index to get to the next sample in y-direction.

Implementations

`impl SampleLayout`

`pub fn row_major_packed(channels: u8, width: u32, height: u32) -> Self`

Describe a row-major image packed in all directions.

The resulting will surely be 'NormalForm::RowMajorPacked'. It can therefore be converted to safely to an 'ImageBuffer' with a large enough underlying buffer.

"' # use image::flat::{NormalForm, SampleLayout}; let layout = SampleLayout::row_major_packed(3, 640, 480); assert!(layout.is_normal(NormalForm::RowMajorPacked)); "'

# Panics

On platforms where 'usize' has the same size as 'u32' this panics when the resulting stride in the 'height' direction would be larger than 'usize::max_value()'. On other platforms where it can surely accomodate 'u8::max_value() * u32::max_value(), this can never happen.

`pub fn column_major_packed(channels: u8, width: u32, height: u32) -> Self`

Describe a column-major image packed in all directions.

The resulting will surely be ‘NormalForm::ColumnMajorPacked‘. This is not particularly useful for conversion but can be used to describe such a buffer without pitfalls.

‘‘‘ # use image::flat::{NormalForm, SampleLayout}; let layout = SampleLayout::column_major_packed(3, 640, 480); assert!(layout.is_normal(NormalForm::ColumnMajorPacked)); ‘‘‘

# Panics

On platforms where ‘usize‘ has the same size as ‘u32‘ this panics when the resulting stride in the ‘width‘ direction would be larger than ‘usize::max_value()‘. On other platforms where it can surely accomodate ‘u8::max_value() * u32::max_value(), this can never happen.

```
pub fn strides_cwh(self:  &Self) -> (usize,usize,usize)
```

Get the strides for indexing matrix-like ‘[(c, w, h)]‘.

For a row-major layout with grouped samples, this tuple is strictly increasing.

```
pub fn extents(self:  &Self) -> (usize,usize,usize)
```

Get the dimensions ‘(channels, width, height)‘.

The interface is optimized for use with ‘strides_cwh‘ instead. The channel extent will be before width and height.

```
pub fn bounds(self:  &Self) -> (u8,u32,u32)
```

Tuple of bounds in the order of coordinate inputs.

This function should be used whenever working with image coordinates opposed to buffer coordinates. The only difference compared to ‘extents‘ is the output type.

```
pub fn min_length(self:  &Self) -> Option<usize>
```

Get the minimum length of a buffer such that all in-bounds samples have valid indices.

This method will allow zero strides, allowing compact representations of monochrome images. To check that no aliasing occurs, try ‘check_alias_invariants‘. For compact images (no aliasing and no unindexed samples) this is ‘width*height*channels‘. But for both of the other cases, the reasoning is slightly more involved.

# Explanation

Note that there is a difference between ‘min_length‘ and the index of the sample 'one-past-the-end‘. This is due to strides that may be larger than the dimension below.

## Example with holes

Let’s look at an example of a grayscale image with * ‘width_stride = 1‘ * ‘width = 2‘ * ‘height_stride = 3‘ * ‘height = 2‘

‘‘‘text | x x | x x m | $ min_length m ^ ^ one-past-the-end $ ‘‘‘

The difference is also extreme for empty images with large strides. The one-past-the-end sample index is still as large as the largest of these strides while ‘min_length = 0‘.

## Example with aliasing

The concept gets even more important when you allow samples to alias each other. Here we have the buffer of a small grayscale image where this is the case, this time we will first show the buffer and then the individual rows below.

* ‘width_stride = 1‘ * ‘width = 3‘ * ‘height_stride = 2‘ * ‘height = 2‘

“‘text 1 2 3 4 5 m |1 2 3| row one |3 4 5| row two ^ m min_length ^ ??? one-past-the-end “‘

This time 'one-past-the-end' is not even simply the largest stride times the extent of its dimension. That still points inside the image because ‘height*height_stride = 4‘ but also ‘index_of(1, 2) = 4‘.

```
pub fn fits(self: &Self, len: usize) -> bool
```

Check if a buffer of length ‘len‘ is large enough.

```
pub fn has_aliased_samples(self: &Self) -> bool
```

If there are any samples aliasing each other.

If this is not the case, it would always be safe to allow mutable access to two different samples at the same time. Otherwise, this operation would need additional checks. When one dimension overflows ‘usize‘ with its stride we also consider this aliasing.

```
pub fn is_normal(self: &Self, form: NormalForm) -> bool
```

Check if a buffer fulfills the requirements of a normal form.

Certain conversions have preconditions on the structure of the sample buffer that are not captured (by design) by the type system. These are then checked before the conversion. Such checks can all be done in constant time and will not inspect the buffer content. You can perform these checks yourself when the conversion is not required at this moment but maybe still performed later.

```
pub fn in_bounds(self: &Self, channel: u8, x: u32, y: u32) -> bool
```

Check that the pixel and the channel index are in bounds.

An in-bound coordinate does not yet guarantee that the corresponding calculation of a buffer index does not overflow. However, if such a buffer large enough to hold all samples actually exists in memory, this porperty of course follows.

```
pub fn index(self: &Self, channel: u8, x: u32, y: u32) -> Option<usize>
```

Resolve the index of a particular sample.

‘None‘ if the index is outside the bounds or does not fit into a ‘usize‘.

```
pub fn index_ignoring_bounds(self: &Self, channel: usize, x: usize, y: usize) -> Option<usi
```

Get the theoretical position of sample (channel, x, y).

The 'check' is for overflow during index calculation, not that it is contained in the image. Two samples may return the same index, even when one of them is out of bounds. This happens when all strides are ‘0‘, i.e. the image is an arbitrarily large monochrome image.

```
pub fn in_bounds_index(self: &Self, c: u8, x: u32, y: u32) -> usize
```

Get an index provided it is inbouds.

Assumes that the image is backed by some sufficiently large buffer. Then computation can not overflow as we could represent the maximum coordinate. Since overflow is defined either way, this method can not be unsafe.

```
pub fn shrink_to(self: &mut Self, channels: u8, width: u32, height: u32)
```

Shrink the image to the minimum of current and given extents.

This does not modify the strides, so that the resulting sample buffer may have holes created by the shrinking operation. Shrinking could also lead to an non-aliasing image when samples had aliased each other before.

```
impl Send for SampleLayout

impl Sync for SampleLayout

impl Unpin for SampleLayout

impl UnwindSafe for SampleLayout

impl RefUnwindSafe for SampleLayout

impl Clone for SampleLayout

 fn clone(self: &Self) -> SampleLayout

impl Copy for SampleLayout

impl Eq for SampleLayout

impl PartialEq<SampleLayout> for SampleLayout

 fn eq(self: &Self, other: &SampleLayout) -> bool

 fn ne(self: &Self, other: &SampleLayout) -> bool

impl Debug for SampleLayout

 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Hash for SampleLayout

 fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl StructuralPartialEq for SampleLayout

impl StructuralEq for SampleLayout
```

1.90 Struct pub image::flat::View

```rust
pub struct View<Buffer, P: Pixel>
where
    Buffer: AsRef<[<P as >::Subpixel]>{
    // some fields omitted
}
```

A flat buffer that can be used as an image view.

This is a nearly trivial wrapper around a buffer but at least sanitizes by checking the buffer length first and constraining the pixel type.

Note that this does not eliminate panics as the `AsRef<[T]>` implementation of `Buffer` may be unreliable, i.e. return different buffers at different times. This of course is a non-issue for all common collections where the bounds check once must be enough.

1.90.1 Inner invariants

- For all indices inside bounds, the corresponding index is valid in the buffer
- `P::channel_count()` agrees with `self.inner.layout.channels`

Implementations

`impl<Buffer, P: Pixel> where Buffer:  AsRef<[<P as >::Subpixel]> View<Buffer, P>`

`pub fn into_inner(self:  Self) -> FlatSamples<Buffer>`

Take out the sample buffer.

Gives up the normalization invariants on the buffer format.

`pub fn flat(self:  &Self) -> &FlatSamples<Buffer>`

Get a reference on the inner sample descriptor.

There is no mutable counterpart as modifying the buffer format, including strides and lengths, could invalidate the accessibility invariants of the 'View'. It is not specified if the inner buffer is the same as the buffer of the image from which this view was created. It might have been truncated as an optimization.

`pub fn samples(self:  &Self) -> &Buffer`

Get a reference on the inner buffer.

There is no mutable counter part since it is not intended to allow you to reassign the buffer or otherwise change its size or properties.

`pub fn get_sample(self:  &Self, channel:  u8, x:  u32, y:  u32) -> Option<&<P as >::Subpixel>`

Get a reference to a selected subpixel if it is in-bounds.

This method will return 'None' when the sample is out-of-bounds. All errors that could occur due to overflow have been eliminated while construction the 'View'.

```
 pub fn get_mut_sample where Buffer:  AsMut<[<P as >::Subpixel]>(self:  &mut Self, channel:
u8, x:  u32, y:  u32) -> Option<&mut <P as >::Subpixel>
```

Get a mutable reference to a selected subpixel if it is in-bounds.

This is relevant only when constructed with 'FlatSamples::as_view_with_mut_samples'. This method will return 'None' when the sample is out-of-bounds. All errors that could occur due to overflow have been eliminated while construction the 'View'.

WARNING: Note that of course samples may alias, so that the mutable reference returned here can in fact modify more than the coordinate in the argument.

```
 pub fn min_length(self:  &Self) -> usize
```

Get the minimum length of a buffer such that all in-bounds samples have valid indices.

See 'FlatSamples::min_length'. This method will always succeed.

```
 pub fn image_slice(self:  &Self) -> &[<P as >::Subpixel]
```

Return the portion of the buffer that holds sample values.

While this can not fail–the validity of all coordinates has been validated during the conversion from 'FlatSamples'–the resulting slice may still contain holes.

```
 pub fn image_mut_slice where Buffer:  AsMut<[<P as >::Subpixel]>(self:  &mut Self) ->
&mut [<P as >::Subpixel]
```

Return the mutable portion of the buffer that holds sample values.

This is relevant only when constructed with 'FlatSamples::as_view_with_mut_samples'. While this can not fail–the validity of all coordinates has been validated during the conversion from 'FlatSamples'–the resulting slice may still contain holes.

```
 pub fn shrink_to(self:  &mut Self, width:  u32, height:  u32)
```

Shrink the inner image.

The new dimensions will be the minimum of the previous dimensions. Since the set of in-bounds pixels afterwards is a subset of the current ones, this is allowed on a 'View'. Note that you can not change the number of channels as an intrinsic property of 'P'.

```
 pub fn try_upgrade where Buffer:  AsMut<[<P as >::Subpixel]>(self:  Self) -> Result<ViewMut<Buf
P>, (Error,Self)>
```

Try to convert this into an image with mutable pixels.

The resulting image implements 'GenericImage' in addition to 'GenericImageView'. While this has mutable samples, it does not enforce that pixel can not alias and that samples are packed enough for a mutable pixel reference. This is slightly cheaper than the chain 'self.into_inner().as_view_mut()' and keeps the 'View' alive on failure.

"' # use image::RgbImage; # use image::Rgb; let mut buffer = RgbImage::new(480, 640).into_flat_samples(); let view = buffer.as_view_with_mut_samples::<Rgb<u8»().unwrap();

// Inspect some pixels, . . .

// Doesn't fail because it was originally an 'RgbImage'. let view_mut = view.try_upgrade().unwrap(); “‘

```
impl<Buffer, P> where Buffer:  Send, P: Send Send for View<Buffer, P>

impl<Buffer, P> where Buffer:  Sync, P: Sync Sync for View<Buffer, P>

impl<Buffer, P> where Buffer:  Unpin, P: Unpin Unpin for View<Buffer, P>

impl<Buffer, P> where Buffer:  UnwindSafe, P: UnwindSafe UnwindSafe for View<Buffer, P>

impl<Buffer, P> where Buffer:  RefUnwindSafe, P: RefUnwindSafe RefUnwindSafe for View<Buffer,
P>

impl<Buffer, P: Pixel> where Buffer:  AsRef<[<P as >::Subpixel]> GenericImageView for View<Buffe
P>

 type Pixel = P;

 type InnerImageView = Self;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn bounds(self:  &Self) -> (u32,u32,u32,u32)

 fn in_bounds(self:  &Self, x:  u32, y:  u32) -> bool

 fn get_pixel(self:  &Self, x:  u32, y:  u32) -> <Self as >::Pixel

 fn inner(self:  &Self) -> &Self

impl<Buffer:  $crate::clone::Clone, P: $crate::clone::Clone + Pixel> where Buffer:  AsRef<[<P
as >::Subpixel]> Clone for View<Buffer, P>

 fn clone(self:  &Self) -> View<Buffer, P>

impl<Buffer:  $crate::fmt::Debug, P: $crate::fmt::Debug + Pixel> where Buffer:  AsRef<[<P
as >::Subpixel]> Debug for View<Buffer, P>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

1.91 Struct pub image::flat::ViewMut

```
pub struct ViewMut<Buffer, P: Pixel>
where
    Buffer: AsMut<[<P as >::Subpixel]>{
    // some fields omitted
}
```

A mutable owning version of a flat buffer.

While this wraps a buffer similar to `ImageBuffer`, this is mostly intended as a utility. The library endorsed normalized representation is still `ImageBuffer`. Also, the implementation of `AsMut<[P::Subpixel]>` must always yield the same buffer. Therefore there is no public way to construct this with an owning buffer.

1.91.1 Inner invariants

- For all indices inside bounds, the corresponding index is valid in the buffer
- There is no aliasing of samples
- The samples are packed, i.e. `self.inner.layout.sample_stride == 1`
- `P::channel_count()` agrees with `self.inner.layout.channels`

Implementations

```
impl<Buffer, P: Pixel> where Buffer:  AsMut<[<P as >::Subpixel]> ViewMut<Buffer, P>
```

```
 pub fn into_inner(self:  Self) -> FlatSamples<Buffer>
```

Take out the sample buffer.

Gives up the normalization invariants on the buffer format.

```
 pub fn flat(self:  &Self) -> &FlatSamples<Buffer>
```

Get a reference on the sample buffer descriptor.

There is no mutable counterpart as modifying the buffer format, including strides and lengths, could invalidate the accessibility invariants of the 'View'. It is not specified if the inner buffer is the same as the buffer of the image from which this view was created. It might have been truncated as an optimization.

```
 pub fn samples(self:  &Self) -> &Buffer
```

Get a reference on the inner buffer.

There is no mutable counter part since it is not intended to allow you to reassign the buffer or otherwise change its size or properties. However, its contents can be accessed mutable through a slice with 'image_mut_slice'.

```
 pub fn min_length(self:  &Self) -> usize
```

Get the minimum length of a buffer such that all in-bounds samples have valid indices.

See 'FlatSamples::min_length'. This method will always succeed.

```
 pub fn get_sample where Buffer:  AsRef<[<P as >::Subpixel]>(self:  &Self, channel:  u8,
x:  u32, y:  u32) -> Option<&<P as >::Subpixel>
```

Get a reference to a selected subpixel.

This method will return 'None' when the sample is out-of-bounds. All errors that could occur due to overflow have been eliminated while construction the 'View'.

```
 pub fn get_mut_sample(self:  &mut Self, channel:  u8, x:  u32, y:  u32) -> Option<&mut
<P as >::Subpixel>
```

Get a mutable reference to a selected sample.

This method will return 'None' when the sample is out-of-bounds. All errors that could occur due to overflow have been eliminated while construction the 'View'.

```
 pub fn image_slice where Buffer:  AsRef<[<P as >::Subpixel]>(self:  &Self) -> &[<P as
>::Subpixel]
```

Return the portion of the buffer that holds sample values.

While this can not fail–the validity of all coordinates has been validated during the conversion from 'FlatSamples'–the resulting slice may still contain holes.

```
 pub fn image_mut_slice(self:  &mut Self) -> &mut [<P as >::Subpixel]
```

Return the mutable buffer that holds sample values.

```
 pub fn shrink_to(self:  &mut Self, width:  u32, height:  u32)
```

Shrink the inner image.

The new dimensions will be the minimum of the previous dimensions. Since the set of in-bounds pixels afterwards is a subset of the current ones, this is allowed on a 'View'. Note that you can not change the number of channels as an intrinsic property of 'P'.

```
impl<Buffer, P> where Buffer:  Send, P: Send Send for ViewMut<Buffer, P>

impl<Buffer, P> where Buffer:  Sync, P: Sync Sync for ViewMut<Buffer, P>

impl<Buffer, P> where Buffer:  Unpin, P: Unpin Unpin for ViewMut<Buffer, P>

impl<Buffer, P> where Buffer:  UnwindSafe, P: UnwindSafe UnwindSafe for ViewMut<Buffer,
P>

impl<Buffer, P> where Buffer:  RefUnwindSafe, P: RefUnwindSafe RefUnwindSafe for ViewMut<Buffer,
P>

impl<Buffer, P: Pixel> where Buffer:  AsMut<[<P as >::Subpixel]> + AsRef<[<P as >::Subpixel]>
GenericImageView for ViewMut<Buffer, P>

 type Pixel = P;

 type InnerImageView = Self;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn bounds(self:  &Self) -> (u32,u32,u32,u32)

 fn in_bounds(self:  &Self, x:  u32, y:  u32) -> bool

 fn get_pixel(self:  &Self, x:  u32, y:  u32) -> <Self as >::Pixel
 fn inner(self:  &Self) -> &Self

impl<Buffer, P: Pixel> where Buffer:  AsMut<[<P as >::Subpixel]> + AsRef<[<P as >::Subpixel]>
GenericImage for ViewMut<Buffer, P>

 type InnerImage = Self;

 fn get_pixel_mut(self:  &mut Self, x:  u32, y:  u32) -> &mut <Self as >::Pixel

 fn put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  <Self as >::Pixel)

 fn blend_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  <Self as >::Pixel)

 fn inner_mut(self:  &mut Self) -> &mut Self

impl<Buffer:  $crate::clone::Clone, P: $crate::clone::Clone + Pixel> where Buffer:  AsMut<[<P
as >::Subpixel]> Clone for ViewMut<Buffer, P>

 fn clone(self:  &Self) -> ViewMut<Buffer, P>

impl<Buffer:  $crate::fmt::Debug, P: $crate::fmt::Debug + Pixel> where Buffer:  AsMut<[<P
as >::Subpixel]> Debug for ViewMut<Buffer, P>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

1.92 Enum pub image::flat::Error

```
pub enum image::flat::Error {
    TooLarge,
    NormalFormRequired,
    WrongColor,
}
```

Denotes invalid flat sample buffers when trying to convert to stricter types.

The biggest use case being `ImageBuffer` which expects closely packed samples in a row major matrix representation. But this error type may be resused for other import functions. A more versatile user may also try to correct the underlying representation depending on the error variant.

Variants

`TooLarge` The represented image was too large.

The optional value denotes a possibly accepted maximal bound.

`NormalFormRequired` The represented image can not use this representation.

Has an additional value of the normalized form that would be accepted.

`WrongColor` The color format did not match the channel count.

In some cases you might be able to fix this by lowering the reported pixel count of the buffer without touching the strides.

In very special circumstances you *may* do the opposite. This is **VERY** dangerous but not directly memory unsafe although that will likely alias pixels. One scenario is when you want to construct an `Rgba` image but have only 3 bytes per pixel and for some reason don't care about the value of the alpha channel even though you need `Rgba`.

Implementations

```
impl Send for Error

impl Sync for Error

impl Unpin for Error

impl UnwindSafe for Error

impl RefUnwindSafe for Error

impl From<Error> for ImageError

 fn from(error:  Error) -> ImageError

impl Clone for Error

 fn clone(self:  &Self) -> Error

impl Copy for Error

impl Eq for Error

impl PartialEq<Error> for Error

 fn eq(self:  &Self, other:  &Error) -> bool

 fn ne(self:  &Self, other:  &Error) -> bool

impl Debug for Error

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Display for Error

 fn fmt(self:  &Self, f:  &mut fmt::Formatter<'_>) -> fmt::Result

impl Hash for Error

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl StructuralPartialEq for Error

impl StructuralEq for Error

impl Error for Error
```

1.93 Enum pub image::flat::NormalForm

```
pub enum image::flat::NormalForm {
    Unaliased,
    PixelPacked,
    ImagePacked,
    RowMajorPacked,
    ColumnMajorPacked,
}
```

Different normal forms of buffers.

A normal form is an unaliased buffer with some additional constraints. The ÌmageBuffer uses row major form with packed samples.

Variants

`Unaliased` No pixel aliases another.

Unaliased also guarantees that all index calculations in the image bounds using `dim_index*dim_stride` (such as `x*width_stride + y*height_stride`) do not overflow.

`PixelPacked` At least pixels are packed.

Images of these types can wrap `[T]`-slices into the standard color types. This is a precondition for `GenericImage` which requires by-reference access to pixels.

`ImagePacked` All samples are packed.

This is orthogonal to `PixelPacked`. It requires that there are no holes in the image but it is not necessary that the pixel samples themselves are adjacent. An example of this behaviour is a planar image layout.

`RowMajorPacked` The samples are in row-major form and all samples are packed.

In addition to `PixelPacked` and `ImagePacked` this also asserts that the pixel matrix is in row-major form.

`ColumnMajorPacked` The samples are in column-major form and all samples are packed.

In addition to `PixelPacked` and `ImagePacked` this also asserts that the pixel matrix is in column-major form.

Implementations

`impl Send for NormalForm`

`impl Sync for NormalForm`

`impl Unpin for NormalForm`

`impl UnwindSafe for NormalForm`

`impl RefUnwindSafe for NormalForm`

`impl Clone for NormalForm`

```
 fn clone(self: &Self) -> NormalForm
```

```
impl Copy for NormalForm
```

```
impl Eq for NormalForm
```

```
impl PartialEq<NormalForm> for NormalForm
```

```
 fn eq(self: &Self, other: &NormalForm) -> bool
```

```
impl PartialOrd<NormalForm> for NormalForm
```

```
 fn partial_cmp(self: &Self, other: &Self) -> Option<cmp::Ordering>
```

Compares the logical preconditions.

'a < b' if the normal form 'a' has less preconditions than 'b'.

```
impl Debug for NormalForm
```

```
 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Hash for NormalForm
```

```
 fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()
```

```
impl StructuralPartialEq for NormalForm
```

```
impl StructuralEq for NormalForm
```

1.94 Module pub image::codecs

- image::codecs::bmp
- image::codecs::dds
- image::codecs::dxt
- image::codecs::farbfeld
- image::codecs::gif
- image::codecs::hdr
- image::codecs::ico
- image::codecs::jpeg
- image::codecs::png
- image::codecs::pnm
- image::codecs::tga
- image::codecs::tiff
- image::codecs::webp

1.95 Module pub image::codecs::bmp

- image::codecs::bmp::BmpDecoder
- image::codecs::bmp::BmpEncoder
- image::codecs::bmp::BMPEncoder

1.96 Struct pub image::codecs::bmp::BmpDecoder

```
pub struct BmpDecoder<R>{
    // some fields omitted
}
```

A bmp decoder

Implementations

`impl<R: Read + Seek> BmpDecoder<R>`

`pub fn new(reader: R) -> ImageResult<BmpDecoder<R»`

Create a new decoder that decodes from the stream "'r"'

`impl<R> where R: Send Send for BmpDecoder<R>`

`impl<R> where R: Sync Sync for BmpDecoder<R>`

`impl<R> where R: Unpin Unpin for BmpDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for BmpDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for BmpDecoder<R>`

`impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for BmpDecoder<R>`

`type Reader = BmpReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

`impl<'a, R: 'a + Read + Seek> ImageDecoderExt<'a> for BmpDecoder<R>`

`fn read_rect_with_progress<F: Fn>( self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F ) -> ImageResult<()>`

1.97 Struct pub image::codecs::bmp::BmpEncoder

```
pub struct BmpEncoder<'a, W: 'a>{
    // some fields omitted
}
```

The representation of a BMP encoder.

Implementations

```
impl<'a, W: Write + 'a> BmpEncoder<'a, W>
```

```
 pub fn new(w:  &'a mut W) -> Self
```

Create a new encoder that writes its output to "'w"'.

```
 pub fn encode( self:  &mut Self, image:  &[u8], width:  u32, height:  u32, c:  color::ColorType
) -> ImageResult<()>
```

Encodes the image "'image"' that has dimensions "'width"' and "'height"' and "'ColorType"' "'c"'.

```
impl<'a, W> where W: Send Send for BmpEncoder<'a, W>
```

```
impl<'a, W> where W: Sync Sync for BmpEncoder<'a, W>
```

```
impl<'a, W> Unpin for BmpEncoder<'a, W>
```

```
impl<'a, W> !UnwindSafe for BmpEncoder<'a, W>
```

```
impl<'a, W> where W: RefUnwindSafe RefUnwindSafe for BmpEncoder<'a, W>
```

```
impl<'a, W: Write> ImageEncoder for BmpEncoder<'a, W>
```

```
 fn write_image( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  color::Co
) -> ImageResult<()>
```

1.98 Typedef pub image::codecs::bmp::BMPEncoder

```
pub type BMPEncoder = BmpEncoder<'a, W>;
```

BMP Encoder

An alias of `struct.BmpEncoder.html`.

TODO: remove

1.99 Module pub image::codecs::dds

- image::codecs::dds::DdsDecoder

1.100 Struct pub image::codecs::dds::DdsDecoder

```
pub struct DdsDecoder<R: Read>{
    // some fields omitted
}
```

The representation of a DDS decoder

Implementations

```
impl<R: Read> DdsDecoder<R>
```

```
 pub fn new(mut r:  R) -> ImageResult<Self>
```

Create a new decoder that decodes from the stream 'r'

```
impl<R> where R: Send Send for DdsDecoder<R>
```

```
impl<R> where R: Sync Sync for DdsDecoder<R>
```

```
impl<R> where R: Unpin Unpin for DdsDecoder<R>
```

```
impl<R> where R: UnwindSafe UnwindSafe for DdsDecoder<R>
```

```
impl<R> where R: RefUnwindSafe RefUnwindSafe for DdsDecoder<R>
```

```
impl<'a, R: 'a + Read> ImageDecoder<'a> for DdsDecoder<R>
```

```
 type Reader = DxtReader<R>;
```

```
 fn dimensions(self:  &Self) -> (u32,u32)
```

```
 fn color_type(self:  &Self) -> ColorType
```

```
 fn scanline_bytes(self:  &Self) -> u64
```

```
 fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>
```

```
 fn read_image(self:  Self, buf:  &mut [u8]) -> ImageResult<()>
```

1.101 Module pub image::codecs::dxt

- image::codecs::dxt::DXTVariant
- image::codecs::dxt::DxtVariant
- image::codecs::dxt::DxtDecoder
- image::codecs::dxt::DxtReader
- image::codecs::dxt::DXTReader
- image::codecs::dxt::DxtEncoder
- image::codecs::dxt::DXTEncoder

1.102 Enum pub image::codecs::dxt::DXTVariant

```
pub enum image::codecs::dxt::DXTVariant {
    DXT1,
    DXT3,
    DXT5,
}
```

What version of DXT compression are we using? Note that DXT2 and DXT4 are left away as they're just DXT3 and DXT5 with premultiplied alpha

DEPRECATED: The name of this enum will be changed to `type.DxtVariant.html`.

TODO: rename to `type.DxtVariant.html`

Variants

`DXT1` The DXT1 format. 48 bytes of RGB data in a 4x4 pixel square is compressed into an 8 byte block of DXT1 data

`DXT3` The DXT3 format. 64 bytes of RGBA data in a 4x4 pixel square is compressed into a 16 byte block of DXT3 data

`DXT5` The DXT5 format. 64 bytes of RGBA data in a 4x4 pixel square is compressed into a 16 byte block of DXT5 data

Implementations

`impl DXTVariant`

`pub fn color_type(self: Self) -> ColorType`

Returns the color type that is stored in this DXT variant

`impl Send for DXTVariant`

`impl Sync for DXTVariant`

`impl Unpin for DXTVariant`

`impl UnwindSafe for DXTVariant`

`impl RefUnwindSafe for DXTVariant`

`impl Clone for DXTVariant`

`fn clone(self: &Self) -> DXTVariant`

`impl Copy for DXTVariant`

`impl Eq for DXTVariant`

`impl PartialEq<DXTVariant> for DXTVariant`

`fn eq(self: &Self, other: &DXTVariant) -> bool`

`impl Debug for DXTVariant`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for DXTVariant`

`impl StructuralEq for DXTVariant`

1.103 Typedef pub image::codecs::dxt::DxtVariant

```
pub type DxtVariant = DXTVariant;
```

DXT compression version.

An alias of `enum.DXTVariant.html`.

TODO: remove when `enum.DXTVariant.html` is renamed.

1.104 Struct pub image::codecs::dxt::DxtDecoder

```
pub struct DxtDecoder<R: Read>{
    // some fields omitted
}
```

DXT decoder

Implementations

`impl<R: Read> DxtDecoder<R>`

`pub fn new( r: R, width: u32, height: u32, variant: DXTVariant ) -> Result<DxtDecoder<R>, ImageError>`

Create a new DXT decoder that decodes from the stream "'r"'. As DXT is often stored as raw buffers with the width/height somewhere else the width and height of the image need to be passed in "'width"' and "'height"', as well as the DXT variant in "'variant"'. width and height are required to be powers of 2 and at least 4. otherwise an error will be returned

`impl<R> where R: Send Send for DxtDecoder<R>`

`impl<R> where R: Sync Sync for DxtDecoder<R>`

`impl<R> where R: Unpin Unpin for DxtDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for DxtDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for DxtDecoder<R>`

`impl<'a, R: 'a + Read> ImageDecoder<'a> for DxtDecoder<R>`

`type Reader = DxtReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn scanline_bytes(self: &Self) -> u64`

`fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

impl<'a, R: 'a + Read + Seek> ImageDecoderExt<'a> for DxtDecoder<R>

fn read_rect_with_progress<F: Fn>(self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F) -> ImageResult<()>

1.105 Struct pub image::codecs::dxt::DxtReader

```
pub struct DxtReader<R: Read>{
    // some fields omitted
}
```

DXT reader

Implementations

impl<R> where R: Send Send for DxtReader<R>

impl<R> where R: Sync Sync for DxtReader<R>

impl<R> where R: Unpin Unpin for DxtReader<R>

impl<R> where R: UnwindSafe UnwindSafe for DxtReader<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for DxtReader<R>

impl<R: Read> Read for DxtReader<R>

fn read(self: &mut Self, buf: &mut [u8]) -> io::Result<usize>

1.106 Typedef pub image::codecs::dxt::DXTReader

```
pub type DXTReader = DxtReader<R>;
```

DXT reader

An alias of `struct.DxtReader.html`.

TODO: remove

1.107 Struct pub image::codecs::dxt::DxtEncoder

```
pub struct DxtEncoder<W: Write>{
    // some fields omitted
}
```

DXT encoder

Implementations

impl<W: Write> DxtEncoder<W>

```
pub fn new(w:  W) -> DxtEncoder<W>
```

Create a new encoder that writes its output to "'w"'

```
pub fn encode( mut self:  Self, data:  &[u8], width:  u32, height:  u32, variant:  DXTVariant
) -> ImageResult<()>
```

Encodes the image data "'data"' that has dimensions "'width"' and "'height"' in "'DXTVariant"' "'variant"' data is assumed to be in variant.color_type()

`impl<W> where W: Send Send for DxtEncoder<W>`

`impl<W> where W: Sync Sync for DxtEncoder<W>`

`impl<W> where W: Unpin Unpin for DxtEncoder<W>`

`impl<W> where W: UnwindSafe UnwindSafe for DxtEncoder<W>`

`impl<W> where W: RefUnwindSafe RefUnwindSafe for DxtEncoder<W>`

1.108 Typedef pub image::codecs::dxt::DXTEncoder

```rust
pub type DXTEncoder = DxtEncoder<W>;
```

DXT encoder

An alias of `struct.DxtEncoder.html`.

TODO: remove

1.109 Module pub image::codecs::farbfeld

- image::codecs::farbfeld::FarbfeldReader
- image::codecs::farbfeld::FarbfeldDecoder
- image::codecs::farbfeld::FarbfeldEncoder

1.110 Struct pub image::codecs::farbfeld::FarbfeldReader

```rust
pub struct FarbfeldReader<R: Read>{
    // some fields omitted
}
```

farbfeld Reader

Implementations

`impl<R> where R: Send Send for FarbfeldReader<R>`

`impl<R> where R: Sync Sync for FarbfeldReader<R>`

`impl<R> where R: Unpin Unpin for FarbfeldReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for FarbfeldReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for FarbfeldReader<R>`

`impl<R: Read> Read for FarbfeldReader<R>`

`fn read(self: &mut Self, mut buf: &mut [u8]) -> io::Result<usize>`

`impl<R: Read + Seek> Seek for FarbfeldReader<R>`

`fn seek(self: &mut Self, pos: SeekFrom) -> io::Result<u64>`

1.111 Struct pub image::codecs::farbfeld::FarbfeldDecoder

```
pub struct FarbfeldDecoder<R: Read>{
    // some fields omitted
}
```

farbfeld decoder

Implementations

`impl<R: Read> FarbfeldDecoder<R>`

`pub fn new(r: R) -> ImageResult<FarbfeldDecoder<R»`

Creates a new decoder that decodes from the stream “‘r“‘

`impl<R> where R: Send Send for FarbfeldDecoder<R>`

`impl<R> where R: Sync Sync for FarbfeldDecoder<R>`

`impl<R> where R: Unpin Unpin for FarbfeldDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for FarbfeldDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for FarbfeldDecoder<R>`

`impl<’a, R: ’a + Read> ImageDecoder<’a> for FarbfeldDecoder<R>`

`type Reader = FarbfeldReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn scanline_bytes(self: &Self) -> u64`

`impl<’a, R: ’a + Read + Seek> ImageDecoderExt<’a> for FarbfeldDecoder<R>`

`fn read_rect_with_progress<F: Fn>( self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F ) -> ImageResult<()>`

1.112 Struct pub image::codecs::farbfeld::FarbfeldEncoder

```
pub struct FarbfeldEncoder<W: Write>{
    // some fields omitted
}
```

farbfeld encoder

Implementations

`impl<W: Write> FarbfeldEncoder<W>`

`pub fn new(w: W) -> FarbfeldEncoder<W>`

Create a new encoder that writes its output to “‘w“‘

`pub fn encode(self: Self, data: &[u8], width: u32, height: u32) -> ImageResult<()>`

Encodes the image “‘data“‘ (native endian) that has dimensions “‘width“‘ and “‘height“‘

`impl<W> where W: Send Send for FarbfeldEncoder<W>`

`impl<W> where W: Sync Sync for FarbfeldEncoder<W>`

`impl<W> where W: Unpin Unpin for FarbfeldEncoder<W>`

`impl<W> where W: UnwindSafe UnwindSafe for FarbfeldEncoder<W>`

`impl<W> where W: RefUnwindSafe RefUnwindSafe for FarbfeldEncoder<W>`

`impl<W: Write> ImageEncoder for FarbfeldEncoder<W>`

```
fn write_image( self: Self, buf: &[u8], width: u32, height: u32, color_type: ColorType
) -> ImageResult<()>
```

1.113 Module pub image::codecs::gif

- image::codecs::gif::GifDecoder
- image::codecs::gif::GifReader
- image::codecs::gif::Repeat
- image::codecs::gif::GifEncoder
- image::codecs::gif::Encoder

1.114 Struct pub image::codecs::gif::GifDecoder

```
pub struct GifDecoder<R: Read>{
    // some fields omitted
}
```

GIF decoder

Implementations

`impl<R: Read> GifDecoder<R>`

` pub fn new(r:  R) -> ImageResult<GifDecoder<R»`

Creates a new decoder that decodes the input steam “‘r“‘

`impl<R> where R: Send Send for GifDecoder<R>`

`impl<R> !Sync for GifDecoder<R>`

`impl<R> where R: Unpin Unpin for GifDecoder<R>`

`impl<R> !UnwindSafe for GifDecoder<R>`

`impl<R> !RefUnwindSafe for GifDecoder<R>`

`impl<’a, R: ’a + Read> ImageDecoder<’a> for GifDecoder<R>`

` type Reader = GifReader<R>;`

` fn dimensions(self:  &Self) -> (u32,u32)`

` fn color_type(self:  &Self) -> ColorType`

` fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>`

` fn read_image(mut self:  Self, buf:  &mut [u8]) -> ImageResult<()>`

`impl<’a, R: Read + ’a> AnimationDecoder<’a> for GifDecoder<R>`

` fn into_frames(self:  Self) -> animation::Frames<’a>`

1.115 Struct pub image::codecs::gif::GifReader

```
pub struct GifReader<R>(
    // some fields omitted
)
```

Wrapper struct around a `Cursor<Vec<u8»`

Implementations

`impl<R> where R: Send Send for GifReader<R>`

`impl<R> where R: Sync Sync for GifReader<R>`

`impl<R> where R: Unpin Unpin for GifReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for GifReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for GifReader<R>`

`impl<R> Read for GifReader<R>`

` fn read(self:  &mut Self, buf:  &mut [u8]) -> io::Result<usize>`

` fn read_to_end(self:  &mut Self, buf:  &mut Vec<u8>) -> io::Result<usize>`

1.116 Enum pub image::codecs::gif::Repeat

```
pub enum image::codecs::gif::Repeat {
    Finite,
    Infinite,
}
```

Number of repetitions for a GIF animation

Variants

`Finite` Finite number of repetitions

`Infinite` Looping GIF

Implementations

`impl Send for Repeat`

`impl Sync for Repeat`

`impl Unpin for Repeat`

`impl UnwindSafe for Repeat`

`impl RefUnwindSafe for Repeat`

`impl Clone for Repeat`

`fn clone(self: &Self) -> Repeat`

`impl Copy for Repeat`

1.117 Struct pub image::codecs::gif::GifEncoder

```
pub struct GifEncoder<W: Write>{
    // some fields omitted
}
```

GIF encoder.

Implementations

`impl<W: Write> GifEncoder<W>`

`pub fn new(w: W) -> GifEncoder<W>`

Creates a new GIF encoder.

`pub fn new_with_speed(w: W, speed: i32) -> GifEncoder<W>`

Create a new GIF encoder, and has the speed parameter 'speed'. See ['Frame::from_rgba_speed'](/gif/struct.Frame
for more information.

`pub fn set_repeat(self: &mut Self, repeat: Repeat) -> ImageResult<()>`

Set the repeat behaviour of the encoded GIF

`pub fn encode( self: &mut Self, data: &[u8], width: u32, height: u32, color: ColorType ) -> ImageResult<()>`

Encode a single image.

`pub fn encode_frame(self: &mut Self, img_frame: animation::Frame) -> ImageResult<()>`

Encode one frame of animation.

`pub fn encode_frames<F> where F: IntoIterator<Item=animation::Frame>(self: &mut Self, frames: F) -> ImageResult<()>`

Encodes Frames. Consider using 'try_encode_frames' instead to encode an 'animation::Frames' like iterator.

`pub fn try_encode_frames<F> where F: IntoIterator<Item=ImageResult<animation::Frame»(self: &mut Self, frames: F) -> ImageResult<()>`

Try to encode a collection of 'ImageResult<animation::Frame>' objects. Use this function to encode an 'animation::Frames' like iterator. Whenever an 'Err' item is encountered, that value is returned without further actions.

`impl<W> where W: Send Send for GifEncoder<W>`

`impl<W> where W: Sync Sync for GifEncoder<W>`

`impl<W> where W: Unpin Unpin for GifEncoder<W>`

`impl<W> where W: UnwindSafe UnwindSafe for GifEncoder<W>`

`impl<W> where W: RefUnwindSafe RefUnwindSafe for GifEncoder<W>`

1.118 Typedef pub image::codecs::gif::Encoder

```
pub type Encoder = GifEncoder<W>;
```

GIF encoder

An alias of `struct.GifEncoder.html`.

TODO: remove

1.119 Module pub image::codecs::hdr

- image::codecs::hdr::HdrAdapter
- image::codecs::hdr::HDRAdapter

- image::codecs::hdr::HdrReader
- image::codecs::hdr::SIGNATURE
- image::codecs::hdr::HdrDecoder
- image::codecs::hdr::Rgbe8Pixel
- image::codecs::hdr::RGBE8Pixel
- image::codecs::hdr::rgbe8
- image::codecs::hdr::HdrImageDecoderIterator
- image::codecs::hdr::HDRImageDecoderIterator
- image::codecs::hdr::HdrMetadata
- image::codecs::hdr::HDRMetadata
- image::codecs::hdr::read_raw_file
- image::codecs::hdr::HdrEncoder
- image::codecs::hdr::HDREncoder
- image::codecs::hdr::to_rgbe8

1.120 Struct pub image::codecs::hdr::HdrAdapter

```
pub struct HdrAdapter<R: BufRead>{
    // some fields omitted
}
```

Adapter to conform to `ImageDecoder` trait

Implementations

`impl<R: BufRead> HdrAdapter<R>`

`pub fn new(r: R) -> ImageResult<HdrAdapter<R»`

Creates adapter

`pub fn new_nonstrict(r: R) -> ImageResult<HdrAdapter<R»`

Allows reading old Radiance HDR images

`impl<R> where R: Send Send for HdrAdapter<R>`

`impl<R> where R: Sync Sync for HdrAdapter<R>`

`impl<R> where R: Unpin Unpin for HdrAdapter<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrAdapter<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrAdapter<R>`

`impl<'a, R: 'a + BufRead> ImageDecoder<'a> for HdrAdapter<R>`

`type Reader = HdrReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType` `fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

`impl<'a, R: 'a + BufRead + Seek> ImageDecoderExt<'a> for HdrAdapter<R>`

`fn read_rect_with_progress<F: Fn>( self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F ) -> ImageResult<()>`

`impl<R: $crate::fmt::Debug + BufRead> Debug for HdrAdapter<R>`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.121 Typedef pub image::codecs::hdr::HDRAdapter

```
pub type HDRAdapter = HdrAdapter<R>;
```

HDR Adapter

An alias of `struct.HdrAdapter.html`.

TODO: remove

1.122 Struct pub image::codecs::hdr::HdrReader

```
pub struct HdrReader<R>(
    // some fields omitted
)
```

Wrapper struct around a `Cursor<Vec<u8»`

Implementations

`impl<R> where R: Send Send for HdrReader<R>`

`impl<R> where R: Sync Sync for HdrReader<R>`

`impl<R> where R: Unpin Unpin for HdrReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrReader<R>`

`impl<R> Read for HdrReader<R>`

`fn read(self: &mut Self, buf: &mut [u8]) -> io::Result<usize>`

`fn read_to_end(self: &mut Self, buf: &mut Vec<u8>) -> io::Result<usize>`

1.123 Constant pub image::codecs::hdr::SIGNATURE

```
pub const SIGNATURE: &[u8] = b"#?RADIANCE";
```

Radiance HDR file signature

1.124 Struct pub image::codecs::hdr::HdrDecoder

```
pub struct HdrDecoder<R>{
    // some fields omitted
}
```

An Radiance HDR decoder

Implementations

`impl<R: BufRead> HdrDecoder<R>`

`pub fn new(reader: R) -> ImageResult<HdrDecoder<R»`

Reads Radiance HDR image header from stream "'r"' if the header is valid, creates HdrDecoder strict mode is enabled

`pub fn with_strictness(mut reader: R, strict: bool) -> ImageResult<HdrDecoder<R»`

Reads Radiance HDR image header from stream "'reader"', if the header is valid, creates "'HdrDecoder"'.

strict enables strict mode

Warning! Reading wrong file in non-strict mode could consume file size worth of memory in the process.

`pub fn metadata(self: &Self) -> HdrMetadata`

Returns file metadata. Refer to "'HDRMetadata"' for details.

`pub fn read_image_native(mut self: Self) -> ImageResult<Vec<Rgbe8Pixel»`

Consumes decoder and returns a vector of RGBE8 pixels

`pub fn read_image_transform<T: Send, F: Send + Sync + Fn>(mut self: Self, f: F, output_slice:`
`&mut [T]) -> ImageResult<()>`

Consumes decoder and returns a vector of transformed pixels

`pub fn read_image_ldr(self: Self) -> ImageResult<Vec<Rgb<u8»>`

Consumes decoder and returns a vector of Rgb<u8> pixels. scale = 1, gamma = 2.2

`pub fn read_image_hdr(self: Self) -> ImageResult<Vec<Rgb<f32»>`

Consumes decoder and returns a vector of Rgb<f32> pixels.

`impl<R> where R: Send Send for HdrDecoder<R>`

`impl<R> where R: Sync Sync for HdrDecoder<R>`

`impl<R> where R: Unpin Unpin for HdrDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrDecoder<R>`

`impl<R: BufRead> IntoIterator for HdrDecoder<R>`

`type Item = ImageResult<Rgbe8Pixel>;`

`type IntoIter = HdrImageDecoderIterator<R>;`

`fn into_iter(self:  Self) -> <Self as >::IntoIter`

`impl<R: $crate::fmt::Debug> Debug for HdrDecoder<R>`

`fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.125 Struct pub image::codecs::hdr::Rgbe8Pixel

```
#[repr(C)]
pub struct Rgbe8Pixel{
    pub c: [u8; 3],
    pub e: u8,
}
```

Refer to wikipedia

Fields

`c:  [u8; 3]` Color components

`e:  u8` Exponent

Implementations

`impl Rgbe8Pixel`

`pub fn to_hdr(self:  Self) -> Rgb<f32>`

Converts "'RGBE8Pixel"' into "'Rgb<f32>"' linearly

`pub fn to_ldr<T: Primitive + Zero>(self:  Self) -> Rgb<T>`

Converts "'RGBE8Pixel"' into "'Rgb<T>"' with scale=1 and gamma=2.2

color_ldr = (color_hdr*scale)<sup>gamma</sup>

# Panic

Panics when “‘T::max_value()“‘ cannot be represented as f32.

```
 pub fn to_ldr_scale_gamma<T: Primitive + Zero>(self:  Self, scale:  f32, gamma:  f32)
-> Rgb<T>
```

Converts RGBE8Pixel into Rgb<T> using provided scale and gamma

color_ldr = (color_hdr*scale)<sup>gamma</sup>

# Panic

Panics when T::max_value() cannot be represented as f32. Panics when scale or gamma is NaN

```
impl Send for Rgbe8Pixel
```

```
impl Sync for Rgbe8Pixel
```

```
impl Unpin for Rgbe8Pixel
```

```
impl UnwindSafe for Rgbe8Pixel
```

```
impl RefUnwindSafe for Rgbe8Pixel
```

```
impl Clone for Rgbe8Pixel
```

```
 fn clone(self:  &Self) -> Rgbe8Pixel
```

```
impl Copy for Rgbe8Pixel
```

```
impl Default for Rgbe8Pixel
```

```
 fn default() -> Rgbe8Pixel
```

```
impl Eq for Rgbe8Pixel
```

```
impl PartialEq<Rgbe8Pixel> for Rgbe8Pixel
```

```
 fn eq(self:  &Self, other:  &Rgbe8Pixel) -> bool
```

```
 fn ne(self:  &Self, other:  &Rgbe8Pixel) -> bool
```

```
impl Debug for Rgbe8Pixel
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl StructuralPartialEq for Rgbe8Pixel
```

```
impl StructuralEq for Rgbe8Pixel
```

1.126 Typedef pub image::codecs::hdr::RGBE8Pixel

```
pub type RGBE8Pixel = Rgbe8Pixel;
```

Refer to wikipedia

An alias of `struct.Rgbe8Pixel.html`.

TODO: remove

1.127 Function pub image::codecs::hdr::rgbe8

```
pub fn rgbe8(r: u8, g: u8, b: u8, e: u8) -> Rgbe8Pixel
```

Creates `RGBE8Pixel` from components

1.128 Struct pub image::codecs::hdr::HdrImageDecoderIterator

```
pub struct HdrImageDecoderIterator<R: BufRead>{
    // some fields omitted
}
```

Scanline buffered pixel by pixel iterator

Implementations

`impl<R> where R: Send Send for HdrImageDecoderIterator<R>`

`impl<R> where R: Sync Sync for HdrImageDecoderIterator<R>`

`impl<R> where R: Unpin Unpin for HdrImageDecoderIterator<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrImageDecoderIterator<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrImageDecoderIterator<R>`

`impl<R: BufRead> ExactSizeIterator for HdrImageDecoderIterator<R>`

`impl<R: BufRead> Iterator for HdrImageDecoderIterator<R>`

`type Item = ImageResult<Rgbe8Pixel>;`

`fn next(self: &mut Self) -> Option«Self as >::Item>`

`fn size_hint(self: &Self) -> (usize,Option<usize>)`

1.129 Typedef pub image::codecs::hdr::HDRImageDecoderIterator

```
pub type HDRImageDecoderIterator = HdrImageDecoderIterator<R>;
```

Scanline buffered pixel by pixel iterator

An alias of `struct.HdrImageDecoderIterator.html`.

TODO: remove

1.130 Struct pub image::codecs::hdr::HdrMetadata

```
pub struct HdrMetadata{
    pub width: u32,
    pub height: u32,
    pub orientation: ((i8,i8),(i8,i8)),
    pub exposure: Option<f32>,
    pub color_correction: Option<(f32,f32,f32)>,
    pub pixel_aspect_ratio: Option<f32>,
    pub custom_attributes: Vec<(String,String)>,
}
```

Metadata for Radiance HDR image

Fields

`width: u32` Width of decoded image. It could be either scanline length, or scanline count, depending on image orientation.

`height: u32` Height of decoded image. It depends on orientation too.

`orientation: ((i8,i8),(i8,i8))` Orientation matrix. For standard orientation it is ((1,0),(0,1)) - left to right, top to bottom. First pair tells how resulting pixel coordinates change along a scanline. Second pair tells how they change from one scanline to the next.

`exposure: Option<f32>` Divide color values by exposure to get to get physical radiance in watts/steradian/m<sup>2</sup>

Image may not contain physical data, even if this field is set.

`color_correction: Option<(f32,f32,f32)>` Divide color values by corresponding tuple member (r, g, b) to get to get physical radiance in watts/steradian/m<sup>2</sup>

Image may not contain physical data, even if this field is set.

`pixel_aspect_ratio: Option<f32>` Pixel height divided by pixel width

`custom_attributes: Vec<(String,String)>` All lines contained in image header are put here. Ordering of lines is preserved. Lines in the form “key=value” are represented as (“key”, “value”). All other lines are (“”, “line”)

Implementations

`impl Send for HdrMetadata`

`impl Sync for HdrMetadata`

`impl Unpin for HdrMetadata`

`impl UnwindSafe for HdrMetadata`

`impl RefUnwindSafe for HdrMetadata`

`impl Clone for HdrMetadata`

`fn clone(self: &Self) -> HdrMetadata`

```
impl Debug for HdrMetadata
```

```
 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

1.131 Typedef pub image::codecs::hdr::HDRMetadata

```rust
pub type HDRMetadata = HdrMetadata;
```

HDR MetaData

An alias of `struct.HdrMetadata.html`.

TODO: remove

1.132 Function pub image::codecs::hdr::read_raw_file

```rust
pub fn read_raw_file(path: P) -> ::std::io::Result<Vec<Rgb<f32>>>
```

Helper function for reading raw 3-channel f32 images

1.133 Struct pub image::codecs::hdr::HdrEncoder

```rust
pub struct HdrEncoder<W: Write>{
    // some fields omitted
}
```

Radiance HDR encoder

Implementations

```
impl<W: Write> HdrEncoder<W>
```

```
 pub fn new(w: W) -> HdrEncoder<W>
```

Creates encoder

```
 pub fn encode(mut self: Self, data: &[Rgb<f32>], width: usize, height: usize) -> ImageResul
```

Encodes the image "'data"' that has dimensions "'width"' and "'height"'

```
impl<W> where W: Send Send for HdrEncoder<W>
```

```
impl<W> where W: Sync Sync for HdrEncoder<W>
```

```
impl<W> where W: Unpin Unpin for HdrEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for HdrEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for HdrEncoder<W>
```

1.134 Typedef pub image::codecs::hdr::HDREncoder

```
pub type HDREncoder = HdrEncoder<R>;
```

HDR Encoder

An alias of `struct.HdrEncoder.html`.

TODO: remove

1.135 Function pub image::codecs::hdr::to_rgbe8

```
pub fn to_rgbe8(pix: Rgb<f32>) -> Rgbe8Pixel
```

Converts `Rgb<f32>` into `RGBE8Pixel`

1.136 Module pub image::codecs::ico

- image::codecs::ico::IcoDecoder
- image::codecs::ico::IcoEncoder
- image::codecs::ico::ICOEncoder

1.137 Struct pub image::codecs::ico::IcoDecoder

```
pub struct IcoDecoder<R: Read>{
    // some fields omitted
}
```

An ico decoder

Implementations

`impl<R: Read + Seek> IcoDecoder<R>`

`pub fn new(mut r: R) -> ImageResult<IcoDecoder<R»`

Create a new decoder that decodes from the stream “‘r“‘

`impl<R> where R: Send Send for IcoDecoder<R>`

`impl<R> where R: Sync Sync for IcoDecoder<R>`

`impl<R> where R: Unpin Unpin for IcoDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for IcoDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for IcoDecoder<R>`

`impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for IcoDecoder<R>`

```
 type Reader = IcoReader<R>;
```

```
 fn dimensions(self:  &Self) -> (u32,u32)
```

```
 fn color_type(self:  &Self) -> ColorType
```

```
 fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>
```

```
 fn read_image(self:  Self, buf:  &mut [u8]) -> ImageResult<()>
```

1.138 Struct pub image::codecs::ico::IcoEncoder

```
pub struct IcoEncoder<W: Write>{
    // some fields omitted
}
```

ICO encoder

Implementations

```
impl<W: Write> IcoEncoder<W>
```

```
 pub fn new(w:  W) -> IcoEncoder<W>
```

Create a new encoder that writes its output to ‘‘‘w‘‘‘.

```
 pub fn encode( mut self:  Self, data:  &[u8], width:  u32, height:  u32, color:  ColorType
) -> ImageResult<()>
```

Encodes the image ‘‘‘image‘‘‘ that has dimensions ‘‘‘width‘‘‘ and ‘‘‘height‘‘‘ and ‘‘‘ColorType‘‘‘ ‘‘‘c‘‘‘. The dimensions of the image must be between 1 and 256 (inclusive) or an error will be returned.

```
impl<W> where W: Send Send for IcoEncoder<W>
```

```
impl<W> where W: Sync Sync for IcoEncoder<W>
```

```
impl<W> where W: Unpin Unpin for IcoEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for IcoEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for IcoEncoder<W>
```

```
impl<W: Write> ImageEncoder for IcoEncoder<W>
```

```
 fn write_image( self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.139 Typedef pub image::codecs::ico::ICOEncoder

```
pub type ICOEncoder = IcoEncoder<W>;
```

ICO encoder

An alias of `struct.IcoEncoder.html`.

TODO: remove

1.140 Module pub image::codecs::jpeg

- image::codecs::jpeg::JpegDecoder
- image::codecs::jpeg::JpegEncoder
- image::codecs::jpeg::PixelDensity
- image::codecs::jpeg::PixelDensityUnit
- image::codecs::jpeg::JPEGEncoder

1.141 Struct pub image::codecs::jpeg::JpegDecoder

```rust
pub struct JpegDecoder<R>{
    // some fields omitted
}
```

JPEG decoder

Implementations

`impl<R: Read> JpegDecoder<R>`

` pub fn new(r:  R) -> ImageResult<JpegDecoder<R»`

Create a new decoder that decodes from the stream "'r"'

` pub fn scale( self:  &mut Self, requested_width:  u16, requested_height:  u16 ) -> ImageResult<`

Configure the decoder to scale the image during decoding.

This efficiently scales the image by the smallest supported scale factor that produces an image larger than or equal to the requested size in at least one axis. The currently implemented scale factors are 1/8, 1/4, 1/2 and 1.

To generate a thumbnail of an exact size, pass the desired size and then scale to the final size using a traditional resampling algorithm.

The size of the image to be loaded, with the scale factor applied, is returned.

`impl<R> where R: Send Send for JpegDecoder<R>`

`impl<R> where R: Sync Sync for JpegDecoder<R>`

`impl<R> where R: Unpin Unpin for JpegDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for JpegDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for JpegDecoder<R>`

`impl<'a, R: 'a + Read> ImageDecoder<'a> for JpegDecoder<R>`

`type Reader = JpegReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(mut self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

1.142 Struct pub image::codecs::jpeg::JpegEncoder

```
pub struct JpegEncoder<'a, W: 'a>{
    // some fields omitted
}
```

The representation of a JPEG encoder

Implementations

`impl<'a, W: Write> JpegEncoder<'a, W>`

`pub fn new(w: &mut W) -> JpegEncoder<'_, W>`

Create a new encoder that writes its output to "'w"'

`pub fn new_with_quality(w: &mut W, quality: u8) -> JpegEncoder<'_, W>`

Create a new encoder that writes its output to "'w"', and has the quality parameter "'quality"' with a value in the range 1-100 where 1 is the worst and 100 is the best.

`pub fn set_pixel_density(self: &mut Self, pixel_density: PixelDensity)`

Set the pixel density of the images the encoder will encode. If this method is not called, then a default pixel aspect ratio of 1x1 will be applied, and no DPI information will be stored in the image.

`pub fn encode( self: &mut Self, image: &[u8], width: u32, height: u32, color_type: ColorType ) -> ImageResult<()>`

Encodes the image stored in the raw byte buffer "'image"' that has dimensions "'width"' and "'height"' and "'ColorType"' "'c"'

The Image in encoded with subsampling ratio 4:2:2

`pub fn encode_image<I: GenericImageView>(self: &mut Self, image: &I) -> ImageResult<()>`

Encodes the given image.

As a special feature this does not require the whole image to be present in memory at the same time such that it may be computed on the fly, which is why this method exists on this encoder but not on others. Instead the encoder will iterate over 8-by-8 blocks of pixels at a time, inspecting each pixel exactly once. You can rely on this behaviour when calling this method.

The Image in encoded with subsampling ratio 4:2:2

```
impl<'a, W> where W: Send Send for JpegEncoder<'a, W>
```

```
impl<'a, W> where W: Sync Sync for JpegEncoder<'a, W>
```

```
impl<'a, W> Unpin for JpegEncoder<'a, W>
```

```
impl<'a, W> !UnwindSafe for JpegEncoder<'a, W>
```

```
impl<'a, W> where W: RefUnwindSafe RefUnwindSafe for JpegEncoder<'a, W>
```

```
impl<'a, W: Write> ImageEncoder for JpegEncoder<'a, W>
```

```
 fn write_image( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.143 Struct pub image::codecs::jpeg::PixelDensity

```
pub struct PixelDensity{
    pub density: (u16,u16),
    pub unit: PixelDensityUnit,
}
```

Represents the pixel density of an image

For example, a 300 DPI image is represented by:

```
use image::jpeg::*;
let hdpi = PixelDensity::dpi(300);
assert_eq!(hdpi, PixelDensity {density: (300,300), unit: PixelDensityUnit::Inches})
```

Fields

`density: (u16,u16)` A couple of values for (Xdensity, Ydensity)

`unit: PixelDensityUnit` The unit in which the density is measured

Implementations

```
impl PixelDensity
```

```
 pub fn dpi(density:  u16) -> Self
```

Creates the most common pixel density type: the horizontal and the vertical density are equal, and measured in pixels per inch.

```
impl Send for PixelDensity
```

```
impl Sync for PixelDensity
```

`impl Unpin for PixelDensity`

`impl UnwindSafe for PixelDensity`

`impl RefUnwindSafe for PixelDensity`

`impl Clone for PixelDensity`

`fn clone(self: &Self) -> PixelDensity`

`impl Copy for PixelDensity`

`impl Default for PixelDensity`

`fn default() -> Self`

Returns a pixel density with a pixel aspect ratio of 1

`impl Eq for PixelDensity`

`impl PartialEq<PixelDensity> for PixelDensity`

`fn eq(self: &Self, other: &PixelDensity) -> bool`

`fn ne(self: &Self, other: &PixelDensity) -> bool`

`impl Debug for PixelDensity`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for PixelDensity`

`impl StructuralEq for PixelDensity`

1.144 Enum pub image::codecs::jpeg::PixelDensityUnit

```
pub enum image::codecs::jpeg::PixelDensityUnit {
    PixelAspectRatio,
    Inches,
    Centimeters,
}
```

Represents a unit in which the density of an image is measured

Variants

`PixelAspectRatio` Represents the absence of a unit, the values indicate only a pixel aspect ratio

`Inches` Pixels per inch (2.54 cm)

`Centimeters` Pixels per centimeter

Implementations

```
impl Send for PixelDensityUnit

impl Sync for PixelDensityUnit

impl Unpin for PixelDensityUnit

impl UnwindSafe for PixelDensityUnit

impl RefUnwindSafe for PixelDensityUnit

impl Clone for PixelDensityUnit

 fn clone(self:  &Self) -> PixelDensityUnit

impl Copy for PixelDensityUnit

impl Eq for PixelDensityUnit

impl PartialEq<PixelDensityUnit> for PixelDensityUnit

 fn eq(self:  &Self, other:  &PixelDensityUnit) -> bool

impl Debug for PixelDensityUnit

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for PixelDensityUnit

impl StructuralEq for PixelDensityUnit
```

1.145 Typedef pub image::codecs::jpeg::JPEGEncoder

```rust
pub type JPEGEncoder = JpegEncoder<'a, W>;
```

JPEG Encoder

An alias of `struct.JpegEncoder.html`.

TODO: remove

1.146 Module pub image::codecs::png

- image::codecs::png::PngReader
- image::codecs::png::PNGReader
- image::codecs::png::PngDecoder
- image::codecs::png::ApngDecoder
- image::codecs::png::PngEncoder
- image::codecs::png::PNGEncoder
- image::codecs::png::CompressionType
- image::codecs::png::FilterType

1.147 Struct pub image::codecs::png::PngReader

```
pub struct PngReader<R: Read>{
    // some fields omitted
}
```

Png Reader

This reader will try to read the png one row at a time, however for interlaced png files this is not possible and these are therefore read at once.

Implementations

`impl<R> where R: Send Send for PngReader<R>`

`impl<R> where R: Sync Sync for PngReader<R>`

`impl<R> where R: Unpin Unpin for PngReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for PngReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for PngReader<R>`

`impl<R: Read> Read for PngReader<R>`

`fn read(self: &mut Self, mut buf: &mut [u8]) -> io::Result<usize>`

`fn read_to_end(self: &mut Self, buf: &mut Vec<u8>) -> io::Result<usize>`

1.148 Typedef pub image::codecs::png::PNGReader

```
pub type PNGReader = PngReader<R>;
```

PNG Reader

An alias of `struct.PngReader.html`.

TODO: remove

1.149 Struct pub image::codecs::png::PngDecoder

```
pub struct PngDecoder<R: Read>{
    // some fields omitted
}
```

PNG decoder

Implementations

`impl<R: Read> PngDecoder<R>`

`pub fn new(r: R) -> ImageResult<PngDecoder<R»`

Creates a new decoder that decodes from the stream "'r"'

```
pub fn apng(self: Self) -> ApngDecoder<R>
```

Turn this into an iterator over the animation frames.

Reading the complete animation requires more memory than reading the data from the IDAT frame–multiple frame buffers need to be reserved at the same time. We further do not support compositing 16-bit colors. In any case this would be lossy as the interface of animation decoders does not support 16-bit colors.

If something is not supported or a limit is violated then the decoding step that requires them will fail and an error will be returned instead of the frame. No further frames will be returned.

```
pub fn is_apng(self: &Self) -> bool
```

Returns if the image contains an animation.

Note that the file itself decides if the default image is considered to be part of the animation. When it is not the common interpretation is to use it as a thumbnail.

If a non-animated image is converted into an 'ApngDecoder' then its iterator is empty.

```
impl<R> where R: Send Send for PngDecoder<R>

impl<R> where R: Sync Sync for PngDecoder<R>

impl<R> where R: Unpin Unpin for PngDecoder<R>

impl<R> where R: UnwindSafe UnwindSafe for PngDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for PngDecoder<R>

impl<'a, R: 'a + Read> ImageDecoder<'a> for PngDecoder<R>

 type Reader = PngReader<R>;

 fn dimensions(self: &Self) -> (u32,u32)

 fn color_type(self: &Self) -> ColorType

 fn into_reader(self: Self) -> ImageResult«Self as >::Reader>

 fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>

 fn scanline_bytes(self: &Self) -> u64
```

1.150 Struct pub image::codecs::png::ApngDecoder

```
pub struct ApngDecoder<R: Read>{
    // some fields omitted
}
```

An ../trait.AnimationDecoder.html adapter of struct.PngDecoder.html.

See struct.PngDecoder.html#method.apng for more information.

Implementations

`impl<R> where R: Send Send for ApngDecoder<R>`

`impl<R> where R: Sync Sync for ApngDecoder<R>`

`impl<R> where R: Unpin Unpin for ApngDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for ApngDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for ApngDecoder<R>`

`impl<'a, R: Read + 'a> AnimationDecoder<'a> for ApngDecoder<R>`

`fn into_frames(self: Self) -> Frames<'a>`

1.151 Struct pub image::codecs::png::PngEncoder

```
pub struct PngEncoder<W: Write>{
    // some fields omitted
}
```

PNG encoder

Implementations

`impl<W: Write> PngEncoder<W>`

`pub fn new(w: W) -> PngEncoder<W>`

Create a new encoder that writes its output to "'w"'

`pub fn new_with_quality(w: W, compression: CompressionType, filter: FilterType) -> PngEncoder<W>`

Create a new encoder that writes its output to 'w' with 'CompressionType' 'compression' and 'FilterType' 'filter'.

It is best to view the options as a _hint_ to the implementation on the smallest or fastest option for encoding a particular image. That is, using options that map directly to a PNG image parameter will use this parameter where possible. But variants that have no direct mapping may be interpreted differently in minor versions. The exact output is expressly __not__ part the SemVer stability guarantee.

Note that it is not optimal to use a single filter type. It is likely that the library used will at some point gain the ability to use adaptive filtering methods per pixel row (or even interlaced row). We might make it the new default variant in which case choosing a particular filter method likely produces larger images. Be sure to check the release notes once in a while.

```
 pub fn encode(self:  Self, data:  &[u8], width:  u32, height:  u32, color:  ColorType)
-> ImageResult<()>
```

Encodes the image 'data' that has dimensions 'width' and 'height' and 'ColorType' 'c'.

`impl<W> where W: Send Send for PngEncoder<W>`

`impl<W> where W: Sync Sync for PngEncoder<W>`

`impl<W> where W: Unpin Unpin for PngEncoder<W>`

`impl<W> where W: UnwindSafe UnwindSafe for PngEncoder<W>`

`impl<W> where W: RefUnwindSafe RefUnwindSafe for PngEncoder<W>`

`impl<W: Write> ImageEncoder for PngEncoder<W>`

```
 fn write_image( self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.152 Typedef pub image::codecs::png::PNGEncoder

```
pub type PNGEncoder = PngEncoder<W>;
```

PNG Encoder

An alias of `struct.PngEncoder.html`.

TODO: remove

1.153 Enum pub image::codecs::png::CompressionType

```
pub enum image::codecs::png::CompressionType {
    Default,
    Fast,
    Best,
    Huffman,
    Rle,
    // some variants omitted
}
```

Compression level of a PNG encoder. The default setting is `Fast`.

Variants

`Default` Default compression level

`Fast` Fast, minimal compression

`Best` High compression level

`Huffman` Huffman coding compression

`Rle` Run-length encoding compression

Implementations

`impl Send for CompressionType`

`impl Sync for CompressionType`

`impl Unpin for CompressionType`

`impl UnwindSafe for CompressionType`

`impl RefUnwindSafe for CompressionType`

`impl Clone for CompressionType`

` fn clone(self: &Self) -> CompressionType`

`impl Copy for CompressionType`

`impl Default for CompressionType`

` fn default() -> Self`

`impl Eq for CompressionType`

`impl PartialEq<CompressionType> for CompressionType`

` fn eq(self: &Self, other: &CompressionType) -> bool`

` fn ne(self: &Self, other: &CompressionType) -> bool`

`impl Debug for CompressionType`

` fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for CompressionType`

`impl StructuralEq for CompressionType`

1.154 Enum pub image::codecs::png::FilterType

```rust
pub enum image::codecs::png::FilterType {
    NoFilter,
    Sub,
    Up,
    Avg,
    Paeth,
    // some variants omitted
}
```

Filter algorithms used to process image data to improve compression.

The default filter is `Sub` though this default may change in the future, most notable if an adaptive encoding option is implemented.

Variants

`NoFilter` No processing done, best used for low bit depth greyscale or data with a low color count

`Sub` Filters based on previous pixel in the same scanline

`Up` Filters based on the scanline above

`Avg` Filters based on the average of left and right neighbor pixels

`Paeth` Algorithm that takes into account the left, upper left, and above pixels

Implementations

```
impl Send for FilterType

impl Sync for FilterType

impl Unpin for FilterType

impl UnwindSafe for FilterType

impl RefUnwindSafe for FilterType

impl Clone for FilterType

 fn clone(self:  &Self) -> FilterType

impl Copy for FilterType

impl Default for FilterType

 fn default() -> Self

impl Eq for FilterType

impl PartialEq<FilterType> for FilterType

 fn eq(self:  &Self, other:  &FilterType) -> bool

 fn ne(self:  &Self, other:  &FilterType) -> bool

impl Debug for FilterType

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for FilterType

impl StructuralEq for FilterType
```

1.155 Module pub image::codecs::pnm

- image::codecs::pnm::PnmDecoder

- image::codecs::pnm::PnmEncoder
- image::codecs::pnm::PNMEncoder
- image::codecs::pnm::ArbitraryHeader
- image::codecs::pnm::ArbitraryTuplType
- image::codecs::pnm::BitmapHeader
- image::codecs::pnm::GraymapHeader
- image::codecs::pnm::PixmapHeader
- image::codecs::pnm::PnmHeader
- image::codecs::pnm::PNMHeader
- image::codecs::pnm::PNMSubtype
- image::codecs::pnm::PnmSubtype
- image::codecs::pnm::SampleEncoding

1.156 Struct pub image::codecs::pnm::PnmDecoder

```
pub struct PnmDecoder<R>{
    // some fields omitted
}
```

PNM decoder

Implementations

`impl<R: Read> PnmDecoder<R>`

`pub fn new(read: R) -> ImageResult<PnmDecoder<R»`

Create a new decoder that decodes from the stream “‘read“‘

`pub fn into_inner(self: Self) -> (R,PnmHeader)`

Extract the reader and header after an image has been read.

`impl<R: Read> PnmDecoder<R>`

`pub fn subtype(self: &Self) -> PNMSubtype`

Get the pnm subtype, depending on the magic constant contained in the header

`impl<R> where R: Send Send for PnmDecoder<R>`

`impl<R> where R: Sync Sync for PnmDecoder<R>`

`impl<R> where R: Unpin Unpin for PnmDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for PnmDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for PnmDecoder<R>`

`impl<’a, R: ’a + Read> ImageDecoder<’a> for PnmDecoder<R>`

`type Reader = PnmReader<R>;`

```
fn dimensions(self: &Self) -> (u32,u32)
```

```
fn color_type(self: &Self) -> ColorType
```

```
fn original_color_type(self: &Self) -> ExtendedColorType
```

```
fn into_reader(self: Self) -> ImageResult«Self as >::Reader>
```

```
fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>
```

1.157 Struct pub image::codecs::pnm::PnmEncoder

```
pub struct PnmEncoder<W: Write>{
    // some fields omitted
}
```

Encodes images to any of the pnm image formats.

Implementations

```
impl<W: Write> PnmEncoder<W>
```

```
pub fn new(writer: W) -> Self
```

Create new PNMEncoder from the ‘writer‘.

The encoded images will have some ‘pnm‘ format. If more control over the image type is required, use either one of ‘with_subtype‘ or ‘with_header‘. For more information on the behaviour, see ‘with_dynamic_header‘.

```
pub fn with_subtype(self: Self, subtype: PNMSubtype) -> Self
```

Encode a specific pnm subtype image.

The magic number and encoding type will be chosen as provided while the rest of the header data will be generated dynamically. Trying to encode incompatible images (e.g. encoding an RGB image as Graymap) will result in an error.

This will overwrite the effect of earlier calls to ‘with_header‘ and ‘with_dynamic_header‘.

```
pub fn with_header(self: Self, header: PnmHeader) -> Self
```

Enforce the use of a chosen header.

While this option gives the most control over the actual written data, the encoding process will error in case the header data and image parameters do not agree. It is the users obligation to ensure that the width and height are set accordingly, for example.

Choose this option if you want a lossless decoding/encoding round trip.

This will overwrite the effect of earlier calls to ‘with_subtype‘ and ‘with_dynamic_header‘.

```
pub fn with_dynamic_header(self: Self) -> Self
```

Create the header dynamically for each image.

This is the default option upon creation of the encoder. With this, most images should be encodable but the specific format chosen is out of the users control. The pnm subtype is chosen arbitrarily by the library.

This will overwrite the effect of earlier calls to 'with_subtype' and 'with_header'.

```
 pub fn encode<'s, S> where S: Into<FlatSamples<'s»(self:  &mut Self, image:  S, width:
u32, height:  u32, color:  ColorType) -> ImageResult<()>
```

Encode an image whose samples are represented as 'u8'.

Some 'pnm' subtypes are incompatible with some color options, a chosen header most certainly with any deviation from the original decoded image.

```
impl<W> where W: Send Send for PnmEncoder<W>
```

```
impl<W> where W: Sync Sync for PnmEncoder<W>
```

```
impl<W> where W: Unpin Unpin for PnmEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for PnmEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for PnmEncoder<W>
```

```
impl<W: Write> ImageEncoder for PnmEncoder<W>
```

```
 fn write_image( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.158 Typedef pub image::codecs::pnm::PNMEncoder

```
pub type PNMEncoder = PnmEncoder<W>;
```

PNM Encoder

An alias of `struct.PnmEncoder.html`.

TODO: remove

1.159 Struct pub image::codecs::pnm::ArbitraryHeader

```
pub struct ArbitraryHeader{
    pub height: u32,
    pub width: u32,
    pub depth: u32,
    pub maxval: u32,
    pub tupltype: Option<ArbitraryTuplType>,
}
```

Header produced by a `pam` file ("Portable Arbitrary Map")

Fields

`height:` `u32` Height of the image file

`width:` `u32` Width of the image file

`depth:` `u32` Number of color channels

`maxval:` `u32` Maximum sample value within the image

`tupltype:` `Option<ArbitraryTuplType>` Color interpretation of image pixels

Implementations

`impl Send for ArbitraryHeader`

`impl Sync for ArbitraryHeader`

`impl Unpin for ArbitraryHeader`

`impl UnwindSafe for ArbitraryHeader`

`impl RefUnwindSafe for ArbitraryHeader`

`impl From<ArbitraryHeader> for PnmHeader`

`fn from(header: ArbitraryHeader) -> Self`

`impl Clone for ArbitraryHeader`

`fn clone(self: &Self) -> ArbitraryHeader`

`impl Debug for ArbitraryHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.160 Enum pub image::codecs::pnm::ArbitraryTuplType

```
pub enum image::codecs::pnm::ArbitraryTuplType {
    BlackAndWhite,
    BlackAndWhiteAlpha,
    Grayscale,
    GrayscaleAlpha,
    RGB,
    RGBAlpha,
    Custom,
}
```

Standardized tuple type specifiers in the header of a `pam`.

Variants

`BlackAndWhite` Pixels are either black (0) or white (1)

`BlackAndWhiteAlpha` Pixels are either black (0) or white (1) and a second alpha channel

`Grayscale` Pixels represent the amount of white

`GrayscaleAlpha` Grayscale with an additional alpha channel

`RGB` Three channels: Red, Green, Blue

`RGBAlpha` Four channels: Red, Green, Blue, Alpha

`Custom` An image format which is not standardized

Implementations

`impl Send for ArbitraryTuplType`

`impl Sync for ArbitraryTuplType`

`impl Unpin for ArbitraryTuplType`

`impl UnwindSafe for ArbitraryTuplType`

`impl RefUnwindSafe for ArbitraryTuplType`

`impl Clone for ArbitraryTuplType`

`fn clone(self: &Self) -> ArbitraryTuplType`

`impl Debug for ArbitraryTuplType`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.161 Struct pub image::codecs::pnm::BitmapHeader

```
pub struct BitmapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
}
```

Header produced by a `pbm` file ("Portable Bit Map")

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height: u32` Height of the image file

`width: u32` Width of the image file

Implementations

`impl Send for BitmapHeader`

`impl Sync for BitmapHeader`

`impl Unpin for BitmapHeader`

`impl UnwindSafe for BitmapHeader`

`impl RefUnwindSafe for BitmapHeader`

`impl From<BitmapHeader> for PnmHeader`

`fn from(header: BitmapHeader) -> Self`

`impl Clone for BitmapHeader`

`fn clone(self: &Self) -> BitmapHeader`

`impl Copy for BitmapHeader`

`impl Debug for BitmapHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.162 Struct pub image::codecs::pnm::GraymapHeader

```
pub struct GraymapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
    pub maxwhite: u32,
}
```

Header produced by a pgm file ("Portable Gray Map")

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height: u32` Height of the image file

`width: u32` Width of the image file

`maxwhite: u32` Maximum sample value within the image

Implementations

`impl Send for GraymapHeader`

`impl Sync for GraymapHeader`

`impl Unpin for GraymapHeader`

`impl UnwindSafe for GraymapHeader`

`impl RefUnwindSafe for GraymapHeader`

`impl From<GraymapHeader> for PnmHeader`

`fn from(header: GraymapHeader) -> Self`

`impl Clone for GraymapHeader`

`fn clone(self: &Self) -> GraymapHeader`

`impl Copy for GraymapHeader`

`impl Debug for GraymapHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.163 Struct pub image::codecs::pnm::PixmapHeader

```
pub struct PixmapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
    pub maxval: u32,
}
```

Header produced by a ppm file ("Portable Pixel Map")

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height: u32` Height of the image file

`width: u32` Width of the image file

`maxval: u32` Maximum sample value within the image

Implementations

`impl Send for PixmapHeader`

`impl Sync for PixmapHeader`

`impl Unpin for PixmapHeader`

`impl UnwindSafe for PixmapHeader`

`impl RefUnwindSafe for PixmapHeader`

`impl From<PixmapHeader> for PnmHeader`

`fn from(header: PixmapHeader) -> Self`

`impl Clone for PixmapHeader`

`fn clone(self: &Self) -> PixmapHeader`

```
impl Copy for PixmapHeader
```

```
impl Debug for PixmapHeader
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

1.164 Struct pub image::codecs::pnm::PnmHeader

```
pub struct PnmHeader{
    // some fields omitted
}
```

Stores the complete header data of a file.

Internally, provides mechanisms for lossless reencoding. After reading a file with the decoder it is possible to recover the header and construct an encoder. Using the encoder on the just loaded image should result in a byte copy of the original file (for single image pnms without additional trailing data).

Implementations

```
impl PnmHeader
```

```
 pub fn subtype(self:  &Self) -> PNMSubtype
```

Retrieve the format subtype from which the header was created.

```
 pub fn width(self:  &Self) -> u32
```

The width of the image this header is for.

```
 pub fn height(self:  &Self) -> u32
```

The height of the image this header is for.

```
 pub fn maximal_sample(self:  &Self) -> u32
```

The biggest value a sample can have. In other words, the colour resolution.

```
 pub fn as_bitmap(self:  &Self) -> Option<&BitmapHeader>
```

Retrieve the underlying bitmap header if any

```
 pub fn as_graymap(self:  &Self) -> Option<&GraymapHeader>
```

Retrieve the underlying graymap header if any

```
 pub fn as_pixmap(self:  &Self) -> Option<&PixmapHeader>
```

Retrieve the underlying pixmap header if any

```
 pub fn as_arbitrary(self:  &Self) -> Option<&ArbitraryHeader>
```

Retrieve the underlying arbitrary header if any

```
 pub fn write(self:  &Self, writer:  &mut io::Write) -> io::Result<()>
```

Write the header back into a binary stream

```
impl Send for PnmHeader
```

```
impl Sync for PnmHeader
```

```
impl Unpin for PnmHeader
```

```
impl UnwindSafe for PnmHeader
```

```
impl RefUnwindSafe for PnmHeader
```

```
impl From<BitmapHeader> for PnmHeader
```

```
 fn from(header:  BitmapHeader) -> Self
```

```
impl From<GraymapHeader> for PnmHeader
```

```
 fn from(header:  GraymapHeader) -> Self
```

```
impl From<PixmapHeader> for PnmHeader
```

```
 fn from(header:  PixmapHeader) -> Self
```

```
impl From<ArbitraryHeader> for PnmHeader
```

```
 fn from(header:  ArbitraryHeader) -> Self
```

1.165 Typedef pub image::codecs::pnm::PNMHeader

```
pub type PNMHeader = PnmHeader;
```

PNM Header

An alias of `struct.PnmHeader.html`.

TODO: remove

1.166 Enum pub image::codecs::pnm::PNMSubtype

```
pub enum image::codecs::pnm::PNMSubtype {
    Bitmap,
    Graymap,
    Pixmap,
    ArbitraryMap,
}
```

Denotes the category of the magic number

DEPRECATED: The name of this enum will be changed to `type.PnmSubtype.html`.

TODO: rename to `type.PnmSubtype.html`.

Variants

`Bitmap` Magic numbers P1 and P4

`Graymap` Magic numbers P2 and P5

`Pixmap` Magic numbers P3 and P6

`ArbitraryMap` Magic number P7

Implementations

```
impl PNMSubtype

 pub fn magic_constant(self:  Self) -> &'static [u8; 2]
```

Get the two magic constant bytes corresponding to this format subtype.

```
 pub fn sample_encoding(self:  Self) -> SampleEncoding
```

Whether samples are stored as binary or as decimal ascii

```
impl Send for PNMSubtype

impl Sync for PNMSubtype

impl Unpin for PNMSubtype

impl UnwindSafe for PNMSubtype

impl RefUnwindSafe for PNMSubtype

impl Clone for PNMSubtype

 fn clone(self:  &Self) -> PNMSubtype

impl Copy for PNMSubtype

impl Eq for PNMSubtype

impl PartialEq<PNMSubtype> for PNMSubtype

 fn eq(self:  &Self, other:  &PNMSubtype) -> bool

 fn ne(self:  &Self, other:  &PNMSubtype) -> bool

impl Debug for PNMSubtype

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

impl StructuralPartialEq for PNMSubtype

impl StructuralEq for PNMSubtype

1.167 Typedef pub image::codecs::pnm::PnmSubtype

```rust
pub type PnmSubtype = PNMSubtype;
```

PNM Subtype

An alias of `enum.PNMSubtype.html`.

TODO: remove when [`DXTVariant`] is renamed.

1.168 Enum pub image::codecs::pnm::SampleEncoding

```rust
pub enum image::codecs::pnm::SampleEncoding {
    Binary,
    Ascii,
}
```

The kind of encoding used to store sample values

Variants

`Binary` Samples are unsigned binary integers in big endian

`Ascii` Samples are encoded as decimal ascii strings separated by whitespace

Implementations

impl Send for SampleEncoding

impl Sync for SampleEncoding

impl Unpin for SampleEncoding

impl UnwindSafe for SampleEncoding

impl RefUnwindSafe for SampleEncoding

impl Clone for SampleEncoding

 fn clone(self: &Self) -> SampleEncoding

impl Copy for SampleEncoding

impl Eq for SampleEncoding

impl PartialEq<SampleEncoding> for SampleEncoding

 fn eq(self: &Self, other: &SampleEncoding) -> bool

```
impl Debug for SampleEncoding

 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for SampleEncoding

impl StructuralEq for SampleEncoding
```

1.169 Module pub image::codecs::tga

- image::codecs::tga::TgaDecoder
- image::codecs::tga::TgaEncoder

1.170 Struct pub image::codecs::tga::TgaDecoder

```
pub struct TgaDecoder<R>{
    // some fields omitted
}
```

The representation of a TGA decoder

Implementations

```
impl<R: Read + Seek> TgaDecoder<R>

 pub fn new(r: R) -> ImageResult<TgaDecoder<R»
```

Create a new decoder that decodes from the stream 'r'

```
impl<R> where R: Send Send for TgaDecoder<R>

impl<R> where R: Sync Sync for TgaDecoder<R>

impl<R> where R: Unpin Unpin for TgaDecoder<R>

impl<R> where R: UnwindSafe UnwindSafe for TgaDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for TgaDecoder<R>

impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for TgaDecoder<R>

 type Reader = TGAReader<R>;

 fn dimensions(self: &Self) -> (u32,u32)

 fn color_type(self: &Self) -> ColorType

 fn scanline_bytes(self: &Self) -> u64

 fn into_reader(self: Self) -> ImageResult«Self as >::Reader>

 fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>
```

1.171 Struct pub image::codecs::tga::TgaEncoder

```
pub struct TgaEncoder<W: Write>{
    // some fields omitted
}
```

TGA encoder.

Implementations

```
impl<W: Write> TgaEncoder<W>
```

```
 pub fn new(w:  W) -> TgaEncoder<W>
```

Create a new encoder that writes its output to “‘w“‘.

```
 pub fn encode( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

Encodes the image “‘buf“‘ that has dimensions “‘width“‘ and “‘height“‘ and “‘ColorType“‘ “‘color_type“‘.

The dimensions of the image must be between 0 and 65535 (inclusive) or an error will be returned.

```
impl<W> where W: Send Send for TgaEncoder<W>
```

```
impl<W> where W: Sync Sync for TgaEncoder<W>
```

```
impl<W> where W: Unpin Unpin for TgaEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for TgaEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for TgaEncoder<W>
```

```
impl<W: Write> ImageEncoder for TgaEncoder<W>
```

```
 fn write_image( self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.172 Module pub image::codecs::tiff

- image::codecs::tiff::TiffDecoder
- image::codecs::tiff::TiffReader
- image::codecs::tiff::TiffEncoder

1.173 Struct pub image::codecs::tiff::TiffDecoder

```
pub struct TiffDecoder<R>
where
    R: Read + Seek{
    // some fields omitted
}
```

Decoder for TIFF images.

Implementations

`impl<R> where R: Read + Seek TiffDecoder<R>`

`pub fn new(r: R) -> Result<TiffDecoder<R>, ImageError>`

Create a new TiffDecoder.

`impl<R> where R: Send Send for TiffDecoder<R>`

`impl<R> where R: Sync Sync for TiffDecoder<R>`

`impl<R> where R: Unpin Unpin for TiffDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for TiffDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for TiffDecoder<R>`

`impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for TiffDecoder<R>`

`type Reader = TiffReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(mut self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

1.174 Struct pub image::codecs::tiff::TiffReader

```
pub struct TiffReader<R>(
    // some fields omitted
)
```

Wrapper struct around a `Cursor<Vec<u8»`

Implementations

`impl<R> where R: Send Send for TiffReader<R>`

`impl<R> where R: Sync Sync for TiffReader<R>`

`impl<R> where R: Unpin Unpin for TiffReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for TiffReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for TiffReader<R>`

`impl<R> Read for TiffReader<R>`

`fn read(self: &mut Self, buf: &mut [u8]) -> io::Result<usize>`

`fn read_to_end(self: &mut Self, buf: &mut Vec<u8>) -> io::Result<usize>`

1.175 Struct pub image::codecs::tiff::TiffEncoder

```
pub struct TiffEncoder<W>{
    // some fields omitted
}
```

Encoder for tiff images

Implementations

```
impl<W: Write + Seek> TiffEncoder<W>
```

```
 pub fn new(w:  W) -> TiffEncoder<W>
```

Create a new encoder that writes its output to ‘w‘

```
 pub fn encode(self:  Self, data:  &[u8], width:  u32, height:  u32, color:  ColorType)
-> ImageResult<()>
```

Encodes the image ‘image‘ that has dimensions ‘width‘ and ‘height‘ and ‘ColorType‘ ‘c‘.

16-bit types assume the buffer is native endian.

```
impl<W> where W: Send Send for TiffEncoder<W>
```

```
impl<W> where W: Sync Sync for TiffEncoder<W>
```

```
impl<W> where W: Unpin Unpin for TiffEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for TiffEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for TiffEncoder<W>
```

```
impl<W: Write + Seek> ImageEncoder for TiffEncoder<W>
```

```
 fn write_image( self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.176 Module pub image::codecs::webp

- image::codecs::webp::vp8
- image::codecs::webp::WebPDecoder

1.177 Module pub image::codecs::webp::vp8

- image::codecs::webp::vp8::Frame
- image::codecs::webp::vp8::Vp8Decoder

1.178 Struct pub image::codecs::webp::vp8::Frame

```
pub struct Frame{
    pub width: u16,
    pub height: u16,
    pub ybuf: Vec<u8>,
    pub keyframe: bool,
    pub for_display: bool,
    pub pixel_type: u8,
    // some fields omitted
}
```

A Representation of the last decoded video frame

Fields

`width: u16` The width of the luma plane

`height: u16` The height of the luma plane

`ybuf: Vec<u8>` The luma plane of the frame

`keyframe: bool` Indicates whether this frame is a keyframe

`for_display: bool` Indicates whether this frame is intended for display

`pixel_type: u8` The pixel type of the frame as defined by Section 9.2 of the VP8 Specification

Implementations

`impl Send for Frame`

`impl Sync for Frame`

`impl Unpin for Frame`

`impl UnwindSafe for Frame`

`impl RefUnwindSafe for Frame`

`impl Clone for Frame`

`fn clone(self: &Self) -> Frame`

`impl Default for Frame`

`fn default() -> Frame`

`impl Debug for Frame`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.179 Struct pub image::codecs::webp::vp8::Vp8Decoder

```
pub struct Vp8Decoder<R>{
    // some fields omitted
}
```

VP8 Decoder

Only decodes keyframes

Implementations

`impl<R: Read> Vp8Decoder<R>`

`pub fn new(r: R) -> Vp8Decoder<R>`

Create a new decoder. The reader must present a raw vp8 bitstream to the decoder

`pub fn decode_frame(self: &mut Self) -> ImageResult<&Frame>`

Decodes the current frame and returns a reference to it

`impl<R> where R: Send Send for Vp8Decoder<R>`

`impl<R> where R: Sync Sync for Vp8Decoder<R>`

`impl<R> where R: Unpin Unpin for Vp8Decoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for Vp8Decoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for Vp8Decoder<R>`

1.180 Struct pub image::codecs::webp::WebPDecoder

```
pub struct WebPDecoder<R>{
    // some fields omitted
}
```

WebP Image format decoder. Currently only supportes the luma channel (meaning that decoded images will be grayscale).

Implementations

`impl<R: Read> WebPDecoder<R>`

`pub fn new(r: R) -> ImageResult<WebPDecoder<R»`

Create a new WebPDecoder from the Reader "'r"'. This function takes ownership of the Reader.

`impl<R> where R: Send Send for WebPDecoder<R>`

`impl<R> where R: Sync Sync for WebPDecoder<R>`

`impl<R> where R: Unpin Unpin for WebPDecoder<R>`

impl<R> where R: UnwindSafe UnwindSafe for WebPDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for WebPDecoder<R>

impl<'a, R: 'a + Read> ImageDecoder<'a> for WebPDecoder<R>

type Reader = WebpReader<R>;

fn dimensions(self: &Self) -> (u32,u32)

fn color_type(self: &Self) -> color::ColorType

fn into_reader(self: Self) -> ImageResult«Self as >::Reader>

fn read_image(self: Self, buf: &mut [u8]) -> ImageResult<()>

1.181 Module pub image::bmp

- image::codecs::bmp::BMPEncoder
- image::codecs::bmp::BmpDecoder
- image::codecs::bmp::BmpEncoder

1.182 Typedef pub image::codecs::bmp::BMPEncoder

```
pub type BMPEncoder = BmpEncoder<'a, W>;
```

BMP Encoder

An alias of `struct.BmpEncoder.html`.

TODO: remove

1.183 Struct pub image::codecs::bmp::BmpDecoder

```
pub struct BmpDecoder<R>{
    // some fields omitted
}
```

A bmp decoder

Implementations

impl<R: Read + Seek> BmpDecoder<R>

pub fn new(reader: R) -> ImageResult<BmpDecoder<R»

Create a new decoder that decodes from the stream "'r"'

impl<R> where R: Send Send for BmpDecoder<R>

`impl<R> where R: Sync Sync for BmpDecoder<R>`

`impl<R> where R: Unpin Unpin for BmpDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for BmpDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for BmpDecoder<R>`

`impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for BmpDecoder<R>`

`type Reader = BmpReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

`impl<'a, R: 'a + Read + Seek> ImageDecoderExt<'a> for BmpDecoder<R>`

`fn read_rect_with_progress<F: Fn>( self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F ) -> ImageResult<()>`

1.184 Struct pub image::codecs::bmp::BmpEncoder

```
pub struct BmpEncoder<'a, W: 'a>{
    // some fields omitted
}
```

The representation of a BMP encoder.

Implementations

`impl<'a, W: Write + 'a> BmpEncoder<'a, W>`

`pub fn new(w: &'a mut W) -> Self`

Create a new encoder that writes its output to “‘w“‘.

`pub fn encode( self: &mut Self, image: &[u8], width: u32, height: u32, c: color::ColorType ) -> ImageResult<()>`

Encodes the image “‘image“‘ that has dimensions “‘width“‘ and “‘height“‘ and “‘ColorType“‘ “‘c“‘.

`impl<'a, W> where W: Send Send for BmpEncoder<'a, W>`

`impl<'a, W> where W: Sync Sync for BmpEncoder<'a, W>`

`impl<'a, W> Unpin for BmpEncoder<'a, W>`

`impl<'a, W> !UnwindSafe for BmpEncoder<'a, W>`

```
impl<'a, W> where W: RefUnwindSafe RefUnwindSafe for BmpEncoder<'a, W>
```

```
impl<'a, W: Write> ImageEncoder for BmpEncoder<'a, W>
```

```
 fn write_image( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  color::Co
) -> ImageResult<()>
```

1.185 Module pub image::dds

-

1.186 Re-export pub DdsDecoder

```
pub use DdsDecoder;
```

1.187 Module pub image::dxt

-
-
-
-
-
-
-

1.188 Re-export pub DXTEncoder

```
pub use DXTEncoder;
```

1.189 Re-export pub DXTReader

```
pub use DXTReader;
```

1.190 Re-export pub DXTVariant

```
pub use DXTVariant;
```

1.191 Re-export pub DxtDecoder

```
pub use DxtDecoder;
```

1.192 Re-export pub DxtEncoder

```
pub use DxtEncoder;
```

1.193 Re-export pub DxtReader

```
pub use DxtReader;
```

1.194 Re-export pub DxtVariant

```
pub use DxtVariant;
```

1.195 Module pub image::farbfeld

-
-
-

1.196 Re-export pub FarbfeldDecoder

```
pub use FarbfeldDecoder;
```

1.197 Re-export pub FarbfeldEncoder

```
pub use FarbfeldEncoder;
```

1.198 Re-export pub FarbfeldReader

```
pub use FarbfeldReader;
```

1.199 Module pub image::gif

-
-
-
-
-

1.200 Re-export pub Encoder

```
pub use Encoder;
```

1.201 Re-export pub GifDecoder

```
pub use GifDecoder;
```

1.202 Re-export pub GifEncoder

```
pub use GifEncoder;
```

1.203 Re-export pub GifReader

```
pub use GifReader;
```

1.204 Re-export pub Repeat

```
pub use Repeat;
```

1.205 Module pub image::hdr

- image::codecs::hdr::read_raw_file
- image::codecs::hdr::rgbe8
- image::codecs::hdr::to_rgbe8
- image::codecs::hdr::HDRAdapter
- image::codecs::hdr::HDREncoder
- image::codecs::hdr::HDRImageDecoderIterator
- image::codecs::hdr::HDRMetadata
- image::codecs::hdr::HdrAdapter
- image::codecs::hdr::HdrDecoder
- image::codecs::hdr::HdrEncoder
- image::codecs::hdr::HdrImageDecoderIterator
- image::codecs::hdr::HdrMetadata
- image::codecs::hdr::HdrReader
- image::codecs::hdr::RGBE8Pixel
- image::codecs::hdr::Rgbe8Pixel
- image::codecs::hdr::SIGNATURE

1.206 Function pub image::codecs::hdr::read_raw_file

```
pub fn read_raw_file(path: P) -> ::std::io::Result<Vec<Rgb<f32>>>
```

Helper function for reading raw 3-channel f32 images

1.207 Function pub image::codecs::hdr::rgbe8

```
pub fn rgbe8(r: u8, g: u8, b: u8, e: u8) -> Rgbe8Pixel
```

Creates `RGBE8Pixel` from components

1.208 Function pub image::codecs::hdr::to_rgbe8

```
pub fn to_rgbe8(pix: Rgb<f32>) -> Rgbe8Pixel
```

Converts `Rgb<f32>` into `RGBE8Pixel`

1.209 Typedef pub image::codecs::hdr::HDRAdapter

```
pub type HDRAdapter = HdrAdapter<R>;
```

HDR Adapter

An alias of `struct.HdrAdapter.html`.

TODO: remove

1.210 Typedef pub image::codecs::hdr::HDREncoder

```
pub type HDREncoder = HdrEncoder<R>;
```

HDR Encoder

An alias of `struct.HdrEncoder.html`.

TODO: remove

1.211 Typedef pub image::codecs::hdr::HDRImageDecoderIterator

```
pub type HDRImageDecoderIterator = HdrImageDecoderIterator<R>;
```

Scanline buffered pixel by pixel iterator

An alias of `struct.HdrImageDecoderIterator.html`.

TODO: remove

1.212 Typedef pub image::codecs::hdr::HDRMetadata

```
pub type HDRMetadata = HdrMetadata;
```

HDR MetaData

An alias of `struct.HdrMetadata.html`.

TODO: remove

1.213 Struct pub image::codecs::hdr::HdrAdapter

```
pub struct HdrAdapter<R: BufRead>{
    // some fields omitted
}
```

Adapter to conform to `ImageDecoder` trait

Implementations

`impl<R: BufRead> HdrAdapter<R>`

`pub fn new(r: R) -> ImageResult<HdrAdapter<R»`

Creates adapter

`pub fn new_nonstrict(r: R) -> ImageResult<HdrAdapter<R»`

Allows reading old Radiance HDR images

`impl<R> where R: Send Send for HdrAdapter<R>`

`impl<R> where R: Sync Sync for HdrAdapter<R>`

`impl<R> where R: Unpin Unpin for HdrAdapter<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrAdapter<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrAdapter<R>`

`impl<'a, R: 'a + BufRead> ImageDecoder<'a> for HdrAdapter<R>`

`type Reader = HdrReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(self: Self) -> ImageResult«Self as >::Reader>`

`fn read_image(mut self: Self, buf: &mut [u8]) -> ImageResult<()>`

`impl<'a, R: 'a + BufRead + Seek> ImageDecoderExt<'a> for HdrAdapter<R>`

`fn read_rect_with_progress<F: Fn>( self: &mut Self, x: u32, y: u32, width: u32, height: u32, buf: &mut [u8], progress_callback: F ) -> ImageResult<()>`

`impl<R: $crate::fmt::Debug + BufRead> Debug for HdrAdapter<R>`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.214 Struct pub image::codecs::hdr::HdrDecoder

```
pub struct HdrDecoder<R>{
    // some fields omitted
}
```

An Radiance HDR decoder

Implementations

`impl<R: BufRead> HdrDecoder<R>`

`pub fn new(reader: R) -> ImageResult<HdrDecoder<R»`

Reads Radiance HDR image header from stream "'r"' if the header is valid, creates HdrDecoder strict mode is enabled

`pub fn with_strictness(mut reader: R, strict: bool) -> ImageResult<HdrDecoder<R»`

Reads Radiance HDR image header from stream "'reader"', if the header is valid, creates "'HdrDecoder"'.

strict enables strict mode

Warning! Reading wrong file in non-strict mode could consume file size worth of memory in the process.

`pub fn metadata(self: &Self) -> HdrMetadata`

Returns file metadata. Refer to "'HDRMetadata"' for details.

`pub fn read_image_native(mut self: Self) -> ImageResult<Vec<Rgbe8Pixel»`

Consumes decoder and returns a vector of RGBE8 pixels

`pub fn read_image_transform<T: Send, F: Send + Sync + Fn>(mut self: Self, f: F, output_slice:`
`&mut [T]) -> ImageResult<()>`

Consumes decoder and returns a vector of transformed pixels

`pub fn read_image_ldr(self: Self) -> ImageResult<Vec<Rgb<u8»>`

Consumes decoder and returns a vector of Rgb<u8> pixels. scale = 1, gamma = 2.2

`pub fn read_image_hdr(self: Self) -> ImageResult<Vec<Rgb<f32»>`

Consumes decoder and returns a vector of Rgb<f32> pixels.

`impl<R> where R: Send Send for HdrDecoder<R>`

`impl<R> where R: Sync Sync for HdrDecoder<R>`

`impl<R> where R: Unpin Unpin for HdrDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrDecoder<R>`

```
impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrDecoder<R>
```

```
impl<R: BufRead> IntoIterator for HdrDecoder<R>
```

```
 type Item = ImageResult<Rgbe8Pixel>;
```

```
 type IntoIter = HdrImageDecoderIterator<R>;
```

```
 fn into_iter(self:  Self) -> <Self as >::IntoIter
```

```
impl<R: $crate::fmt::Debug> Debug for HdrDecoder<R>
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

1.215 Struct pub image::codecs::hdr::HdrEncoder

```
pub struct HdrEncoder<W: Write>{
    // some fields omitted
}
```

Radiance HDR encoder

Implementations

```
impl<W: Write> HdrEncoder<W>
```

```
 pub fn new(w:  W) -> HdrEncoder<W>
```

Creates encoder

```
 pub fn encode(mut self:  Self, data:  &[Rgb<f32>], width:  usize, height:  usize) -> ImageResul
```

Encodes the image "'data"' that has dimensions "'width"' and "'height"'

```
impl<W> where W: Send Send for HdrEncoder<W>
```

```
impl<W> where W: Sync Sync for HdrEncoder<W>
```

```
impl<W> where W: Unpin Unpin for HdrEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for HdrEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for HdrEncoder<W>
```

1.216 Struct pub image::codecs::hdr::HdrImageDecoderIterator

```
pub struct HdrImageDecoderIterator<R: BufRead>{
    // some fields omitted
}
```

Scanline buffered pixel by pixel iterator

Implementations

`impl<R> where R: Send Send for HdrImageDecoderIterator<R>`

`impl<R> where R: Sync Sync for HdrImageDecoderIterator<R>`

`impl<R> where R: Unpin Unpin for HdrImageDecoderIterator<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrImageDecoderIterator<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrImageDecoderIterator<R>`

`impl<R: BufRead> ExactSizeIterator for HdrImageDecoderIterator<R>`

`impl<R: BufRead> Iterator for HdrImageDecoderIterator<R>`

`type Item = ImageResult<Rgbe8Pixel>;`

`fn next(self:  &mut Self) -> Option«Self as >::Item>`

`fn size_hint(self:  &Self) -> (usize,Option<usize>)`

1.217 Struct pub image::codecs::hdr::HdrMetadata

```
pub struct HdrMetadata{
    pub width: u32,
    pub height: u32,
    pub orientation: ((i8,i8),(i8,i8)),
    pub exposure: Option<f32>,
    pub color_correction: Option<(f32,f32,f32)>,
    pub pixel_aspect_ratio: Option<f32>,
    pub custom_attributes: Vec<(String,String)>,
}
```

Metadata for Radiance HDR image

Fields

`width:` u32 Width of decoded image. It could be either scanline length, or scanline count, depending on image orientation.

`height:` u32 Height of decoded image. It depends on orientation too.

`orientation:` `((i8,i8),(i8,i8))` Orientation matrix. For standard orientation it is ((1,0),(0,1)) - left to right, top to bottom. First pair tells how resulting pixel coordinates change along a scanline. Second pair tells how they change from one scanline to the next.

`exposure:` `Option<f32>` Divide color values by exposure to get to get physical radiance in watts/steradian/m<sup>2</sup>

Image may not contain physical data, even if this field is set.

`color_correction:` `Option<(f32,f32,f32)>` Divide color values by corresponding tuple member (r, g, b) to get to get physical radiance in watts/steradian/m<sup>2</sup>

Image may not contain physical data, even if this field is set.

`pixel_aspect_ratio:` `Option<f32>` Pixel height divided by pixel width

`custom_attributes:` `Vec<(String,String)>` All lines contained in image header are put here. Ordering of lines is preserved. Lines in the form "key=value" are represented as ("key", "value"). All other lines are ("", "line")

Implementations

`impl Send for HdrMetadata`

`impl Sync for HdrMetadata`

`impl Unpin for HdrMetadata`

`impl UnwindSafe for HdrMetadata`

`impl RefUnwindSafe for HdrMetadata`

`impl Clone for HdrMetadata`

` fn clone(self: &Self) -> HdrMetadata`

`impl Debug for HdrMetadata`

` fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.218 Struct pub image::codecs::hdr::HdrReader

```
pub struct HdrReader<R>(
    // some fields omitted
)
```

Wrapper struct around a `Cursor<Vec<u8»`

Implementations

`impl<R> where R: Send Send for HdrReader<R>`

`impl<R> where R: Sync Sync for HdrReader<R>`

`impl<R> where R: Unpin Unpin for HdrReader<R>`

`impl<R> where R: UnwindSafe UnwindSafe for HdrReader<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for HdrReader<R>`

`impl<R> Read for HdrReader<R>`

` fn read(self: &mut Self, buf: &mut [u8]) -> io::Result<usize>`

` fn read_to_end(self: &mut Self, buf: &mut Vec<u8>) -> io::Result<usize>`

1.219 Typedef pub image::codecs::hdr::RGBE8Pixel

```
pub type RGBE8Pixel = Rgbe8Pixel;
```

Refer to wikipedia

An alias of `struct.Rgbe8Pixel.html`.

TODO: remove

1.220 Struct pub image::codecs::hdr::Rgbe8Pixel

```
#[repr(C)]
pub struct Rgbe8Pixel{
    pub c: [u8; 3],
    pub e: u8,
}
```

Refer to wikipedia

Fields

`c: [u8; 3]` Color components

`e: u8` Exponent

Implementations

`impl Rgbe8Pixel`

`pub fn to_hdr(self: Self) -> Rgb<f32>`

Converts “‘RGBE8Pixel“‘ into “‘Rgb<f32>“‘ linearly

`pub fn to_ldr<T: Primitive + Zero>(self: Self) -> Rgb<T>`

Converts “‘RGBE8Pixel“‘ into “‘Rgb<T>“‘ with scale=1 and gamma=2.2

color_ldr = (color_hdr*scale)<sup>gamma</sup>

# Panic

Panics when “‘T::max_value()“‘ cannot be represented as f32.

`pub fn to_ldr_scale_gamma<T: Primitive + Zero>(self: Self, scale: f32, gamma: f32) -> Rgb<T>`

Converts RGBE8Pixel into Rgb<T> using provided scale and gamma

color_ldr = (color_hdr*scale)<sup>gamma</sup>

# Panic

Panics when T::max_value() cannot be represented as f32. Panics when scale or gamma is NaN

```
impl Send for Rgbe8Pixel
```

```
impl Sync for Rgbe8Pixel
```

```
impl Unpin for Rgbe8Pixel
```

```
impl UnwindSafe for Rgbe8Pixel
```

```
impl RefUnwindSafe for Rgbe8Pixel
```

```
impl Clone for Rgbe8Pixel
```

```
 fn clone(self:  &Self) -> Rgbe8Pixel
```

```
impl Copy for Rgbe8Pixel
```

```
impl Default for Rgbe8Pixel
```

```
 fn default() -> Rgbe8Pixel
```

```
impl Eq for Rgbe8Pixel
```

```
impl PartialEq<Rgbe8Pixel> for Rgbe8Pixel
```

```
 fn eq(self:  &Self, other:  &Rgbe8Pixel) -> bool
```

```
 fn ne(self:  &Self, other:  &Rgbe8Pixel) -> bool
```

```
impl Debug for Rgbe8Pixel
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl StructuralPartialEq for Rgbe8Pixel
```

```
impl StructuralEq for Rgbe8Pixel
```

1.221 Constant pub image::codecs::hdr::SIGNATURE

```
pub const SIGNATURE: &[u8] = b"#?RADIANCE";
```

Radiance HDR file signature

1.222 Module pub image::ico

- image::codecs::ico::ICOEncoder
- image::codecs::ico::IcoDecoder
- image::codecs::ico::IcoEncoder

1.223 Typedef pub image::codecs::ico::ICOEncoder

```
pub type ICOEncoder = IcoEncoder<W>;
```

ICO encoder

An alias of `struct.IcoEncoder.html`.

TODO: remove

1.224 Struct pub image::codecs::ico::IcoDecoder

```
pub struct IcoDecoder<R: Read>{
    // some fields omitted
}
```

An ico decoder

Implementations

`impl<R: Read + Seek> IcoDecoder<R>`

`pub fn new(mut r: R) -> ImageResult<IcoDecoder<R>>`

Create a new decoder that decodes from the stream “‘r“‘

`impl<R> where R: Send Send for IcoDecoder<R>`

`impl<R> where R: Sync Sync for IcoDecoder<R>`

`impl<R> where R: Unpin Unpin for IcoDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for IcoDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for IcoDecoder<R>`

`impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for IcoDecoder<R>`

`type Reader = IcoReader<R>;`

`fn dimensions(self: &Self) -> (u32,u32)`

`fn color_type(self: &Self) -> ColorType`

`fn into_reader(self: Self) -> ImageResult<<Self as >::Reader>`

`fn read_image(self: Self, buf: &mut [u8]) -> ImageResult<()>`

1.225 Struct pub image::codecs::ico::IcoEncoder

```
pub struct IcoEncoder<W: Write>{
    // some fields omitted
}
```

ICO encoder

Implementations

`impl<W: Write> IcoEncoder<W>`

`pub fn new(w: W) -> IcoEncoder<W>`

Create a new encoder that writes its output to ‘“w“‘.

`pub fn encode( mut self: Self, data: &[u8], width: u32, height: u32, color: ColorType ) -> ImageResult<()>`

Encodes the image ‘“image“‘ that has dimensions ‘“width“‘ and ‘“height“‘ and ‘“ColorType“‘ ‘“c“‘. The dimensions of the image must be between 1 and 256 (inclusive) or an error will be returned.

`impl<W> where W: Send Send for IcoEncoder<W>`

`impl<W> where W: Sync Sync for IcoEncoder<W>`

`impl<W> where W: Unpin Unpin for IcoEncoder<W>`

`impl<W> where W: UnwindSafe UnwindSafe for IcoEncoder<W>`

`impl<W> where W: RefUnwindSafe RefUnwindSafe for IcoEncoder<W>`

`impl<W: Write> ImageEncoder for IcoEncoder<W>`

`fn write_image( self: Self, buf: &[u8], width: u32, height: u32, color_type: ColorType ) -> ImageResult<()>`

1.226 Module pub image::jpeg

- image::codecs::jpeg::JPEGEncoder
- image::codecs::jpeg::JpegDecoder
- image::codecs::jpeg::JpegEncoder
- image::codecs::jpeg::PixelDensity
- image::codecs::jpeg::PixelDensityUnit

1.227 Typedef pub image::codecs::jpeg::JPEGEncoder

```
pub type JPEGEncoder = JpegEncoder<'a, W>;
```

JPEG Encoder

An alias of `struct.JpegEncoder.html`.

TODO: remove

1.228 Struct pub image::codecs::jpeg::JpegDecoder

```
pub struct JpegDecoder<R>{
    // some fields omitted
}
```

JPEG decoder

Implementations

`impl<R: Read> JpegDecoder<R>`

`pub fn new(r:  R) -> ImageResult<JpegDecoder<R>>`

Create a new decoder that decodes from the stream "'r"'

`pub fn scale( self:  &mut Self, requested_width:  u16, requested_height:  u16 ) -> ImageResult<`

Configure the decoder to scale the image during decoding.

This efficiently scales the image by the smallest supported scale factor that produces an image larger than or equal to the requested size in at least one axis. The currently implemented scale factors are 1/8, 1/4, 1/2 and 1.

To generate a thumbnail of an exact size, pass the desired size and then scale to the final size using a traditional resampling algorithm.

The size of the image to be loaded, with the scale factor applied, is returned.

`impl<R> where R: Send Send for JpegDecoder<R>`

`impl<R> where R: Sync Sync for JpegDecoder<R>`

`impl<R> where R: Unpin Unpin for JpegDecoder<R>`

`impl<R> where R: UnwindSafe UnwindSafe for JpegDecoder<R>`

`impl<R> where R: RefUnwindSafe RefUnwindSafe for JpegDecoder<R>`

`impl<'a, R: 'a + Read> ImageDecoder<'a> for JpegDecoder<R>`

`type Reader = JpegReader<R>;`

`fn dimensions(self:  &Self) -> (u32,u32)`

`fn color_type(self:  &Self) -> ColorType`

`fn into_reader(mut self:  Self) -> ImageResult<<Self as >::Reader>`

`fn read_image(mut self:  Self, buf:  &mut [u8]) -> ImageResult<()>`

1.229 Struct pub image::codecs::jpeg::JpegEncoder

```rust
pub struct JpegEncoder<'a, W: 'a>{
    // some fields omitted
}
```

The representation of a JPEG encoder

Implementations

`impl<'a, W: Write> JpegEncoder<'a, W>`

`pub fn new(w: &mut W) -> JpegEncoder<'_, W>`

Create a new encoder that writes its output to "'w"'

`pub fn new_with_quality(w: &mut W, quality: u8) -> JpegEncoder<'_, W>`

Create a new encoder that writes its output to "'w"', and has the quality parameter "'quality"' with a value in the range 1-100 where 1 is the worst and 100 is the best.

`pub fn set_pixel_density(self: &mut Self, pixel_density: PixelDensity)`

Set the pixel density of the images the encoder will encode. If this method is not called, then a default pixel aspect ratio of 1x1 will be applied, and no DPI information will be stored in the image.

`pub fn encode( self: &mut Self, image: &[u8], width: u32, height: u32, color_type: ColorType ) -> ImageResult<()>`

Encodes the image stored in the raw byte buffer "'image"' that has dimensions "'width"' and "'height"' and "'ColorType"' "'c"'

The Image in encoded with subsampling ratio 4:2:2

`pub fn encode_image<I: GenericImageView>(self: &mut Self, image: &I) -> ImageResult<()>`

Encodes the given image.

As a special feature this does not require the whole image to be present in memory at the same time such that it may be computed on the fly, which is why this method exists on this encoder but not on others. Instead the encoder will iterate over 8-by-8 blocks of pixels at a time, inspecting each pixel exactly once. You can rely on this behaviour when calling this method.

The Image in encoded with subsampling ratio 4:2:2

`impl<'a, W> where W: Send Send for JpegEncoder<'a, W>`

`impl<'a, W> where W: Sync Sync for JpegEncoder<'a, W>`

`impl<'a, W> Unpin for JpegEncoder<'a, W>`

`impl<'a, W> !UnwindSafe for JpegEncoder<'a, W>`

`impl<'a, W> where W: RefUnwindSafe RefUnwindSafe for JpegEncoder<'a, W>`

```
impl<'a, W: Write> ImageEncoder for JpegEncoder<'a, W>
```

```
 fn write_image( mut self: Self, buf: &[u8], width: u32, height: u32, color_type: ColorType
) -> ImageResult<()>
```

1.230 Struct pub image::codecs::jpeg::PixelDensity

```
pub struct PixelDensity{
    pub density: (u16,u16),
    pub unit: PixelDensityUnit,
}
```

Represents the pixel density of an image

For example, a 300 DPI image is represented by:

```
use image::jpeg::*;
let hdpi = PixelDensity::dpi(300);
assert_eq!(hdpi, PixelDensity {density: (300,300), unit: PixelDensityUnit::Inches})
```

Fields

`density: (u16,u16)` A couple of values for (Xdensity, Ydensity)

`unit: PixelDensityUnit` The unit in which the density is measured

Implementations

`impl PixelDensity`

` pub fn dpi(density: u16) -> Self`

Creates the most common pixel density type: the horizontal and the vertical density are equal, and measured in pixels per inch.

`impl Send for PixelDensity`

`impl Sync for PixelDensity`

`impl Unpin for PixelDensity`

`impl UnwindSafe for PixelDensity`

`impl RefUnwindSafe for PixelDensity`

`impl Clone for PixelDensity`

` fn clone(self: &Self) -> PixelDensity`

`impl Copy for PixelDensity`

`impl Default for PixelDensity`

` fn default() -> Self`

Returns a pixel density with a pixel aspect ratio of 1

`impl Eq for PixelDensity`

`impl PartialEq<PixelDensity> for PixelDensity`

`fn eq(self: &Self, other: &PixelDensity) -> bool`

`fn ne(self: &Self, other: &PixelDensity) -> bool`

`impl Debug for PixelDensity`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for PixelDensity`

`impl StructuralEq for PixelDensity`

1.231 Enum pub image::codecs::jpeg::PixelDensityUnit

```
pub enum image::codecs::jpeg::PixelDensityUnit {
    PixelAspectRatio,
    Inches,
    Centimeters,
}
```

Represents a unit in which the density of an image is measured

Variants

`PixelAspectRatio` Represents the absence of a unit, the values indicate only a pixel aspect ratio

`Inches` Pixels per inch (2.54 cm)

`Centimeters` Pixels per centimeter

Implementations

`impl Send for PixelDensityUnit`

`impl Sync for PixelDensityUnit`

`impl Unpin for PixelDensityUnit`

`impl UnwindSafe for PixelDensityUnit`

`impl RefUnwindSafe for PixelDensityUnit`

`impl Clone for PixelDensityUnit`

`fn clone(self: &Self) -> PixelDensityUnit`

`impl Copy for PixelDensityUnit`

```
impl Eq for PixelDensityUnit

impl PartialEq<PixelDensityUnit> for PixelDensityUnit

 fn eq(self:  &Self, other:  &PixelDensityUnit) -> bool

impl Debug for PixelDensityUnit

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for PixelDensityUnit

impl StructuralEq for PixelDensityUnit
```

1.232 Module pub image::png

-
-
-
-
-
-
-
-

1.233 Re-export pub ApngDecoder

```
pub use ApngDecoder;
```

1.234 Re-export pub CompressionType

```
pub use CompressionType;
```

1.235 Re-export pub FilterType

```
pub use FilterType;
```

1.236 Re-export pub PNGEncoder

```
pub use PNGEncoder;
```

1.237 Re-export pub PNGReader

```
pub use PNGReader;
```

1.238 Re-export pub PngDecoder

```
pub use PngDecoder;
```

1.239 Re-export pub PngEncoder

```
pub use PngEncoder;
```

1.240 Re-export pub PngReader

```
pub use PngReader;
```

1.241 Module pub image::pnm

- image::codecs::pnm::ArbitraryHeader
- image::codecs::pnm::ArbitraryTuplType
- image::codecs::pnm::BitmapHeader
- image::codecs::pnm::GraymapHeader
- image::codecs::pnm::PNMEncoder
- image::codecs::pnm::PNMHeader
- image::codecs::pnm::PNMSubtype
- image::codecs::pnm::PixmapHeader
- image::codecs::pnm::PnmDecoder
- image::codecs::pnm::PnmEncoder
- image::codecs::pnm::PnmHeader
- image::codecs::pnm::PnmSubtype
- image::codecs::pnm::SampleEncoding

1.242 Struct pub image::codecs::pnm::ArbitraryHeader

```
pub struct ArbitraryHeader{
    pub height: u32,
    pub width: u32,
    pub depth: u32,
    pub maxval: u32,
    pub tupltype: Option<ArbitraryTuplType>,
}
```

Header produced by a `pam` file ("Portable Arbitrary Map")

Fields

`height:` u32 Height of the image file

`width:` u32 Width of the image file

`depth:` u32 Number of color channels

`maxval:` u32 Maximum sample value within the image `tupltype: Option<ArbitraryTuplType>` Color interpretation of image pixels

Implementations

`impl Send for ArbitraryHeader`

`impl Sync for ArbitraryHeader`

`impl Unpin for ArbitraryHeader`

`impl UnwindSafe for ArbitraryHeader`

`impl RefUnwindSafe for ArbitraryHeader`

`impl From<ArbitraryHeader> for PnmHeader`

`fn from(header: ArbitraryHeader) -> Self`

`impl Clone for ArbitraryHeader`

`fn clone(self: &Self) -> ArbitraryHeader`

`impl Debug for ArbitraryHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.243 Enum pub image::codecs::pnm::ArbitraryTuplType

```
pub enum image::codecs::pnm::ArbitraryTuplType {
    BlackAndWhite,
    BlackAndWhiteAlpha,
    Grayscale,
    GrayscaleAlpha,
    RGB,
    RGBAlpha,
    Custom,
}
```

Standardized tuple type specifiers in the header of a `pam`.

Variants

`BlackAndWhite` Pixels are either black (0) or white (1)

`BlackAndWhiteAlpha` Pixels are either black (0) or white (1) and a second alpha channel

`Grayscale` Pixels represent the amount of white

`GrayscaleAlpha` Grayscale with an additional alpha channel

`RGB` Three channels: Red, Green, Blue

`RGBAlpha` Four channels: Red, Green, Blue, Alpha

`Custom` An image format which is not standardized

Implementations

`impl Send for ArbitraryTuplType`

`impl Sync for ArbitraryTuplType`

`impl Unpin for ArbitraryTuplType`

`impl UnwindSafe for ArbitraryTuplType`

`impl RefUnwindSafe for ArbitraryTuplType`

`impl Clone for ArbitraryTuplType`

` fn clone(self: &Self) -> ArbitraryTuplType`

`impl Debug for ArbitraryTuplType`

` fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.244 Struct pub image::codecs::pnm::BitmapHeader

```
pub struct BitmapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
}
```

Header produced by a `pbm` file ("Portable Bit Map")

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height:` u32 Height of the image file

`width:` u32 Width of the image file

Implementations

`impl Send for BitmapHeader`

`impl Sync for BitmapHeader`

`impl Unpin for BitmapHeader`

`impl UnwindSafe for BitmapHeader`

`impl RefUnwindSafe for BitmapHeader`

`impl From<BitmapHeader> for PnmHeader`

` fn from(header: BitmapHeader) -> Self` `impl Clone for BitmapHeader`

`fn clone(self: &Self) -> BitmapHeader`

`impl Copy for BitmapHeader`

`impl Debug for BitmapHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.245 Struct pub image::codecs::pnm::GraymapHeader

```
pub struct GraymapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
    pub maxwhite: u32,
}
```

Header produced by a `pgm` file ("Portable Gray Map")

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height:` u32 Height of the image file

`width:` u32 Width of the image file

`maxwhite:` u32 Maximum sample value within the image

Implementations

`impl Send for GraymapHeader`

`impl Sync for GraymapHeader`

`impl Unpin for GraymapHeader`

`impl UnwindSafe for GraymapHeader`

`impl RefUnwindSafe for GraymapHeader`

`impl From<GraymapHeader> for PnmHeader`

`fn from(header: GraymapHeader) -> Self`

`impl Clone for GraymapHeader`

`fn clone(self: &Self) -> GraymapHeader`

`impl Copy for GraymapHeader`

`impl Debug for GraymapHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.246 Typedef pub image::codecs::pnm::PNMEncoder

```
pub type PNMEncoder = PnmEncoder<W>;
```

PNM Encoder

An alias of `struct.PnmEncoder.html`.

TODO: remove

1.247 Typedef pub image::codecs::pnm::PNMHeader

```
pub type PNMHeader = PnmHeader;
```

PNM Header

An alias of `struct.PnmHeader.html`.

TODO: remove

1.248 Enum pub image::codecs::pnm::PNMSubtype

```
pub enum image::codecs::pnm::PNMSubtype {
    Bitmap,
    Graymap,
    Pixmap,
    ArbitraryMap,
}
```

Denotes the category of the magic number

DEPRECATED: The name of this enum will be changed to `type.PnmSubtype.html`.

TODO: rename to `type.PnmSubtype.html`.

Variants

`Bitmap` Magic numbers P1 and P4

`Graymap` Magic numbers P2 and P5

`Pixmap` Magic numbers P3 and P6

`ArbitraryMap` Magic number P7

Implementations

`impl PNMSubtype`

`pub fn magic_constant(self: Self) -> &'static [u8; 2]`

Get the two magic constant bytes corresponding to this format subtype.

`pub fn sample_encoding(self: Self) -> SampleEncoding`

Whether samples are stored as binary or as decimal ascii

`impl Send for PNMSubtype`

`impl Sync for PNMSubtype`

`impl Unpin for PNMSubtype`

`impl UnwindSafe for PNMSubtype`

`impl RefUnwindSafe for PNMSubtype`

`impl Clone for PNMSubtype`

`fn clone(self: &Self) -> PNMSubtype`

`impl Copy for PNMSubtype`

`impl Eq for PNMSubtype`

`impl PartialEq<PNMSubtype> for PNMSubtype`

`fn eq(self: &Self, other: &PNMSubtype) -> bool`

`fn ne(self: &Self, other: &PNMSubtype) -> bool`

`impl Debug for PNMSubtype`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for PNMSubtype`

`impl StructuralEq for PNMSubtype`

1.249 Struct pub image::codecs::pnm::PixmapHeader

```
pub struct PixmapHeader{
    pub encoding: SampleEncoding,
    pub height: u32,
    pub width: u32,
    pub maxval: u32,
}
```

Header produced by a ppm file (“Portable Pixel Map”)

Fields

`encoding: SampleEncoding` Binary or Ascii encoded file

`height: u32` Height of the image file

`width: u32` Width of the image file

`maxval:` u32 Maximum sample value within the image

Implementations

`impl Send for PixmapHeader`

`impl Sync for PixmapHeader`

`impl Unpin for PixmapHeader`

`impl UnwindSafe for PixmapHeader`

`impl RefUnwindSafe for PixmapHeader`

`impl From<PixmapHeader> for PnmHeader`

`fn from(header: PixmapHeader) -> Self`

`impl Clone for PixmapHeader`

`fn clone(self: &Self) -> PixmapHeader`

`impl Copy for PixmapHeader`

`impl Debug for PixmapHeader`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.250 Struct pub image::codecs::pnm::PnmDecoder

```
pub struct PnmDecoder<R>{
    // some fields omitted
}
```

PNM decoder

Implementations

`impl<R: Read> PnmDecoder<R>`

`pub fn new(read: R) -> ImageResult<PnmDecoder<R»`

Create a new decoder that decodes from the stream "'read"'

`pub fn into_inner(self: Self) -> (R,PnmHeader)`

Extract the reader and header after an image has been read.

`impl<R: Read> PnmDecoder<R>`

`pub fn subtype(self: &Self) -> PNMSubtype`

Get the pnm subtype, depending on the magic constant contained in the header

```
impl<R> where R: Send Send for PnmDecoder<R>

impl<R> where R: Sync Sync for PnmDecoder<R>

impl<R> where R: Unpin Unpin for PnmDecoder<R>

impl<R> where R: UnwindSafe UnwindSafe for PnmDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for PnmDecoder<R>

impl<'a, R: 'a + Read> ImageDecoder<'a> for PnmDecoder<R>

 type Reader = PnmReader<R>;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn color_type(self:  &Self) -> ColorType

 fn original_color_type(self:  &Self) -> ExtendedColorType

 fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>

 fn read_image(mut self:  Self, buf:  &mut [u8]) -> ImageResult<()>
```

1.251 Struct pub image::codecs::pnm::PnmEncoder

```
pub struct PnmEncoder<W: Write>{
    // some fields omitted
}
```

Encodes images to any of the pnm image formats.

Implementations

```
impl<W: Write> PnmEncoder<W>

 pub fn new(writer:  W) -> Self
```

Create new PNMEncoder from the 'writer'.

The encoded images will have some 'pnm' format. If more control over the image type is required, use either one of 'with_subtype' or 'with_header'. For more information on the behaviour, see 'with_dynamic_header'.

```
 pub fn with_subtype(self:  Self, subtype:  PNMSubtype) -> Self
```

Encode a specific pnm subtype image.

The magic number and encoding type will be chosen as provided while the rest of the header data will be generated dynamically. Trying to encode incompatible images (e.g. encoding an RGB image as Graymap) will result in an error.

This will overwrite the effect of earlier calls to 'with_header' and 'with_dynamic_header'.

```
pub fn with_header(self: Self, header: PnmHeader) -> Self
```

Enforce the use of a chosen header.

While this option gives the most control over the actual written data, the encoding process will error in case the header data and image parameters do not agree. It is the users obligation to ensure that the width and height are set accordingly, for example.

Choose this option if you want a lossless decoding/encoding round trip.

This will overwrite the effect of earlier calls to 'with_subtype' and 'with_dynamic_header'.

```
pub fn with_dynamic_header(self: Self) -> Self
```

Create the header dynamically for each image.

This is the default option upon creation of the encoder. With this, most images should be encodable but the specific format chosen is out of the users control. The pnm subtype is chosen arbitrarily by the library.

This will overwrite the effect of earlier calls to 'with_subtype' and 'with_header'.

```
pub fn encode<'s, S> where S: Into<FlatSamples<'s»(self: &mut Self, image: S, width:
u32, height: u32, color: ColorType) -> ImageResult<()>
```

Encode an image whose samples are represented as 'u8'.

Some 'pnm' subtypes are incompatible with some color options, a chosen header most certainly with any deviation from the original decoded image.

```
impl<W> where W: Send Send for PnmEncoder<W>
```

```
impl<W> where W: Sync Sync for PnmEncoder<W>
```

```
impl<W> where W: Unpin Unpin for PnmEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for PnmEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for PnmEncoder<W>
```

```
impl<W: Write> ImageEncoder for PnmEncoder<W>
```

```
fn write_image( mut self: Self, buf: &[u8], width: u32, height: u32, color_type: ColorType
) -> ImageResult<()>
```

1.252 Struct pub image::codecs::pnm::PnmHeader

```rust
pub struct PnmHeader{
    // some fields omitted
}
```

Stores the complete header data of a file.

Internally, provides mechanisms for lossless reencoding. After reading a file with the decoder it is possible to recover the header and construct an encoder. Using the encoder on the just loaded image should result in a byte copy of the original file (for single image pnms without additional trailing data).

Implementations

```
impl PnmHeader
```

```
 pub fn subtype(self:  &Self) -> PNMSubtype
```

Retrieve the format subtype from which the header was created.

```
 pub fn width(self:  &Self) -> u32
```

The width of the image this header is for.

```
 pub fn height(self:  &Self) -> u32
```

The height of the image this header is for.

```
 pub fn maximal_sample(self:  &Self) -> u32
```

The biggest value a sample can have. In other words, the colour resolution.

```
 pub fn as_bitmap(self:  &Self) -> Option<&BitmapHeader>
```

Retrieve the underlying bitmap header if any

```
 pub fn as_graymap(self:  &Self) -> Option<&GraymapHeader>
```

Retrieve the underlying graymap header if any

```
 pub fn as_pixmap(self:  &Self) -> Option<&PixmapHeader>
```

Retrieve the underlying pixmap header if any

```
 pub fn as_arbitrary(self:  &Self) -> Option<&ArbitraryHeader>
```

Retrieve the underlying arbitrary header if any

```
 pub fn write(self:  &Self, writer:  &mut io::Write) -> io::Result<()>
```

Write the header back into a binary stream

```
impl Send for PnmHeader
```

```
impl Sync for PnmHeader
```

```
impl Unpin for PnmHeader
```

```
impl UnwindSafe for PnmHeader
```

```
impl RefUnwindSafe for PnmHeader
impl From<BitmapHeader> for PnmHeader
```

```
 fn from(header:  BitmapHeader) -> Self
```

```
impl From<GraymapHeader> for PnmHeader
```

```
 fn from(header:  GraymapHeader) -> Self
```

```
impl From<PixmapHeader> for PnmHeader
```

```
 fn from(header:  PixmapHeader) -> Self
```

```
impl From<ArbitraryHeader> for PnmHeader
```

```
 fn from(header:  ArbitraryHeader) -> Self
```

1.253 Typedef pub image::codecs::pnm::PnmSubtype

```rust
pub type PnmSubtype = PNMSubtype;
```

PNM Subtype

An alias of `enum.PNMSubtype.html`.

TODO: remove when [`DXTVariant`] is renamed.

1.254 Enum pub image::codecs::pnm::SampleEncoding

```rust
pub enum image::codecs::pnm::SampleEncoding {
    Binary,
    Ascii,
}
```

The kind of encoding used to store sample values

Variants

`Binary` Samples are unsigned binary integers in big endian

`Ascii` Samples are encoded as decimal ascii strings separated by whitespace

Implementations

```
impl Send for SampleEncoding
```

```
impl Sync for SampleEncoding
```

```
impl Unpin for SampleEncoding
```

```
impl UnwindSafe for SampleEncoding
```

```
impl RefUnwindSafe for SampleEncoding
impl Clone for SampleEncoding

 fn clone(self:  &Self) -> SampleEncoding

impl Copy for SampleEncoding

impl Eq for SampleEncoding

impl PartialEq<SampleEncoding> for SampleEncoding

 fn eq(self:  &Self, other:  &SampleEncoding) -> bool

impl Debug for SampleEncoding

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for SampleEncoding

impl StructuralEq for SampleEncoding
```

1.255 Module pub image::tga

- image::codecs::tga::TgaDecoder
- image::codecs::tga::TgaEncoder

1.256 Struct pub image::codecs::tga::TgaDecoder

```
pub struct TgaDecoder<R>{
    // some fields omitted
}
```

The representation of a TGA decoder

Implementations

```
impl<R: Read + Seek> TgaDecoder<R>

 pub fn new(r:  R) -> ImageResult<TgaDecoder<R»
```

Create a new decoder that decodes from the stream ‘r‘

```
impl<R> where R: Send Send for TgaDecoder<R>

impl<R> where R: Sync Sync for TgaDecoder<R>

impl<R> where R: Unpin Unpin for TgaDecoder<R>

impl<R> where R: UnwindSafe UnwindSafe for TgaDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for TgaDecoder<R>
```

```
impl<'a, R: 'a + Read + Seek> ImageDecoder<'a> for TgaDecoder<R>
```

```
 type Reader = TGAReader<R>;
```

```
 fn dimensions(self:  &Self) -> (u32,u32)
```

```
 fn color_type(self:  &Self) -> ColorType
```

```
 fn scanline_bytes(self:  &Self) -> u64
```

```
 fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>
```

```
 fn read_image(mut self:  Self, buf:  &mut [u8]) -> ImageResult<()>
```

1.257 Struct pub image::codecs::tga::TgaEncoder

```
pub struct TgaEncoder<W: Write>{
    // some fields omitted
}
```

TGA encoder.

Implementations

```
impl<W: Write> TgaEncoder<W>
```

```
 pub fn new(w:  W) -> TgaEncoder<W>
```

Create a new encoder that writes its output to "'w"'.

```
 pub fn encode( mut self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

Encodes the image "'buf"' that has dimensions "'width"' and "'height"' and "'ColorType"' "'color_type"'.

The dimensions of the image must be between 0 and 65535 (inclusive) or an error will be returned.

```
impl<W> where W: Send Send for TgaEncoder<W>
```

```
impl<W> where W: Sync Sync for TgaEncoder<W>
```

```
impl<W> where W: Unpin Unpin for TgaEncoder<W>
```

```
impl<W> where W: UnwindSafe UnwindSafe for TgaEncoder<W>
```

```
impl<W> where W: RefUnwindSafe RefUnwindSafe for TgaEncoder<W>
```

```
impl<W: Write> ImageEncoder for TgaEncoder<W>
```

```
 fn write_image( self:  Self, buf:  &[u8], width:  u32, height:  u32, color_type:  ColorType
) -> ImageResult<()>
```

1.258 Module pub image::tiff

-
-
-

1.259 Re-export pub TiffDecoder

```rust
pub use TiffDecoder;
```

1.260 Re-export pub TiffEncoder

```rust
pub use TiffEncoder;
```

1.261 Re-export pub TiffReader

```rust
pub use TiffReader;
```

1.262 Module pub image::webp

-
- image::codecs::webp::WebPDecoder

1.263 Re-export pub vp8

```rust
pub use vp8;
```

1.264 Struct pub image::codecs::webp::WebPDecoder

```rust
pub struct WebPDecoder<R>{
    // some fields omitted
}
```

WebP Image format decoder. Currently only supportes the luma channel (meaning that decoded images will be grayscale).

Implementations

`impl<R: Read> WebPDecoder<R>`

`pub fn new(r:  R) -> ImageResult<WebPDecoder<R»`

Create a new WebPDecoder from the Reader "'r"'. This function takes ownership of the Reader.

```
impl<R> where R: Send Send for WebPDecoder<R>

impl<R> where R: Sync Sync for WebPDecoder<R>

impl<R> where R: Unpin Unpin for WebPDecoder<R>

impl<R> where R: UnwindSafe UnwindSafe for WebPDecoder<R>

impl<R> where R: RefUnwindSafe RefUnwindSafe for WebPDecoder<R>

impl<'a, R: 'a + Read> ImageDecoder<'a> for WebPDecoder<R>

 type Reader = WebpReader<R>;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn color_type(self:  &Self) -> color::ColorType

 fn into_reader(self:  Self) -> ImageResult«Self as >::Reader>

 fn read_image(self:  Self, buf:  &mut [u8]) -> ImageResult<()>
```

1.265 Enum pub image::ColorType

```
pub enum image::ColorType {
    L8,
    La8,
    Rgb8,
    Rgba8,
    L16,
    La16,
    Rgb16,
    Rgba16,
    Bgr8,
    Bgra8,
    // some variants omitted
}
```

An enumeration over supported color types and bit depths

Variants

L8 Pixel is 8-bit luminance

La8 Pixel is 8-bit luminance with an alpha channel

Rgb8 Pixel contains 8-bit R, G and B channels

Rgba8 Pixel is 8-bit RGB with an alpha channel

L16 Pixel is 16-bit luminance

La16 Pixel is 16-bit luminance with an alpha channel

Rgb16 Pixel is 16-bit RGB

`Rgba16` Pixel is 16-bit RGBA

`Bgr8` Pixel contains 8-bit B, G and R channels

`Bgra8` Pixel is 8-bit BGR with an alpha channel

Implementations

`impl ColorType`

` pub fn bytes_per_pixel(self:  Self) -> u8`

Returns the number of bytes contained in a pixel of 'ColorType' "'c"'

` pub fn has_alpha(self:  Self) -> bool`

Returns if there is an alpha channel.

` pub fn has_color(self:  Self) -> bool`

Returns false if the color scheme is grayscale, true otherwise.

` pub fn bits_per_pixel(self:  Self) -> u16`

Returns the number of bits contained in a pixel of 'ColorType' "'c"' (which will always be a multiple of 8).

` pub fn channel_count(self:  Self) -> u8`

Returns the number of color channels that make up this pixel

`impl Send for ColorType`

`impl Sync for ColorType`

`impl Unpin for ColorType`

`impl UnwindSafe for ColorType`

`impl RefUnwindSafe for ColorType`

`impl From<ColorType> for ExtendedColorType`

` fn from(c:  ColorType) -> Self`

`impl Clone for ColorType`

` fn clone(self:  &Self) -> ColorType`

`impl Copy for ColorType`

`impl Eq for ColorType`

`impl PartialEq<ColorType> for ColorType`

```
fn eq(self: &Self, other: &ColorType) -> bool
```

```
fn ne(self: &Self, other: &ColorType) -> bool
```

```
impl Debug for ColorType
```

```
fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Hash for ColorType
```

```
fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()
```

```
impl StructuralPartialEq for ColorType
```

```
impl StructuralEq for ColorType
```

1.266 Enum pub image::ExtendedColorType

```
pub enum image::ExtendedColorType {
    L1,
    La1,
    Rgb1,
    Rgba1,
    L2,
    La2,
    Rgb2,
    Rgba2,
    L4,
    La4,
    Rgb4,
    Rgba4,
    L8,
    La8,
    Rgb8,
    Rgba8,
    L16,
    La16,
    Rgb16,
    Rgba16,
    Bgr8,
    Bgra8,
    Unknown,
    // some variants omitted
}
```

An enumeration of color types encountered in image formats.

This is not exhaustive over all existing image formats but should be granular enough to allow round tripping of decoding and encoding as much as possible. The variants will be extended as necessary to enable this.

Another purpose is to advise users of a rough estimate of the accuracy and effort of the decoding from and encoding to such an image format.

Variants

`L1` Pixel is 1-bit luminance

`La1` Pixel is 1-bit luminance with an alpha channel

`Rgb1` Pixel contains 1-bit R, G and B channels

`Rgba1` Pixel is 1-bit RGB with an alpha channel

`L2` Pixel is 2-bit luminance

`La2` Pixel is 2-bit luminance with an alpha channel

`Rgb2` Pixel contains 2-bit R, G and B channels

`Rgba2` Pixel is 2-bit RGB with an alpha channel

`L4` Pixel is 4-bit luminance

`La4` Pixel is 4-bit luminance with an alpha channel

`Rgb4` Pixel contains 4-bit R, G and B channels

`Rgba4` Pixel is 4-bit RGB with an alpha channel

`L8` Pixel is 8-bit luminance

`La8` Pixel is 8-bit luminance with an alpha channel

`Rgb8` Pixel contains 8-bit R, G and B channels

`Rgba8` Pixel is 8-bit RGB with an alpha channel

`L16` Pixel is 16-bit luminance

`La16` Pixel is 16-bit luminance with an alpha channel

`Rgb16` Pixel contains 16-bit R, G and B channels

`Rgba16` Pixel is 16-bit RGB with an alpha channel

`Bgr8` Pixel contains 8-bit B, G and R channels

`Bgra8` Pixel is 8-bit BGR with an alpha channel

`Unknown` Pixel is of unknown color type with the specified bits per pixel. This can apply to pixels which are associated with an external palette. In that case, the pixel value is an index into the palette.

Implementations

```
impl ExtendedColorType
```

```
 pub fn channel_count(self:  Self) -> u8
```

Get the number of channels for colors of this type.

Note that the ‘Unknown‘ variant returns a value of ‘1‘ since pixels can only be treated as an opaque datum by the library.

```
impl Send for ExtendedColorType
impl Sync for ExtendedColorType

impl Unpin for ExtendedColorType

impl UnwindSafe for ExtendedColorType

impl RefUnwindSafe for ExtendedColorType

impl From<ColorType> for ExtendedColorType

 fn from(c:  ColorType) -> Self

impl Clone for ExtendedColorType

 fn clone(self:  &Self) -> ExtendedColorType

impl Copy for ExtendedColorType

impl Eq for ExtendedColorType

impl PartialEq<ExtendedColorType> for ExtendedColorType

 fn eq(self:  &Self, other:  &ExtendedColorType) -> bool

 fn ne(self:  &Self, other:  &ExtendedColorType) -> bool

impl Debug for ExtendedColorType

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Hash for ExtendedColorType

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl StructuralPartialEq for ExtendedColorType

impl StructuralEq for ExtendedColorType
```

1.267 Struct pub image::Luma

```
#[repr(C)]
#[allow(missing_docs)]
pub struct Luma<T: Primitive>(
    pub [T; 1],
)
```

Grayscale colors

Fields

`0:  [T; 1]`

Implementations

```
impl<T> where T: Send Send for Luma<T>

impl<T> where T: Sync Sync for Luma<T>

impl<T> where T: Unpin Unpin for Luma<T>

impl<T> where T: UnwindSafe UnwindSafe for Luma<T>

impl<T> where T: RefUnwindSafe RefUnwindSafe for Luma<T>

impl<T: Primitive + 'static> Pixel for Luma<T>

 type Subpixel = T;

 const CHANNEL_COUNT: u8 = 1;

 const COLOR_MODEL: &'static str = "Y";

 const COLOR_TYPE: ColorType = [ColorType::L8, ColorType::L16][(std::mem::size_of::<T>()
> 1) as usize];

 fn channels(self: &Self) -> &[T]

 fn channels_mut(self: &mut Self) -> &mut [T]

 fn channels4(self: &Self) -> (T,T,T,T)

 fn from_channels(a: T, b: T, c: T, d: T) -> Luma<T>

 fn from_slice(slice: &[T]) -> &Luma<T>

 fn from_slice_mut(slice: &mut [T]) -> &mut Luma<T>

 fn to_rgb(self: &Self) -> Rgb<T>

 fn to_bgr(self: &Self) -> Bgr<T>

 fn to_rgba(self: &Self) -> Rgba<T>

 fn to_bgra(self: &Self) -> Bgra<T>

 fn to_luma(self: &Self) -> Luma<T>

 fn to_luma_alpha(self: &Self) -> LumaA<T>

 fn map<F> where F: FnMut(self: &Self, f: F) -> Luma<T>

 fn apply<F> where F: FnMut(self: &mut Self, mut f: F)

 fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self: &Self, f: F, g: G) -> Luma<T>

 fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self: &mut Self, mut f: F, mut g:
G)

 fn map2<F> where F: FnMut(self: &Self, other: &Self, f: F) -> Luma<T>
 fn apply2<F> where F: FnMut(self: &mut Self, other: &Luma<T>, mut f: F)

 fn invert(self: &mut Self)

 fn blend(self: &mut Self, other: &Luma<T>)

impl<T: Primitive + 'static> From<[T; 1]> for Luma<T>

 fn from(c: [T; 1]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for Luma<T>

 fn clone(self: &Self) -> Luma<T>

impl<T: $crate::marker::Copy + Primitive> Copy for Luma<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for Luma<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<Luma<T» for Luma<T>

 fn eq(self: &Self, other: &Luma<T>) -> bool

 fn ne(self: &Self, other: &Luma<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for Luma<T>

 fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for Luma<T>

 type Output = T;

 fn index(self: &Self, _index: usize) -> &T

impl<T: Primitive> IndexMut<usize> for Luma<T>

 fn index_mut(self: &mut Self, _index: usize) -> &mut T

impl<T: $crate::hash::Hash + Primitive> Hash for Luma<T>

 fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for Luma<T>

impl<T: Primitive> StructuralEq for Luma<T>
```

1.268 Struct pub image::LumaA

```
#[repr(C)]
#[allow(missing_docs)]
pub struct LumaA<T: Primitive>(
    pub [T; 2],
)
```

Grayscale colors + alpha channel

Fields

```
0:  [T; 2]
```

Implementations

```
impl<T> where T: Send Send for LumaA<T>

impl<T> where T: Sync Sync for LumaA<T>

impl<T> where T: Unpin Unpin for LumaA<T>

impl<T> where T: UnwindSafe UnwindSafe for LumaA<T>

impl<T> where T: RefUnwindSafe RefUnwindSafe for LumaA<T>

impl<T: Primitive + 'static> Pixel for LumaA<T>

 type Subpixel = T;

 const CHANNEL_COUNT: u8 = 2;

 const COLOR_MODEL: &'static str = "YA";

 const COLOR_TYPE: ColorType = [ColorType::La8, ColorType::La16][(std::mem::size_of::<T>()
> 1) as usize];

 fn channels(self:  &Self) -> &[T]

 fn channels_mut(self:  &mut Self) -> &mut [T]

 fn channels4(self:  &Self) -> (T,T,T,T)

 fn from_channels(a:  T, b:  T, c:  T, d:  T) -> LumaA<T>

 fn from_slice(slice:  &[T]) -> &LumaA<T>

 fn from_slice_mut(slice:  &mut [T]) -> &mut LumaA<T>

 fn to_rgb(self:  &Self) -> Rgb<T>

 fn to_bgr(self:  &Self) -> Bgr<T>

 fn to_rgba(self:  &Self) -> Rgba<T>

 fn to_bgra(self:  &Self) -> Bgra<T>

 fn to_luma(self:  &Self) -> Luma<T>

 fn to_luma_alpha(self:  &Self) -> LumaA<T>

 fn map<F> where F: FnMut(self:  &Self, f:  F) -> LumaA<T>
 fn apply<F> where F: FnMut(self:  &mut Self, mut f:  F)

 fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &Self, f:  F, g:  G) -> LumaA<T>

 fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &mut Self, mut f:  F, mut g:
G)

 fn map2<F> where F: FnMut(self:  &Self, other:  &Self, f:  F) -> LumaA<T>

 fn apply2<F> where F: FnMut(self:  &mut Self, other:  &LumaA<T>, mut f:  F)

 fn invert(self:  &mut Self)

 fn blend(self:  &mut Self, other:  &LumaA<T>)

impl<T: Primitive + 'static> From<[T; 2]> for LumaA<T>

 fn from(c:  [T; 2]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for LumaA<T>

 fn clone(self:  &Self) -> LumaA<T>

impl<T: $crate::marker::Copy + Primitive> Copy for LumaA<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for LumaA<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<LumaA<T» for LumaA<T>

 fn eq(self:  &Self, other:  &LumaA<T>) -> bool

 fn ne(self:  &Self, other:  &LumaA<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for LumaA<T>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for LumaA<T>

 type Output = T;

 fn index(self:  &Self, _index:  usize) -> &T

impl<T: Primitive> IndexMut<usize> for LumaA<T>

 fn index_mut(self:  &mut Self, _index:  usize) -> &mut T

impl<T: $crate::hash::Hash + Primitive> Hash for LumaA<T>

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for LumaA<T>

impl<T: Primitive> StructuralEq for LumaA<T>
```

1.269 Struct pub image::Rgb

```
#[repr(C)]
#[allow(missing_docs)]
pub struct Rgb<T: Primitive>(
    pub [T; 3],
)
```

RGB colors

Fields

`0: [T; 3]`

Implementations

`impl<T> where T: Send Send for Rgb<T>`

`impl<T> where T: Sync Sync for Rgb<T>`

`impl<T> where T: Unpin Unpin for Rgb<T>`

`impl<T> where T: UnwindSafe UnwindSafe for Rgb<T>`

`impl<T> where T: RefUnwindSafe RefUnwindSafe for Rgb<T>`

`impl<T: Primitive + 'static> Pixel for Rgb<T>`

`type Subpixel = T;`

`const CHANNEL_COUNT: u8 = 3;`

`const COLOR_MODEL: &'static str = "RGB";`

`const COLOR_TYPE: ColorType = [ColorType::Rgb8, ColorType::Rgb16][(std::mem::size_of::<T>() > 1) as usize];`

`fn channels(self: &Self) -> &[T]`

`fn channels_mut(self: &mut Self) -> &mut [T]`

`fn channels4(self: &Self) -> (T,T,T,T)`

`fn from_channels(a: T, b: T, c: T, d: T) -> Rgb<T>`

`fn from_slice(slice: &[T]) -> &Rgb<T>`

`fn from_slice_mut(slice: &mut [T]) -> &mut Rgb<T>`

`fn to_rgb(self: &Self) -> Rgb<T>`

`fn to_bgr(self: &Self) -> Bgr<T>`

`fn to_rgba(self: &Self) -> Rgba<T>`

```
 fn to_bgra(self:  &Self) -> Bgra<T>

 fn to_luma(self:  &Self) -> Luma<T>

 fn to_luma_alpha(self:  &Self) -> LumaA<T>

 fn map<F> where F: FnMut(self:  &Self, f:  F) -> Rgb<T>

 fn apply<F> where F: FnMut(self:  &mut Self, mut f:  F)

 fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &Self, f:  F, g:  G) -> Rgb<T>

 fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &mut Self, mut f:  F, mut g:
G)

 fn map2<F> where F: FnMut(self:  &Self, other:  &Self, f:  F) -> Rgb<T>

 fn apply2<F> where F: FnMut(self:  &mut Self, other:  &Rgb<T>, mut f:  F)

 fn invert(self:  &mut Self)

 fn blend(self:  &mut Self, other:  &Rgb<T>)

impl<T: Primitive + ’static> From<[T; 3]> for Rgb<T>

 fn from(c:  [T; 3]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for Rgb<T>

 fn clone(self:  &Self) -> Rgb<T>

impl<T: $crate::marker::Copy + Primitive> Copy for Rgb<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for Rgb<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<Rgb<T» for Rgb<T>

 fn eq(self:  &Self, other:  &Rgb<T>) -> bool

 fn ne(self:  &Self, other:  &Rgb<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for Rgb<T>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<’_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for Rgb<T>

 type Output = T;

 fn index(self:  &Self, _index:  usize) -> &T

impl<T: Primitive> IndexMut<usize> for Rgb<T>

 fn index_mut(self:  &mut Self, _index:  usize) -> &mut T
```

impl<T: $crate::hash::Hash + Primitive> Hash for Rgb<T>

fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for Rgb<T>

impl<T: Primitive> StructuralEq for Rgb<T>

1.270 Struct pub image::Rgba

```
#[repr(C)]
#[allow(missing_docs)]
pub struct Rgba<T: Primitive>(
    pub [T; 4],
)
```

RGB colors + alpha channel

Fields

0: [T; 4]

Implementations

impl<T> where T: Send Send for Rgba<T>

impl<T> where T: Sync Sync for Rgba<T>

impl<T> where T: Unpin Unpin for Rgba<T>

impl<T> where T: UnwindSafe UnwindSafe for Rgba<T>

impl<T> where T: RefUnwindSafe RefUnwindSafe for Rgba<T>

impl<T: Primitive + 'static> Pixel for Rgba<T>

type Subpixel = T;

const CHANNEL_COUNT: u8 = 4;

const COLOR_MODEL: &'static str = "RGBA";

const COLOR_TYPE: ColorType = [ColorType::Rgba8, ColorType::Rgba16][(std::mem::size_of::<T>() > 1) as usize];

fn channels(self: &Self) -> &[T]

fn channels_mut(self: &mut Self) -> &mut [T]

fn channels4(self: &Self) -> (T,T,T,T)

fn from_channels(a: T, b: T, c: T, d: T) -> Rgba<T>

fn from_slice(slice: &[T]) -> &Rgba<T>

```
 fn from_slice_mut(slice:  &mut [T]) -> &mut Rgba<T>

 fn to_rgb(self:  &Self) -> Rgb<T>

 fn to_bgr(self:  &Self) -> Bgr<T>

 fn to_rgba(self:  &Self) -> Rgba<T>

 fn to_bgra(self:  &Self) -> Bgra<T>

 fn to_luma(self:  &Self) -> Luma<T>

 fn to_luma_alpha(self:  &Self) -> LumaA<T>

 fn map<F> where F: FnMut(self:  &Self, f:  F) -> Rgba<T>

 fn apply<F> where F: FnMut(self:  &mut Self, mut f:  F)

 fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &Self, f:  F, g:  G) -> Rgba<T>

 fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &mut Self, mut f:  F, mut g:
G)

 fn map2<F> where F: FnMut(self:  &Self, other:  &Self, f:  F) -> Rgba<T>

 fn apply2<F> where F: FnMut(self:  &mut Self, other:  &Rgba<T>, mut f:  F)

 fn invert(self:  &mut Self)

 fn blend(self:  &mut Self, other:  &Rgba<T>)

impl<T: Primitive + 'static> From<[T; 4]> for Rgba<T>

 fn from(c:  [T; 4]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for Rgba<T>

 fn clone(self:  &Self) -> Rgba<T>

impl<T: $crate::marker::Copy + Primitive> Copy for Rgba<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for Rgba<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<Rgba<T» for Rgba<T>

 fn eq(self:  &Self, other:  &Rgba<T>) -> bool

 fn ne(self:  &Self, other:  &Rgba<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for Rgba<T>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for Rgba<T>
```

type Output = T;

fn index(self: &Self, _index: usize) -> &T

impl<T: Primitive> IndexMut<usize> for Rgba<T>

fn index_mut(self: &mut Self, _index: usize) -> &mut T

impl<T: $crate::hash::Hash + Primitive> Hash for Rgba<T>

fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for Rgba<T>

impl<T: Primitive> StructuralEq for Rgba<T>

1.271 Struct pub image::Bgr

```
#[repr(C)]
#[allow(missing_docs)]
pub struct Bgr<T: Primitive>(
    pub [T; 3],
)
```

BGR colors

Fields

0: [T; 3]

Implementations

impl<T> where T: Send Send for Bgr<T>

impl<T> where T: Sync Sync for Bgr<T>

impl<T> where T: Unpin Unpin for Bgr<T>

impl<T> where T: UnwindSafe UnwindSafe for Bgr<T>

impl<T> where T: RefUnwindSafe RefUnwindSafe for Bgr<T>

impl<T: Primitive + 'static> Pixel for Bgr<T>

type Subpixel = T;

const CHANNEL_COUNT: u8 = 3;

const COLOR_MODEL: &'static str = ''BGR'';

const COLOR_TYPE: ColorType = [ColorType::Bgr8, ColorType::Bgr8][(std::mem::size_of::<T>() > 1) as usize];

fn channels(self: &Self) -> &[T]

```
 fn channels_mut(self:  &mut Self) -> &mut [T]

 fn channels4(self:  &Self) -> (T,T,T,T)

 fn from_channels(a:  T, b:  T, c:  T, d:  T) -> Bgr<T>

 fn from_slice(slice:  &[T]) -> &Bgr<T>

 fn from_slice_mut(slice:  &mut [T]) -> &mut Bgr<T>

 fn to_rgb(self:  &Self) -> Rgb<T>

 fn to_bgr(self:  &Self) -> Bgr<T>

 fn to_rgba(self:  &Self) -> Rgba<T>

 fn to_bgra(self:  &Self) -> Bgra<T>

 fn to_luma(self:  &Self) -> Luma<T>

 fn to_luma_alpha(self:  &Self) -> LumaA<T>

 fn map<F> where F: FnMut(self:  &Self, f:  F) -> Bgr<T>

 fn apply<F> where F: FnMut(self:  &mut Self, mut f:  F)

 fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &Self, f:  F, g:  G) -> Bgr<T>

 fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self:  &mut Self, mut f:  F, mut g:
G)

 fn map2<F> where F: FnMut(self:  &Self, other:  &Self, f:  F) -> Bgr<T>

 fn apply2<F> where F: FnMut(self:  &mut Self, other:  &Bgr<T>, mut f:  F)

 fn invert(self:  &mut Self)

 fn blend(self:  &mut Self, other:  &Bgr<T>)

impl<T: Primitive + 'static> From<[T; 3]> for Bgr<T>

 fn from(c:  [T; 3]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for Bgr<T>

 fn clone(self:  &Self) -> Bgr<T>

impl<T: $crate::marker::Copy + Primitive> Copy for Bgr<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for Bgr<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<Bgr<T» for Bgr<T>

 fn eq(self:  &Self, other:  &Bgr<T>) -> bool
```

fn ne(self: &Self, other: &Bgr<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for Bgr<T>

fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for Bgr<T>

type Output = T;

fn index(self: &Self, _index: usize) -> &T

impl<T: Primitive> IndexMut<usize> for Bgr<T>

fn index_mut(self: &mut Self, _index: usize) -> &mut T

impl<T: $crate::hash::Hash + Primitive> Hash for Bgr<T>

fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for Bgr<T>

impl<T: Primitive> StructuralEq for Bgr<T>

1.272 Struct pub image::Bgra

```
#[repr(C)]
#[allow(missing_docs)]
pub struct Bgra<T: Primitive>(
    pub [T; 4],
)
```

BGR colors + alpha channel

Fields

0: [T; 4]

Implementations

impl<T> where T: Send Send for Bgra<T>

impl<T> where T: Sync Sync for Bgra<T>

impl<T> where T: Unpin Unpin for Bgra<T>

impl<T> where T: UnwindSafe UnwindSafe for Bgra<T>

impl<T> where T: RefUnwindSafe RefUnwindSafe for Bgra<T>

impl<T: Primitive + 'static> Pixel for Bgra<T>

type Subpixel = T;

const CHANNEL_COUNT: u8 = 4;

const COLOR_MODEL: &'static str = "BGRA";

const COLOR_TYPE: ColorType = [ColorType::Bgra8, ColorType::Bgra8][(std::mem::size_of::<T>() > 1) as usize];

fn channels(self: &Self) -> &[T]

fn channels_mut(self: &mut Self) -> &mut [T]

fn channels4(self: &Self) -> (T,T,T,T)

fn from_channels(a: T, b: T, c: T, d: T) -> Bgra<T>

fn from_slice(slice: &[T]) -> &Bgra<T>

fn from_slice_mut(slice: &mut [T]) -> &mut Bgra<T>

fn to_rgb(self: &Self) -> Rgb<T>

fn to_bgr(self: &Self) -> Bgr<T>

fn to_rgba(self: &Self) -> Rgba<T>

fn to_bgra(self: &Self) -> Bgra<T>

fn to_luma(self: &Self) -> Luma<T>

fn to_luma_alpha(self: &Self) -> LumaA<T>

fn map<F> where F: FnMut(self: &Self, f: F) -> Bgra<T>

fn apply<F> where F: FnMut(self: &mut Self, mut f: F)

fn map_with_alpha<F, G> where F: FnMut, G: FnMut(self: &Self, f: F, g: G) -> Bgra<T>

fn apply_with_alpha<F, G> where F: FnMut, G: FnMut(self: &mut Self, mut f: F, mut g: G)

fn map2<F> where F: FnMut(self: &Self, other: &Self, f: F) -> Bgra<T>

fn apply2<F> where F: FnMut(self: &mut Self, other: &Bgra<T>, mut f: F)

fn invert(self: &mut Self)

fn blend(self: &mut Self, other: &Bgra<T>)

impl<T: Primitive + 'static> From<[T; 4]> for Bgra<T>

fn from(c: [T; 4]) -> Self

impl<T: $crate::clone::Clone + Primitive> Clone for Bgra<T>

fn clone(self: &Self) -> Bgra<T>

```
impl<T: $crate::marker::Copy + Primitive> Copy for Bgra<T>

impl<T: $crate::cmp::Eq + Primitive> Eq for Bgra<T>

impl<T: $crate::cmp::PartialEq + Primitive> PartialEq<Bgra<T» for Bgra<T>

 fn eq(self:  &Self, other:  &Bgra<T>) -> bool

 fn ne(self:  &Self, other:  &Bgra<T>) -> bool

impl<T: $crate::fmt::Debug + Primitive> Debug for Bgra<T>

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl<T: Primitive> Index<usize> for Bgra<T>

 type Output = T;

 fn index(self:  &Self, _index:  usize) -> &T

impl<T: Primitive> IndexMut<usize> for Bgra<T>

 fn index_mut(self:  &mut Self, _index:  usize) -> &mut T

impl<T: $crate::hash::Hash + Primitive> Hash for Bgra<T>

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl<T: Primitive> StructuralPartialEq for Bgra<T>

impl<T: Primitive> StructuralEq for Bgra<T>
```

1.273 Re-export pub ImageError

```rust
pub use ImageError;
```

1.274 Re-export pub ImageResult

```rust
pub use ImageResult;
```

1.275 Trait pub image::AnimationDecoder

```rust
pub trait AnimationDecoder<'a> {
}
```

AnimationDecoder trait

1.276 Trait pub image::GenericImage

```
pub trait GenericImage: GenericImageView {
    type InnerImage: GenericImage<Pixel=<Self as >::Pixel>;
}
```

A trait for manipulating images.

Associated items

`InnerImage` Underlying image type. This is mainly used by SubImages in order to always have a reference to the original image. This allows for less indirections and it eases the use of nested SubImages.

1.277 Trait pub image::GenericImageView

```
pub trait GenericImageView {
    type Pixel: Pixel;
    type InnerImageView: GenericImageView<Pixel=<Self as >::Pixel>;
}
```

Trait to inspect an image.

Associated items

`Pixel` The type of pixel.

`InnerImageView` Underlying image type. This is mainly used by SubImages in order to always have a reference to the original image. This allows for less indirections and it eases the use of nested SubImages.

1.278 Trait pub image::ImageDecoder

```
pub trait ImageDecoder<'a>: Sized {
    type Reader: Read + 'a;
}
```

The trait that all decoders implement

Associated items

`Reader` The type of reader produced by `into_reader`.

1.279 Trait pub image::ImageDecoderExt

```
pub trait ImageDecoderExt<'a>: ImageDecoder<'a> + Sized {
}
```

Specialized image decoding not be supported by all formats

1.280 Trait pub image::ImageEncoder

```
pub trait ImageEncoder {
}
```

The trait all encoders implement

1.281 Enum pub image::ImageFormat

```
pub enum image::ImageFormat {
    Png,
    Jpeg,
    Gif,
    WebP,
    Pnm,
    Tiff,
    Tga,
    Dds,
    Bmp,
    Ico,
    Hdr,
    Farbfeld,
    Avif,
    // some variants omitted
}
```

An enumeration of supported image formats. Not all formats support both encoding and decoding.

Variants

`Png` An Image in PNG Format

`Jpeg` An Image in JPEG Format

`Gif` An Image in GIF Format

`WebP` An Image in WEBP Format

`Pnm` An Image in general PNM Format

`Tiff` An Image in TIFF Format

`Tga` An Image in TGA Format

`Dds` An Image in DDS Format

`Bmp` An Image in BMP Format

`Ico` An Image in ICO Format

`Hdr` An Image in Radiance HDR Format

`Farbfeld` An Image in farbfeld Format

`Avif` An Image in AVIF format.

Implementations

```
impl ImageFormat
```

```
 pub fn from_extension<S> where S: AsRef<OsStr>(ext:  S) -> Option<Self>
```

Return the image format specified by a path's file extension.

# Example

‘‘‘ use image::ImageFormat;

let format = ImageFormat::from_extension("jpg"); assert_eq!(format, Some(ImageFormat::Jpeg)); ‘‘‘

```
 pub fn from_path<P> where P: AsRef<Path>(path:  P) -> ImageResult<Self>
```

Return the image format specified by the path's file extension.

# Example

‘‘‘ use image::ImageFormat;

let format = ImageFormat::from_path("images/ferris.png")?; assert_eq!(format, ImageFormat::Png);

# Ok::<(), image::error::ImageError>(()) ‘‘‘

```
 pub fn can_read(self:  &Self) -> bool
```

Return if the ImageFormat can be decoded by the lib.

```
 pub fn can_write(self:  &Self) -> bool
```

Return if the ImageFormat can be encoded by the lib.

```
 pub fn extensions_str(self:  Self) -> &'static [&'static str]
```

Return a list of applicable extensions for this format.

All currently recognized image formats specify at least on extension but for future compatibility you should not rely on this fact. The list may be empty if the format has no recognized file representation, for example in case it is used as a purely transient memory format.

The method name ‘extensions‘ remains reserved for introducing another method in the future that yields a slice of ‘OsStr‘ which is blocked by several features of const evaluation.

```
impl Send for ImageFormat
```

```
impl Sync for ImageFormat
```

```
impl Unpin for ImageFormat
```

```
impl UnwindSafe for ImageFormat
```

```
impl RefUnwindSafe for ImageFormat
```

```
impl From<ImageFormat> for ImageFormatHint
```

```
 fn from(format:  ImageFormat) -> Self
```

```
impl From<ImageFormat> for ImageOutputFormat
```

```
 fn from(fmt:  ImageFormat) -> Self
```

```
impl Clone for ImageFormat
```

```
 fn clone(self:  &Self) -> ImageFormat
```

```
impl Copy for ImageFormat
```

```
impl Eq for ImageFormat
```

```
impl PartialEq<ImageFormat> for ImageFormat
```

```
 fn eq(self:  &Self, other:  &ImageFormat) -> bool
```

```
 fn ne(self:  &Self, other:  &ImageFormat) -> bool
```

```
impl Debug for ImageFormat
```

```
 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl Hash for ImageFormat
```

```
 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()
```

```
impl StructuralPartialEq for ImageFormat
```

```
impl StructuralEq for ImageFormat
```

1.282 Enum pub image::ImageOutputFormat

```rust
pub enum image::ImageOutputFormat {
    Png,
    Jpeg,
    Pnm,
    Gif,
    Ico,
    Bmp,
    Farbfeld,
    Tga,
    Unsupported,
    // some variants omitted
}
```

An enumeration of supported image formats for encoding.

Variants

Png An Image in PNG Format

Jpeg An Image in JPEG Format with specified quality

Pnm An Image in one of the PNM Formats

`Gif` An Image in GIF Format

`Ico` An Image in ICO Format

`Bmp` An Image in BMP Format

`Farbfeld` An Image in farbfeld Format

`Tga` An Image in TGA Format

`Unsupported` A value for signalling an error: An unsupported format was requested

Implementations

`impl Send for ImageOutputFormat`

`impl Sync for ImageOutputFormat`

`impl Unpin for ImageOutputFormat`

`impl UnwindSafe for ImageOutputFormat`

`impl RefUnwindSafe for ImageOutputFormat`

`impl From<ImageFormat> for ImageOutputFormat`

`fn from(fmt: ImageFormat) -> Self`

`impl Clone for ImageOutputFormat`

`fn clone(self: &Self) -> ImageOutputFormat`

`impl Eq for ImageOutputFormat`

`impl PartialEq<ImageOutputFormat> for ImageOutputFormat`

`fn eq(self: &Self, other: &ImageOutputFormat) -> bool`

`fn ne(self: &Self, other: &ImageOutputFormat) -> bool`

`impl Debug for ImageOutputFormat`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

`impl StructuralPartialEq for ImageOutputFormat`

`impl StructuralEq for ImageOutputFormat`

1.283 Struct pub image::Progress

```
pub struct Progress{
    // some fields omitted
}
```

Represents the progress of an image operation.

Note that this is not necessarily accurate and no change to the values passed to the progress function during decoding will be considered breaking. A decoder could in theory report the progress `(0, 0)` if progress is unknown, without violating the interface contract of the type.

Implementations

```
impl Progress
```

```
 pub fn current(self:  Self) -> u64
```

A measure of completed decoding.

```
 pub fn total(self:  Self) -> u64
```

A measure of all necessary decoding work.

This is in general greater or equal than 'current'.

```
 pub fn remaining(self:  Self) -> u64
```

Calculate a measure for remaining decoding work.

```
impl Send for Progress

impl Sync for Progress

impl Unpin for Progress

impl UnwindSafe for Progress

impl RefUnwindSafe for Progress

impl Clone for Progress

 fn clone(self:  &Self) -> Progress

impl Copy for Progress

impl Eq for Progress

impl PartialEq<Progress> for Progress

 fn eq(self:  &Self, other:  &Progress) -> bool

 fn ne(self:  &Self, other:  &Progress) -> bool

impl Debug for Progress

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for Progress

impl StructuralEq for Progress
```

1.284 Struct pub image::Pixels

```
pub struct Pixels<'a, I: Sized + 'a>{
    // some fields omitted
}
```

Immutable pixel iterator

Implementations

`impl<'a, I: Sized> where I: Sync Send for Pixels<'a, I>`

`impl<'a, I: Sized> where I: Sync Sync for Pixels<'a, I>`

`impl<'a, I: Sized> Unpin for Pixels<'a, I>`

`impl<'a, I: Sized> where I: RefUnwindSafe UnwindSafe for Pixels<'a, I>`

`impl<'a, I: Sized> where I: RefUnwindSafe RefUnwindSafe for Pixels<'a, I>`

`impl<'a, I: GenericImageView> Iterator for Pixels<'a, I>`

`type Item = (u32,u32,<I as >::Pixel);`

`fn next(self: &mut Self) -> Option<(u32,u32,<I as >::Pixel)>`

`impl<I: Sized> Clone for Pixels<'_, I>`

`fn clone(self: &Self) -> Self`

`impl<'a, I: $crate::fmt::Debug + Sized + 'a> Debug for Pixels<'a, I>`

`fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result`

1.285 Struct pub image::SubImage

```
pub struct SubImage<I>{
    // some fields omitted
}
```

A View into another image

Instances of this struct can be created using:

`age::sub_image` to create a mutable view,
`mageView::view` to create an immutable view,
`SubImage::new` to instantiate the struct directly.

Implementations

`impl<I> SubImage<I>`

`pub fn new(image: I, x: u32, y: u32, width: u32, height: u32) -> SubImage<I>`

Construct a new subimage The coordinates set the position of the top left corner of the SubImage.

```
 pub fn change_bounds(self:  &mut Self, x:  u32, y:  u32, width:  u32, height:  u32)
```

Change the coordinates of this subimage.

```
 pub fn to_image where I: Deref, <I as >::Target:  GenericImage + 'static( self:  &Self
) -> ImageBuffer«<I as Deref>::Target as GenericImageView>::Pixel, Vec««I as Deref>::Target
as GenericImageView>::Pixel as Pixel>::Subpixel»
```

Convert this subimage to an ImageBuffer

```
impl<I> where I: Send Send for SubImage<I>

impl<I> where I: Sync Sync for SubImage<I>

impl<I> where I: Unpin Unpin for SubImage<I>

impl<I> where I: UnwindSafe UnwindSafe for SubImage<I>

impl<I> where I: RefUnwindSafe RefUnwindSafe for SubImage<I>

impl<I> where I: Deref, <I as >::Target:  GenericImageView + Sized GenericImageView for
SubImage<I>

 type Pixel = «I as Deref>::Target as GenericImageView>::Pixel;

 type InnerImageView = <I as >::Target;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn bounds(self:  &Self) -> (u32,u32,u32,u32)

 fn get_pixel(self:  &Self, x:  u32, y:  u32) -> <Self as >::Pixel

 fn view( self:  &Self, x:  u32, y:  u32, width:  u32, height:  u32 ) -> SubImage<&<Self
as >::InnerImageView>

 fn inner(self:  &Self) -> &<Self as >::InnerImageView

impl<I> where I: DerefMut, <I as >::Target:  GenericImage + Sized GenericImage for SubImage<I>

 type InnerImage = <I as >::Target;

 fn get_pixel_mut(self:  &mut Self, x:  u32, y:  u32) -> &mut <Self as >::Pixel

 fn put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  <Self as >::Pixel)

 fn blend_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  <Self as >::Pixel)
```

DEPRECATED: This method will be removed. Blend the pixel directly instead.

```
 fn sub_image( self:  &mut Self, x:  u32, y:  u32, width:  u32, height:  u32 ) -> SubImage<&mut
<Self as >::InnerImage>

 fn inner_mut(self:  &mut Self) -> &mut <Self as >::InnerImage
```

1.286 Typedef pub image::GrayAlphaImage

```
pub type GrayAlphaImage = ImageBuffer<LumaA<u8>, Vec<u8>>;
```

Sendable grayscale + alpha channel image buffer

1.287 Typedef pub image::GrayImage

```
pub type GrayImage = ImageBuffer<Luma<u8>, Vec<u8>>;
```

Sendable grayscale image buffer

1.288 Struct pub image::ImageBuffer

```
pub struct ImageBuffer<P: Pixel, Container>{
    // some fields omitted
}
```

Generic image buffer

This is an image parameterised by its Pixel types, represented by a width and height and a container of channel data. It provides direct access to its pixels and implements the `trait.GenericImageView.html` and `trait.GenericImage.html` traits. In many ways, this is the standard buffer implementing those traits. Using this concrete type instead of a generic type parameter has been shown to improve performance.

The crate defines a few type aliases with regularly used pixel types for your convenience, such as `RgbImage`, `GrayImage` etc.

To convert between images of different Pixel types use `enum.DynamicImage.html`.

You can retrieve a complete description of the buffer's layout and contents through **??** and **??**. This can be handy to also use the contents in a foreign language, map it as a GPU host buffer or other similar tasks.

Examples

Create a simple canvas and paint a small cross.

```
use image::{RgbImage, Rgb};

let mut img = RgbImage::new(32, 32);

for x in 15..=17 {
    for y in 8..24 {
        img.put_pixel(x, y, Rgb([255, 0, 0]));
        img.put_pixel(y, x, Rgb([255, 0, 0]));
    }
}
```

Overlays an image on top of a larger background raster.

```
use image::{GenericImage, GenericImageView, ImageBuffer, open};

let on_top = open("path/to/some.png").unwrap().into_rgb();
let mut img = ImageBuffer::from_fn(512, 512, |x, y| {
    if (x + y) % 2 == 0 {
        image::Rgb([0, 0, 0])
    } else {
        image::Rgb([255, 255, 255])
    }
});

image::imageops::overlay(&mut img, &on_top, 128, 128);
```

Convert an RgbaImage to a GrayImage.

```
use image::{open, DynamicImage};

let rgba = open("path/to/some.png").unwrap().into_rgba();
let gray = DynamicImage::ImageRgba8(rgba).into_luma();
```

Implementations

`impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel: 'static, Container: Deref<Targ`
`as >::Subpixel]> ImageBuffer<P, Container>`

`pub fn from_raw(width: u32, height: u32, buf: Container) -> Option<ImageBuffer<P, Container>`

Contructs a buffer from a generic container (for example a 'Vec' or a slice)

Returns 'None' if the container is not big enough (including when the image dimensions necessitate an allocation of more bytes than supported by the container).

`pub fn into_raw(self: Self) -> Container`

Returns the underlying raw buffer

`pub fn as_raw(self: &Self) -> &Container`

Returns the underlying raw buffer

`pub fn dimensions(self: &Self) -> (u32,u32)`

The width and height of this image.

`pub fn width(self: &Self) -> u32`

The width of this image.

`pub fn height(self: &Self) -> u32`

The height of this image.

`pub fn pixels(self: &Self) -> Pixels<'_, P>`

Returns an iterator over the pixels of this image. The iteration order is x = 0 to width then y = 0 to height

`pub fn rows(self: &Self) -> Rows<'_, P>`

Returns an iterator over the rows of this image.

Only non-empty rows can be iterated in this manner. In particular the iterator will not yield any item when the width of the image is ‘0‘ or a pixel type without any channels is used. This ensures that its length can always be represented by ‘usize‘.

```
 pub fn enumerate_pixels(self:  &Self) -> EnumeratePixels<'_, P>
```

Enumerates over the pixels of the image. The iterator yields the coordinates of each pixel along with a reference to them. The iteration order is x = 0 to width then y = 0 to height Starting from the top left.

```
 pub fn enumerate_rows(self:  &Self) -> EnumerateRows<'_, P>
```

Enumerates over the rows of the image. The iterator yields the y-coordinate of each row along with a reference to them.

```
 pub fn get_pixel(self:  &Self, x:  u32, y:  u32) -> &P
```

Gets a reference to the pixel at location ‘(x, y)‘

# Panics

Panics if ‘(x, y)‘ is out of the bounds ‘(width, height)‘.

```
 pub fn sample_layout(self:  &Self) -> SampleLayout
```

Get the format of the buffer when viewed as a matrix of samples.

```
 pub fn into_flat_samples where Container:  AsRef<[<P as >::Subpixel]>(self:  Self) ->
FlatSamples<Container>
```

Return the raw sample buffer with its stride an dimension information.

The returned buffer is guaranteed to be well formed in all cases. It is layed out by colors, width then height, meaning ‘channel_stride <= width_stride <= height_stride‘. All strides are in numbers of elements but those are mostly ‘u8‘ in which case the strides are also byte strides.

```
 pub fn as_flat_samples where Container:  AsRef<[<P as >::Subpixel]>(self:  &Self) -> FlatSample
as >::Subpixel]>
```

Return a view on the raw sample buffer.

See [‘into_flat_samples‘](#method.into_flat_samples) for more details.

```
 pub fn as_flat_samples_mut where Container:  AsMut<[<P as >::Subpixel]>(self:  &mut Self)
-> FlatSamples<&mut [<P as >::Subpixel]>
```

Return a mutable view on the raw sample buffer.

See [‘into_flat_samples‘](#method.into_flat_samples) for more details.

```
impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel:  'static, Container:  Deref<Targ
as >::Subpixel]> + DerefMut ImageBuffer<P, Container>
```

```
 pub fn pixels_mut(self:  &mut Self) -> PixelsMut<'_, P>
```

Returns an iterator over the mutable pixels of this image.

```
 pub fn rows_mut(self:  &mut Self) -> RowsMut<'_, P>
```

Returns an iterator over the mutable rows of this image.

Only non-empty rows can be iterated in this manner. In particular the iterator will not yield any item when the width of the image is '0' or a pixel type without any channels is used. This ensures that its length can always be represented by 'usize'.

```
 pub fn enumerate_pixels_mut(self:  &mut Self) -> EnumeratePixelsMut<'_, P>
```

Enumerates over the pixels of the image. The iterator yields the coordinates of each pixel along with a mutable reference to them.

```
 pub fn enumerate_rows_mut(self:  &mut Self) -> EnumerateRowsMut<'_, P>
```

Enumerates over the rows of the image. The iterator yields the y-coordinate of each row along with a mutable reference to them.

```
 pub fn get_pixel_mut(self:  &mut Self, x:  u32, y:  u32) -> &mut P
```

Gets a reference to the mutable pixel at location '(x, y)'

# Panics

Panics if '(x, y)' is out of the bounds '(width, height)'.

```
 pub fn put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  P)
```

Puts a pixel at location '(x, y)'

# Panics

Panics if '(x, y)' is out of the bounds '(width, height)'.

```
impl<P, Container> where P: Pixel + 'static, [<P as >::Subpixel]:  EncodableLayout, Container:
Deref<Target=[<P as >::Subpixel]> ImageBuffer<P, Container>
```

```
 pub fn save<Q> where Q: AsRef<Path>(self:  &Self, path:  Q) -> ImageResult<()>
```

Saves the buffer to a file at the path specified.

The image format is derived from the file extension. Currently only jpeg and png files are supported.

```
impl<P, Container> where P: Pixel + 'static, [<P as >::Subpixel]:  EncodableLayout, Container:
Deref<Target=[<P as >::Subpixel]> ImageBuffer<P, Container>
```

```
 pub fn save_with_format<Q> where Q: AsRef<Path>(self:  &Self, path:  Q, format:  ImageFormat)
-> ImageResult<()>
```

Saves the buffer to a file at the specified path in the specified format.

See ['save_buffer_with_format'](fn.save_buffer_with_format.html) for supported types.

```
impl<P: Pixel + 'static> where <P as >::Subpixel:  'static ImageBuffer<P, Vec«P as >::Subpixel»
```

```
 pub fn new(width:  u32, height:  u32) -> ImageBuffer<P, Vec«P as >::Subpixel»
```

Creates a new image buffer based on a 'Vec<P::Subpixel>'.

# Panics

Panics when the resulting image is larger the the maximum size of a vector.

```
 pub fn from_pixel(width:  u32, height:  u32, pixel:  P) -> ImageBuffer<P, Vec«P as >::Subpixel»
```

Constructs a new ImageBuffer by copying a pixel

# Panics

Panics when the resulting image is larger the the maximum size of a vector.

```
 pub fn from_fn<F> where F: FnMut(width:  u32, height:  u32, mut f:  F) -> ImageBuffer<P,
Vec«P as >::Subpixel»
```

Constructs a new ImageBuffer by repeated application of the supplied function.

The arguments to the function are the pixel's x and y coordinates.

# Panics

Panics when the resulting image is larger the the maximum size of a vector.

```
 pub fn from_vec( width:  u32, height:  u32, buf:  Vec«P as >::Subpixel> ) -> Option<ImageBuffer
Vec«P as >::Subpixel»>
```

Creates an image buffer out of an existing buffer. Returns None if the buffer is not big enough.

```
 pub fn into_vec(self:  Self) -> Vec«P as >::Subpixel>
```

Consumes the image buffer and returns the underlying data as an owned buffer

```
impl ImageBuffer<Luma<u8>, Vec<u8, Global»
```

```
 pub fn expand_palette(self:  Self, palette:  &[(u8,u8,u8)], transparent_idx:  Option<u8>)
-> RgbaImage
```

Expands a color palette by re-using the existing buffer. Assumes 8 bit per pixel. Uses an optionally transparent index to adjust it's alpha value accordingly.

```
impl<P, Container> where Container:  Send, P: Send Send for ImageBuffer<P, Container>
```

```
impl<P, Container> where Container:  Sync, P: Sync Sync for ImageBuffer<P, Container>
```

```
impl<P, Container> where Container:  Unpin, P: Unpin Unpin for ImageBuffer<P, Container>
```

```
impl<P, Container> where Container:  UnwindSafe, P: UnwindSafe UnwindSafe for ImageBuffer<P,
Container>
```

```
impl<P, Container> where Container:  RefUnwindSafe, P: RefUnwindSafe RefUnwindSafe for
ImageBuffer<P, Container>
```

```
impl<'a, 'b, Container, FromType:  Pixel + 'static, ToType:  Pixel + 'static> where Container:
Deref<Target=[<FromType as >::Subpixel]>, ToType:  FromColor<FromType>, <FromType as >::Subpixel
'static, <ToType as >::Subpixel:  'static ConvertBuffer<ImageBuffer<ToType, Vec«ToType
as Pixel>::Subpixel, Global»> for ImageBuffer<FromType, Container>
```

```
 fn convert(self:  &Self) -> ImageBuffer<ToType, Vec«ToType as >::Subpixel»
```

# Examples Convert RGB image to gray image. “‘no_run use image::buffer::ConvertBuffer; use image::GrayImage;

let image_path = “examples/fractal.png”; let image = image::open(&image_path) .expect(“Open file failed”) .to_rgba();

let gray_image: GrayImage = image.convert(); “‘

```
impl<P, Container> where P: Pixel + 'static, Container:  Deref<Target=[<P as >::Subpixel]>
+ Deref, <P as >::Subpixel:  'static GenericImageView for ImageBuffer<P, Container>
```

```
 type Pixel = P;
```

```
 type InnerImageView = Self;
```

```
 fn dimensions(self:  &Self) -> (u32,u32)
```

```
 fn bounds(self:  &Self) -> (u32,u32,u32,u32)
```

```
 fn get_pixel(self:  &Self, x:  u32, y:  u32) -> P
```

```
 unsafe fn unsafe_get_pixel(self:  &Self, x:  u32, y:  u32) -> P
```

Returns the pixel located at (x, y), ignoring bounds checking.

```
 fn inner(self:  &Self) -> &<Self as >::InnerImageView
```

```
impl<P, Container> where P: Pixel + 'static, Container:  Deref<Target=[<P as >::Subpixel]>
+ DerefMut, <P as >::Subpixel:  'static GenericImage for ImageBuffer<P, Container>
```

```
 type InnerImage = Self;
```

```
 fn get_pixel_mut(self:  &mut Self, x:  u32, y:  u32) -> &mut P
```

```
 fn put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  P)
```

```
 unsafe fn unsafe_put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  P)
```

Puts a pixel at location (x, y), ignoring bounds checking.

```
 fn blend_pixel(self:  &mut Self, x:  u32, y:  u32, p:  P)
```

Put a pixel at location (x, y), taking into account alpha channels

DEPRECATED: This method will be removed. Blend the pixel directly instead.

```
fn copy_within(self: &mut Self, source: Rect, x: u32, y: u32) -> bool
```

```
fn inner_mut(self: &mut Self) -> &mut <Self as >::InnerImage
```

```
impl<P, Container> where P: Pixel, Container: Deref<Target=[<P as >::Subpixel]> + Clone
Clone for ImageBuffer<P, Container>
```

```
fn clone(self: &Self) -> ImageBuffer<P, Container>
```

```
impl<P, Container> where P: Pixel, Container: Default Default for ImageBuffer<P, Container>
```

```
fn default() -> Self
```

```
impl<P: $crate::cmp::Eq + Pixel, Container: $crate::cmp::Eq> Eq for ImageBuffer<P, Container>
```

```
impl<P: $crate::cmp::PartialEq + Pixel, Container: $crate::cmp::PartialEq> PartialEq<ImageBuffe
Container» for ImageBuffer<P, Container>
```

```
fn eq(self: &Self, other: &ImageBuffer<P, Container>) -> bool
```

```
fn ne(self: &Self, other: &ImageBuffer<P, Container>) -> bool
```

```
impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel: 'static, Container: Deref<Targ
as >::Subpixel]> Deref for ImageBuffer<P, Container>
```

```
type Target = [<P as >::Subpixel];
```

```
fn deref(self: &Self) -> &<Self as Deref>::Target
```

```
impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel: 'static, Container: Deref<Targ
as >::Subpixel]> + DerefMut DerefMut for ImageBuffer<P, Container>
```

```
fn deref_mut(self: &mut Self) -> &mut <Self as Deref>::Target
```

```
impl<P: $crate::fmt::Debug + Pixel, Container: $crate::fmt::Debug> Debug for ImageBuffer<P,
Container>
```

```
fn fmt(self: &Self, f: &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result
```

```
impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel: 'static, Container: Deref<Targ
as >::Subpixel]> Index<(u32,u32)> for ImageBuffer<P, Container>
```

```
type Output = P;
```

```
fn index(self: &Self, (x, y): (u32,u32)) -> &P
```

```
impl<P, Container> where P: Pixel + 'static, <P as >::Subpixel: 'static, Container: Deref<Targ
as >::Subpixel]> + DerefMut IndexMut<(u32,u32)> for ImageBuffer<P, Container>
```

```
fn index_mut(self: &mut Self, (x, y): (u32,u32)) -> &mut P
```

```
impl<P: $crate::hash::Hash + Pixel, Container: $crate::hash::Hash> Hash for ImageBuffer<P,
Container>
```

```
 fn hash<__H: $crate::hash::Hasher>(self: &Self, state: &mut __H) -> ()
```

```
impl<P: Pixel, Container> StructuralPartialEq for ImageBuffer<P, Container>
```

```
impl<P: Pixel, Container> StructuralEq for ImageBuffer<P, Container>
```

1.289 Typedef pub image::RgbImage

```
pub type RgbImage = ImageBuffer<Rgb<u8>, Vec<u8>>;
```

Sendable Rgb image buffer

1.290 Typedef pub image::RgbaImage

```
pub type RgbaImage = ImageBuffer<Rgba<u8>, Vec<u8>>;
```

Sendable Rgb + alpha channel image buffer

1.291 Re-export pub FlatSamples

```
pub use FlatSamples;
```

1.292 Trait pub image::EncodableLayout

```
pub trait EncodableLayout: seals::EncodableLayout {
}
```

Types which are safe to treat as an immutable byte slice in a pixel layout for image encoding.

1.293 Trait pub image::Primitive

```
pub trait Primitive
    : CopyNumCast
    + Num
    + PartialOrd<Self>
    + Clone
    + Bounded {
}
```

Primitive trait from old stdlib

1.294 Trait pub image::Pixel

```
pub trait Pixel: Copy + Clone {
    type Subpixel: Primitive;
    CHANNEL_COUNT: u8;
    COLOR_MODEL: &'static str;
    COLOR_TYPE: ColorType;
}
```

A generalized pixel.

A pixel object is usually not used standalone but as a view into an image buffer.

Associated items

`Subpixel` The underlying subpixel type.

`CHANNEL_COUNT` The number of channels of this pixel type.

`COLOR_MODEL` A string that can help to interpret the meaning each channel See gimp babl.

`COLOR_TYPE` ColorType for this pixel format

1.295 Function pub image::guess_format

```
pub fn guess_format(buffer: &[u8]) -> ImageResult<ImageFormat>
```

Guess image format from memory block

Makes an educated guess about the image format based on the Magic Bytes at the beginning. TGA is not supported by this function. This is not to be trusted on the validity of the whole memory block

1.296 Function pub image::load

```
pub fn load(r: R, format: ImageFormat) -> ImageResult<DynamicImage>
```

Create a new image from a Reader

Try `io/struct.Reader.html` for more advanced uses.

1.297 Function pub image::load_from_memory

```
pub fn load_from_memory(buffer: &[u8]) -> ImageResult<DynamicImage>
```

Create a new image from a byte slice

Makes an educated guess about the image format. TGA is not supported by this function.

Try `io/struct.Reader.html` for more advanced uses.

1.298 Function pub image::load_from_memory_with_format

```
pub fn load_from_memory_with_format(buf: &[u8], format: ImageFormat) -> ImageResult<DynamicImage>
```

Create a new image from a byte slice

This is just a simple wrapper that constructs an `std::io::Cursor` around the buffer and then calls `load` with that reader.

Try `io/struct.Reader.html` for more advanced uses.

1.299 Function pub image::open

```
pub fn open(path: P) -> ImageResult<DynamicImage>
```

Open the image located at the path specified. The image's format is determined from the path's file extension.

Try `io/struct.Reader.html` for more advanced uses, including guessing the format based on the file's content before its path.

1.300 Function pub image::save_buffer

```
pub fn save_buffer(
    path: P, buf: &[u8], width: u32, height: u32, color: color::ColorType
)
  -> ImageResult<()>
```

Saves the supplied buffer to a file at the path specified.

The image format is derived from the file extension. The buffer is assumed to have the correct format according to the specified color type. This will lead to corrupted files if the buffer contains malformed data. Currently only jpeg, png, ico, pnm, bmp and tiff files are supported.

1.301 Function pub image::save_buffer_with_format

```
pub fn save_buffer_with_format(
    path: P, buf: &[u8], width: u32, height: u32, color: color::ColorType, format: ImageFormat
)
  -> ImageResult<()>
```

Saves the supplied buffer to a file at the path specified in the specified format.

The buffer is assumed to have the correct format according to the specified color type. This will lead to corrupted files if the buffer contains malformed data. Currently only jpeg, png, ico, bmp and tiff files are supported.

1.302 Function pub image::image_dimensions

```
pub fn image_dimensions(path: P) -> ImageResult<(u32,u32)>
```

Read the dimensions of the image located at the specified path. This is faster than fully loading the image and then getting its dimensions.

Try `io/struct.Reader.html` for more advanced uses, including guessing the format based on the file's content before its path or manually supplying the format.

1.303 Enum pub image::DynamicImage

```
pub enum image::DynamicImage {
    ImageLuma8,
    ImageLumaA8,
    ImageRgb8,
    ImageRgba8,
    ImageBgr8,
    ImageBgra8,
    ImageLuma16,
    ImageLumaA16,
    ImageRgb16,
    ImageRgba16,
}
```

A Dynamic Image

Variants

`ImageLuma8` Each pixel in this image is 8-bit Luma

`ImageLumaA8` Each pixel in this image is 8-bit Luma with alpha

`ImageRgb8` Each pixel in this image is 8-bit Rgb

`ImageRgba8` Each pixel in this image is 8-bit Rgb with alpha

`ImageBgr8` Each pixel in this image is 8-bit Bgr

`ImageBgra8` Each pixel in this image is 8-bit Bgr with alpha

`ImageLuma16` Each pixel in this image is 16-bit Luma

`ImageLumaA16` Each pixel in this image is 16-bit Luma with alpha

`ImageRgb16` Each pixel in this image is 16-bit Rgb

`ImageRgba16` Each pixel in this image is 16-bit Rgb with alpha

Implementations

`impl DynamicImage`

` pub fn new_luma8(w:  u32, h:  u32) -> DynamicImage`

Creates a dynamic image backed by a buffer of grey pixels.

```
pub fn new_luma_a8(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of grey pixels with transparency.

```
pub fn new_rgb8(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of RGB pixels.

```
pub fn new_rgba8(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of RGBA pixels.

```
pub fn new_bgra8(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of BGRA pixels.

```
pub fn new_bgr8(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of BGR pixels.

```
pub fn new_luma16(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of grey pixels.

```
pub fn new_luma_a16(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of grey pixels with transparency.

```
pub fn new_rgb16(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of RGB pixels.

```
pub fn new_rgba16(w: u32, h: u32) -> DynamicImage
```

Creates a dynamic image backed by a buffer of RGBA pixels.

```
pub fn from_decoder<'a, impl ImageDecoder<'a>: ImageDecoder<'a»(decoder: impl ImageDecoder<'a
-> ImageResult<Self>
```

Decodes an encoded image into a dynamic image.

```
pub fn to_rgb(self: &Self) -> RgbImage
```

Returns a copy of this image as an RGB image.

```
pub fn to_rgb8(self: &Self) -> RgbImage
```

Returns a copy of this image as an RGB image.

```
pub fn to_rgb16(self: &Self) -> ImageBuffer<Rgb<u16>, Vec<u16»
```

Returns a copy of this image as an RGB image.

```
pub fn to_rgba(self: &Self) -> RgbaImage
```

Returns a copy of this image as an RGBA image.

```
pub fn to_rgba8(self:  &Self) -> RgbaImage
```

Returns a copy of this image as an RGBA image.

```
pub fn to_rgba16(self:  &Self) -> ImageBuffer<Rgba<u16>, Vec<u16»
```

Returns a copy of this image as an RGBA image.

```
pub fn to_bgr(self:  &Self) -> ImageBuffer<Bgr<u8>, Vec<u8»
```

Returns a copy of this image as an BGR image.

```
pub fn to_bgr8(self:  &Self) -> ImageBuffer<Bgr<u8>, Vec<u8»
```

Returns a copy of this image as an BGR image.

```
pub fn to_bgra(self:  &Self) -> ImageBuffer<Bgra<u8>, Vec<u8»
```

Returns a copy of this image as an BGRA image.

```
pub fn to_bgra8(self:  &Self) -> ImageBuffer<Bgra<u8>, Vec<u8»
```

Returns a copy of this image as an BGRA image.

```
pub fn to_luma(self:  &Self) -> GrayImage
```

Returns a copy of this image as a Luma image.

```
pub fn to_luma8(self:  &Self) -> GrayImage
```

Returns a copy of this image as a Luma image.

```
pub fn to_luma16(self:  &Self) -> ImageBuffer<Luma<u16>, Vec<u16»
```

Returns a copy of this image as a Luma image.

```
pub fn to_luma_alpha(self:  &Self) -> GrayAlphaImage
```

Returns a copy of this image as a LumaA image.

```
pub fn to_luma_alpha8(self:  &Self) -> GrayAlphaImage
```

Returns a copy of this image as a LumaA image.

```
pub fn to_luma_alpha16(self:  &Self) -> ImageBuffer<LumaA<u16>, Vec<u16»
```

Returns a copy of this image as a LumaA image.

```
pub fn into_rgb(self:  Self) -> RgbImage
```

Consume the image and returns a RGB image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_rgb8(self: Self) -> RgbImage
```

Consume the image and returns a RGB image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_rgb16(self: Self) -> ImageBuffer<Rgb<u16>, Vec<u16»
```

Consume the image and returns a RGB image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_rgba(self: Self) -> RgbaImage
```

Consume the image and returns a RGBA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_rgba8(self: Self) -> RgbaImage
```

Consume the image and returns a RGBA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_rgba16(self: Self) -> ImageBuffer<Rgba<u16>, Vec<u16»
```

Consume the image and returns a RGBA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_bgr(self: Self) -> ImageBuffer<Bgr<u8>, Vec<u8»
```

Consume the image and returns a BGR image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_bgr8(self: Self) -> ImageBuffer<Bgr<u8>, Vec<u8»
```

Consume the image and returns a BGR image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_bgra(self: Self) -> ImageBuffer<Bgra<u8>, Vec<u8»
```

Consume the image and returns a BGRA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_bgra8(self: Self) -> ImageBuffer<Bgra<u8>, Vec<u8»
```

Consume the image and returns a BGRA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma(self: Self) -> GrayImage
```

Consume the image and returns a Luma image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma8(self: Self) -> GrayImage
```

Consume the image and returns a Luma image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma16(self: Self) -> ImageBuffer<Luma<u16>, Vec<u16»
```

Consume the image and returns a Luma image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma_alpha(self: Self) -> GrayAlphaImage
```

Consume the image and returns a LumaA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma_alpha8(self: Self) -> GrayAlphaImage
```

Consume the image and returns a LumaA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn into_luma_alpha16(self: Self) -> ImageBuffer<LumaA<u16>, Vec<u16»
```

Consume the image and returns a LumaA image.

If the image was already the correct format, it is returned as is. Otherwise, a copy is created.

```
pub fn crop(self: &mut Self, x: u32, y: u32, width: u32, height: u32) -> DynamicImage
```

Return a cut-out of this image delimited by the bounding rectangle.

Note: this method does *not* modify the object, and its signature will be replaced with 'crop_imm()''s
in the 0.24 release

```
pub fn crop_imm(self: &Self, x: u32, y: u32, width: u32, height: u32) -> DynamicImage
```

Return a cut-out of this image delimited by the bounding rectangle.

```
pub fn as_rgb8(self: &Self) -> Option<&RgbImage>
```

Return a reference to an 8bit RGB image

```
pub fn as_mut_rgb8(self: &mut Self) -> Option<&mut RgbImage>
```

Return a mutable reference to an 8bit RGB image

```
pub fn as_bgr8(self: &Self) -> Option<&ImageBuffer<Bgr<u8>, Vec<u8»>
```

Return a reference to an 8bit BGR image

```
pub fn as_mut_bgr8(self: &mut Self) -> Option<&mut ImageBuffer<Bgr<u8>, Vec<u8»>
```

Return a mutable reference to an 8bit BGR image

```
pub fn as_rgba8(self: &Self) -> Option<&RgbaImage>
```

Return a reference to an 8bit RGBA image

```
pub fn as_mut_rgba8(self: &mut Self) -> Option<&mut RgbaImage>
```

Return a mutable reference to an 8bit RGBA image

```
pub fn as_bgra8(self: &Self) -> Option<&ImageBuffer<Bgra<u8>, Vec<u8»>
```

Return a reference to an 8bit BGRA image

```
pub fn as_mut_bgra8(self: &mut Self) -> Option<&mut ImageBuffer<Bgra<u8>, Vec<u8»>
```

Return a mutable reference to an 8bit RGBA image

```
pub fn as_luma8(self: &Self) -> Option<&GrayImage>
```

Return a reference to an 8bit Grayscale image

```
pub fn as_mut_luma8(self: &mut Self) -> Option<&mut GrayImage>
```

Return a mutable reference to an 8bit Grayscale image

```
pub fn as_luma_alpha8(self: &Self) -> Option<&GrayAlphaImage>
```

Return a reference to an 8bit Grayscale image with an alpha channel

```
pub fn as_mut_luma_alpha8(self: &mut Self) -> Option<&mut GrayAlphaImage>
```

Return a mutable reference to an 8bit Grayscale image with an alpha channel

```
pub fn as_rgb16(self: &Self) -> Option<&ImageBuffer<Rgb<u16>, Vec<u16»>
```

Return a reference to an 16bit RGB image

```
pub fn as_mut_rgb16(self: &mut Self) -> Option<&mut ImageBuffer<Rgb<u16>, Vec<u16»>
```

Return a mutable reference to an 16bit RGB image

```
pub fn as_rgba16(self: &Self) -> Option<&ImageBuffer<Rgba<u16>, Vec<u16»>
```

Return a reference to an 16bit RGBA image

```
pub fn as_mut_rgba16(self: &mut Self) -> Option<&mut ImageBuffer<Rgba<u16>, Vec<u16»>
```

Return a mutable reference to an 16bit RGBA image

```
pub fn as_luma16(self: &Self) -> Option<&ImageBuffer<Luma<u16>, Vec<u16»>
```

Return a reference to an 16bit Grayscale image

```
pub fn as_mut_luma16(self: &mut Self) -> Option<&mut ImageBuffer<Luma<u16>, Vec<u16»>
```

Return a mutable reference to an 16bit Grayscale image

```
pub fn as_luma_alpha16(self: &Self) -> Option<&ImageBuffer<LumaA<u16>, Vec<u16»>
```

Return a reference to an 16bit Grayscale image with an alpha channel

```
pub fn as_mut_luma_alpha16(self: &mut Self) -> Option<&mut ImageBuffer<LumaA<u16>, Vec<u16»>
```

Return a mutable reference to an 16bit Grayscale image with an alpha channel

```
pub fn as_flat_samples_u8(self: &Self) -> Option<FlatSamples<&[u8]»
```

Return a view on the raw sample buffer for 8 bit per channel images.

```
pub fn as_flat_samples_u16(self: &Self) -> Option<FlatSamples<&[u16]»
```

Return a view on the raw sample buffer for 16 bit per channel images.

```
pub fn as_bytes(self: &Self) -> &[u8]
```

Return this image's pixels as a native endian byte slice.

```
pub fn into_bytes(self: Self) -> Vec<u8>
```

Return this image's pixels as a byte vector. If the 'ImageBuffer' container is 'Vec<u8>', this operation is free. Otherwise, a copy is returned.

```
pub fn to_bytes(self: &Self) -> Vec<u8>
```

Return a copy of this image's pixels as a byte vector.

```
pub fn color(self: &Self) -> color::ColorType
```

Return this image's color type.

```
pub fn grayscale(self: &Self) -> DynamicImage
```

Return a grayscale version of this image.

```
pub fn invert(self: &mut Self)
```

Invert the colors of this image. This method operates inplace.

```
pub fn resize(self: &Self, nwidth: u32, nheight: u32, filter: imageops::FilterType)
-> DynamicImage
```

Resize this image using the specified filter algorithm. Returns a new image. The image's aspect ratio is preserved. The image is scaled to the maximum possible size that fits within the bounds specified by "'nwidth"' and "'nheight"'.

```
 pub fn resize_exact(self: &Self, nwidth: u32, nheight: u32, filter: imageops::FilterType)
-> DynamicImage
```

Resize this image using the specified filter algorithm. Returns a new image. Does not preserve aspect ratio. "'nwidth"' and "'nheight"' are the new image's dimensions

```
 pub fn thumbnail(self: &Self, nwidth: u32, nheight: u32) -> DynamicImage
```

Scale this image down to fit within a specific size. Returns a new image. The image's aspect ratio is preserved. The image is scaled to the maximum possible size that fits within the bounds specified by "'nwidth"' and "'nheight"'.

This method uses a fast integer algorithm where each source pixel contributes to exactly one target pixel. May give aliasing artifacts if new size is close to old size.

```
 pub fn thumbnail_exact(self: &Self, nwidth: u32, nheight: u32) -> DynamicImage
```

Scale this image down to a specific size. Returns a new image. Does not preserve aspect ratio. "'nwidth"' and "'nheight"' are the new image's dimensions. This method uses a fast integer algorithm where each source pixel contributes to exactly one target pixel. May give aliasing artifacts if new size is close to old size.

```
 pub fn resize_to_fill(self: &Self, nwidth: u32, nheight: u32, filter: imageops::FilterType)
-> DynamicImage
```

Resize this image using the specified filter algorithm. Returns a new image. The image's aspect ratio is preserved. The image is scaled to the maximum possible size that fits within the larger (relative to aspect ratio) of the bounds specified by "'nwidth"' and "'nheight"', then cropped to fit within the other bound.

```
 pub fn blur(self: &Self, sigma: f32) -> DynamicImage
```

Performs a Gaussian blur on this image. "'sigma"' is a measure of how much to blur by.

```
 pub fn unsharpen(self: &Self, sigma: f32, threshold: i32) -> DynamicImage
```

Performs an unsharpen mask on this image. "'sigma"' is the amount to blur the image by. "'threshold"' is a control of how much to sharpen.

See <https://en.wikipedia.org/wiki/Unsharp_masking#Digital_unsharp_masking>

```
 pub fn filter3x3(self: &Self, kernel: &[f32]) -> DynamicImage
```

Filters this image with the specified 3x3 kernel.

```
 pub fn adjust_contrast(self: &Self, c: f32) -> DynamicImage
```

Adjust the contrast of this image. "'contrast"' is the amount to adjust the contrast by. Negative values decrease the contrast and positive values increase the contrast.

```
 pub fn brighten(self: &Self, value: i32) -> DynamicImage
```

Brighten the pixels of this image. "'value"' is the amount to brighten each pixel by. Negative values decrease the brightness and positive values increase it.

```
pub fn huerotate(self: &Self, value: i32) -> DynamicImage
```

Hue rotate the supplied image. ‘value‘ is the degrees to rotate each pixel by. 0 and 360 do nothing, the rest rotates by the given degree value. just like the css webkit filter hue-rotate(180)

```
pub fn flipv(self: &Self) -> DynamicImage
```

Flip this image vertically

```
pub fn fliph(self: &Self) -> DynamicImage
```

Flip this image horizontally

```
pub fn rotate90(self: &Self) -> DynamicImage
```

Rotate this image 90 degrees clockwise.

```
pub fn rotate180(self: &Self) -> DynamicImage
```

Rotate this image 180 degrees clockwise.

```
pub fn rotate270(self: &Self) -> DynamicImage
```

Rotate this image 270 degrees clockwise.

```
pub fn write_to<W: Write, F: Into<ImageOutputFormat»(self: &Self, w: &mut W, format:
F) -> ImageResult<()>
```

Encode this image and write it to “‘w“‘

```
pub fn save<Q> where Q: AsRef<Path>(self: &Self, path: Q) -> ImageResult<()>
```

Saves the buffer to a file at the path specified.

The image format is derived from the file extension.

```
pub fn save_with_format<Q> where Q: AsRef<Path>(self: &Self, path: Q, format: ImageFormat)
-> ImageResult<()>
```

Saves the buffer to a file at the specified path in the specified format.

See [‘save_buffer_with_format‘](fn.save_buffer_with_format.html) for supported types.

```
impl Send for DynamicImage
```

```
impl Sync for DynamicImage
```

```
impl Unpin for DynamicImage
```

```
impl UnwindSafe for DynamicImage
```

```
impl RefUnwindSafe for DynamicImage
```

```
impl GenericImageView for DynamicImage
```

```
 type Pixel = color::Rgba<u8>;

 type InnerImageView = Self;

 fn dimensions(self:  &Self) -> (u32,u32)

 fn bounds(self:  &Self) -> (u32,u32,u32,u32)

 fn get_pixel(self:  &Self, x:  u32, y:  u32) -> color::Rgba<u8>

 fn inner(self:  &Self) -> &<Self as >::InnerImageView

impl GenericImage for DynamicImage

 type InnerImage = DynamicImage;

 fn put_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  color::Rgba<u8>)

 fn blend_pixel(self:  &mut Self, x:  u32, y:  u32, pixel:  color::Rgba<u8>)
```

DEPRECATED: Use iterator 'pixels_mut' to blend the pixels directly.

```
 fn get_pixel_mut(self:  &mut Self, _:  u32, _:  u32) -> &mut color::Rgba<u8>
```

DEPRECATED: Do not use is function: It is unimplemented!

```
 fn inner_mut(self:  &mut Self) -> &mut <Self as >::InnerImage

impl Clone for DynamicImage

 fn clone(self:  &Self) -> DynamicImage

impl Eq for DynamicImage

impl PartialEq<DynamicImage> for DynamicImage

 fn eq(self:  &Self, other:  &DynamicImage) -> bool

 fn ne(self:  &Self, other:  &DynamicImage) -> bool

impl Debug for DynamicImage

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl Hash for DynamicImage

 fn hash<__H: $crate::hash::Hasher>(self:  &Self, state:  &mut __H) -> ()

impl StructuralPartialEq for DynamicImage

impl StructuralEq for DynamicImage
```

1.304 Struct pub image::Delay

```
pub struct Delay{
    // some fields omitted
}
```

The delay of a frame relative to the previous one.

Implementations

`impl Delay`

`pub fn from_numer_denom_ms(numerator: u32, denominator: u32) -> Self`

Create a delay from a ratio of milliseconds.

# Examples

‘‘‘ use image::Delay; let delay_10ms = Delay::from_numer_denom_ms(10, 1); ‘‘‘

`pub fn from_saturating_duration(duration: Duration) -> Self`

Convert from a duration, clamped between 0 and an implemented defined maximum.

The maximum is *at least* ‘i32::MAX‘ milliseconds. It should be noted that the accuracy of the result may be relative and very large delays have a coarse resolution.

# Examples

‘‘‘ use std::time::Duration; use image::Delay;

let duration = Duration::from_millis(20); let delay = Delay::from_saturating_duration(duration); ‘‘‘

`pub fn numer_denom_ms(self: Self) -> (u32,u32)`

The numerator and denominator of the delay in milliseconds.

This is guaranteed to be an exact conversion if the ‘Delay‘ was previously created with the ‘from_numer_denom_ms‘ constructor.

`impl Send for Delay`

`impl Sync for Delay`

`impl Unpin for Delay`

`impl UnwindSafe for Delay`

`impl RefUnwindSafe for Delay`

`impl Clone for Delay`

`fn clone(self: &Self) -> Delay`

`impl Copy for Delay`

```
impl Eq for Delay

impl PartialEq<Delay> for Delay

 fn eq(self:  &Self, other:  &Delay) -> bool

 fn ne(self:  &Self, other:  &Delay) -> bool

impl PartialOrd<Delay> for Delay

 fn partial_cmp(self:  &Self, other:  &Delay) -> $crate::option::Option<$crate::cmp::Ordering>

impl Debug for Delay

 fn fmt(self:  &Self, f:  &mut $crate::fmt::Formatter<'_>) -> $crate::fmt::Result

impl StructuralPartialEq for Delay

impl StructuralEq for Delay
```

1.305 Struct pub image::Frame

```rust
pub struct Frame{
    // some fields omitted
}
```

A single animation frame

Implementations

```
impl Frame

 pub fn new(buffer:  RgbaImage) -> Frame
```

Contructs a new frame without any delay.

```
 pub fn from_parts(buffer:  RgbaImage, left:  u32, top:  u32, delay:  Delay) -> Frame
```

Contructs a new frame

```
 pub fn delay(self:  &Self) -> Delay
```

Delay of this frame

```
 pub fn buffer(self:  &Self) -> &RgbaImage
```

Returns the image buffer

```
 pub fn buffer_mut(self:  &mut Self) -> &mut RgbaImage
```

Returns a mutable image buffer

```
 pub fn into_buffer(self:  Self) -> RgbaImage
```

Returns the image buffer

```
pub fn left(self:  &Self) -> u32
```

Returns the x offset

```
pub fn top(self:  &Self) -> u32
```

Returns the y offset

```
impl Send for Frame
```

```
impl Sync for Frame
```

```
impl Unpin for Frame
```

```
impl UnwindSafe for Frame
```

```
impl RefUnwindSafe for Frame
```

```
impl Clone for Frame
```

```
fn clone(self:  &Self) -> Frame
```

1.306 Struct pub image::Frames

```
pub struct Frames<'a>{
    // some fields omitted
}
```

An implementation dependent iterator, reading the frames as requested

Implementations

```
impl<'a> Frames<'a>
```

```
pub fn new(iterator:  Box<Iterator<Item=ImageResult<Frame»>) -> Self
```

Creates a new ‘Frames‘ from an implementation specific iterator.

```
pub fn collect_frames(self:  Self) -> ImageResult<Vec<Frame»
```

Steps through the iterator from the current frame until the end and pushes each frame into a ‘Vec‘. If en error is encountered that error is returned instead.

Note: This is equivalent to ‘Frames::collect::<ImageResult<Vec<Frame»>()‘

```
impl<'a> !Send for Frames<'a>
```

```
impl<'a> !Sync for Frames<'a>
```

```
impl<'a> Unpin for Frames<'a>
```

```
impl<'a> !UnwindSafe for Frames<'a>

impl<'a> !RefUnwindSafe for Frames<'a>

impl<'a> Iterator for Frames<'a>

 type Item = ImageResult<Frame>;

 fn next(self:  &mut Self) -> Option<ImageResult<Frame»
```