Generic Image Library Design Guide

Author:

Lubomir Bourdev (lbourdev@adobe.com) and Hailin Jin (hljin@adobe.com)
Adobe Systems Incorporated

Version:

1.0

Date:

May 5, 2006

This document describes the design of the Generic Image Library, a C++ image-processing library that abstracts image representation from algorithms on images. It covers more than you need to know for a causal use of GIL. You can find a quick, jump-start GIL tutorial on the main GIL page at http://opensource.adobe.com/gil

- 1. Overview
- 2. Comparison with Other Image Libraries
- 3. About Concepts
- 4. Point
- 5. Channel
- 6. Color Space
- 7. Pixel
- 8. Pixel Iterator
  - Fundamental Iterator
  - Iterator Adaptor
  - Step Iterator
  - Pixel Locator
  - Pixel Image Iterator
- 9. Image View
  - Creating Views from Raw Pixels
  - Creating Image Views from Other Image Views
- 10. Image
- 11. Run-time specified images and image views
- 12. Algorithms
- 13. Useful Metafunctions and Typedefs
- 14. Sample Code
  - Pixel-level Sample Code
  - Creating a Copy of an Image with a Safe Buffer
  - Histogram
  - Using Image Views
- 15. Extending the Generic Image Library
  - Defining New Color Spaces
  - Defining New Channel Types
  - Defining New Pixel Iterators and Images
- 16. Conclusion
- 17. Acknowledgements

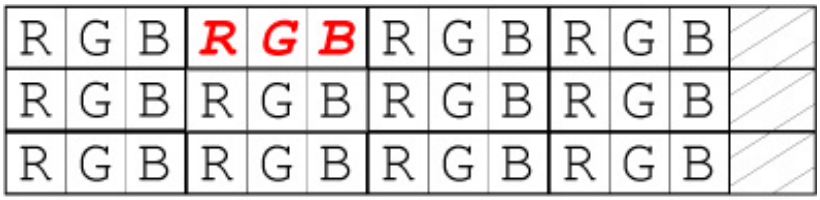
---

1. Overview

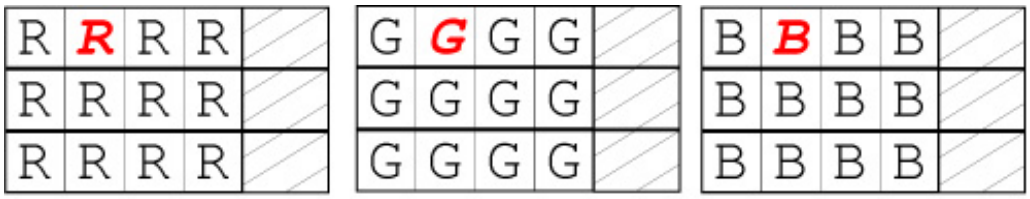
Images are essential in any image processing, vision and video project, and yet the variability in image representations makes it

difficult to write imaging algorithms that are both generic and efficient. In this section we will describe some of the challenges that we would like to address.

In the following discussion an *image* is a 2D array of pixels. A *pixel* is a set of color channels that represents the color at a given point in an image. Each *channel* represents the value of a color component. There are two common memory structures for an image. *Interleaved* images are represented by grouping the pixels together in memory and interleaving all channels together, whereas *planar* images keep the channels in separate color planes. Here is a 4x3 RGB image in which the second pixel of the first row is marked in red, in interleaved form:



and in planar form:



Note also that rows may optionally be aligned resulting in a potential padding at the end of rows.

The Generic Image Library (GIL) supports images that vary in:

- Structure (planar vs. interleaved)
- Color space and presence of alpha (RGB, RGBA, CMYK, etc.)
- Channel depth (8-bit, 16-bit, etc.)
- Order of channels (RGB vs. BGR, etc.)
- Row alignment policy (no alignment, word-alignment, etc.)

It also supports user-defined models of images, and images whose parameters are specified at run-time. GIL abstracts image representation from algorithms applied on images and allows us to write the algorithm once and have it work on any of the above image variations while generating code that is comparable in speed to that of hand-writing the algorithm for a specific image type.

This document follows bottom-up design. Each section defines concepts that build on top of concepts defined in previous sections. It is recommended to read the sections in order.

---

2. Comparison with Other Image Libraries

In this section we compare GIL against some of the most popular Computer Vision/Image Processing libraries.

ITK (http://www.itk.org)

ITK stands for Insight Segmentation and Registration Toolkit. It is an open source C++ system designed to support medical imaging related tasks such as registration and segmentation. The image class in ITK is templated over pixel type and number of dimensions. The user can create an image of an arbitrary pixel type and arbitrary dimensions. Unlike GIL, there is no unification in images of interleaved pixel channels and images of planar pixel channels and pixels of the same color space but of different channel ordering. As a result, an interleaved RGB image is fundamentally different from a planar RGB image which means pixels cannot be directly assigned to or compared with each other; Similarly, a BGR pixel cannot be automatically assigned to a RGB pixel; algorithms written for RGB images have to be re-written for BGR images. ITK also has a weak notation of iterators. For instance, iterators in ITK only support traveling along pre-defined paths while GIL supports 2D pixel navigation. ITK does not support a view of an image with a

different color space. A strong point of ITK is that it has a very large support base. It is developed by three commercial organizations and three academic institutes. It supports a broad class of medical imaging algorithms.

VXL (http://vxl.sourceforge.net)

VXL is a C++ library for Computer Vision. VIL is its image library. The design of the image library in VXL is very similar to that of ITK. The user can create images of arbitrary pixel types and arbitrary dimensions. The image dimension is a run-time parameter instead of being a compile-time template argument in ITK. Similar to ITK, VXL does not unify images of interleaved pixels with images of planar pixels and it does not allow for 2D pixel navigation. Therefore, algorithms are written based on pixel pointers and pixel steps, which makes it hard to abstract algorithms. It has very limited support for creating views of an image. For instance, it allows views of images with different strides but not views with different color spaces. VXL has a reasonable implementation of algorithms. The strong point of VXL is that it has a good implementation of a mathematics library called VNL. VNL supports vector/matrix operations, sparse matrics, linear algebra, polynomials, FFT and more. Indeed, ITK makes use of VNL, too.

CIMG (http://cimg.sourceforge.net)

CIMG stands for "Cool Image." It is fairly lightweight image processing library. It consists only of one header file. CIMG is able to represent images up to 4 dimensions with templated pixel types. The design of the CIMG image class seems to be "One class fits all." For instance, it does not have a notion of iterators. All the algorithms are implemented through the member function interface. It suffers from most limitations mentioned in ITK and VXL. It cannot easily be used with STL algorithms.

OpenCV (http://sourceforge.net/projects/opencvlibrary)

OpenCV is a collection of algorithms for Computer Vision problems. It was initially developed by Intel and then released to the open source community. One strong feature of OpenCV is that it implements many computer vision algorithms, for instance object recognition, 3D reconstruction, and image motion analysis. Another strong feature of OpenCV is that it utilizes Intel Integrated Performance Primitives, suggesting high performance at least on the Intel platform. It is good for people in Computer Vision who want to try some standard algorithms but do not want to implement them from scratch. However, OpenCV is a really C library, and therefore it greatly lacks the features of C++ and Generic Programming. For instance, OpenCV can create images only of concrete pixel types and concrete pixel organization. Most of its functions are written through raw memory buffer interfaces. No iterators, nor image views. Extending to new image types is extremely difficult.

VIGRA (http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra)

VIGRA stands for "Vision with Generic Algorithms". It is developed mainly by Ullrich Köthe at the University of Hamburg. Among the libraries we have reviewed, VIGRA best conforms to the spirit of Generic Programming. Its image classes are very well designed and support two-dimensional images of arbitrary pixel types. VIGRA provides containers to manage arrays of images. It has extensive support for various image iterators including two-dimensional image iterators, multi-dimensional iterators and iterator adaptors for accessing subsets of pixel channels. The iterators allow for STL algorithms to be used. It also has a fairly good collection of basic mathematical tools including vectors, matrices, and polynomials. Although it does not have as many image processing algorithms as ITK, most of the common ones are supported. Compared to GIL, VIGRA noticeably lacks the support for planar pixel structure which means the user cannot create images that have separate color channels and the user cannot operate the existing planar image memory. VIGRA does not have support for unifying pixels of the same color space but different channel orderings: a RGB pixel cannot be automatically assigned to a BGR pixel for instance. Moreover, GIL has a better way of creating image views from existing images and concatenating image views.

All of the aforementioned libraries have the problem that image parameters are compile-time bound. For example, it would be very difficult to read an image from disk and perform an arbitrary image operation on it, in its native (run-time specified) color space and channel depth.

We do not claim that GIL is strictly superior to other image libraries. Currently, one weak point of GIL is that not many image algorithms are implemented in it yet. It is not yet extended to handle multi-dimensional images. The aforementioned libraries have been designed with certain requirements in mind, and they are popular because they deliver very well against those design requirements. The design goals behind GIL are somewhat different, as outlined in the conclusion section.

---

3. About Concepts

All constructs in GIL are models of GIL concepts. A *concept* is a set of requirements that a type (or a set of related types) must fulfill to be used correctly in generic algorithms. The requirements include syntactic and algorithming guarantees. For example, GIL's class `pixel` is a model of GIL's **`PixelConcept`**. The user may substitute the pixel class with one of their own, and, as long as it satisfies the requirements of **`PixelConcept`**, all other GIL classes and algorithms can be used with it. See the boost libraries (www.boost.org) for more information on concepts.

In this document we will use a syntax for defining concepts that is described in an upcoming proposal for a Concepts extension to C++0x. A link to the proposal is not yet available, but the syntax is hopefully intuitive. The code examples may be simplified (for example, "`typename`" is often removed when it is clear that the construct is a type.

Here are some common concepts that will be used in GIL:

```
template <typename T>
concept DefaultConstructible {
    T::T();
};

template <typename T>
concept CopyConstructible {
    T::T(const T&);
};

template <typename T>
concept Assignable {
    T& operator=(const T&);
};

template <typename T>
concept EqualityComparable {
    bool operator==(const T& x, const T& y);
    bool operator!=(const T& x, const T& y) { return !(x==y); }
};

template <typename T>
concept LessThanComparable {
    bool operator<(const T& x, const T& y);
};

template <typename T>
concept ValueType : DefaultConstructible<T>, CopyConstructible<T>, Assignable<T>, EqualityComparable<T>
{};
```

4. Point

A point defines the location of a pixel inside an image. It can also be used to describe the dimensions of an image. In most general terms, points are N-dimensional and model the following pseudo-interface:

```
template <typename P>
concept PointNDConcept : ValueType<P> {
    // the type of coordinate along each axis
    template <size_t D> struct axis { typename coord_t; };

    static const size_t num_dimensions;

    // accessor/modifier of the value of each axis.
    template <size_t D> const axis<D>::coord_t& v() const;
    template <size_t D>       axis<D>::coord_t& v();
};
```

GIL uses a two-dimensional point, which is a refinement of **`PointNDConcept`** in which both dimensions are of the same type:

```
template <typename P>
concept Point2DConcept : PointNDConcept<P> {
    where num_dimensions==2;
    where axis<0>::coord_t == axis<1>::coord_t;

    typename value_type = axis<0>::coord_t;

    value_type x,y;
};
```

Related Concepts:

- **PointNDConcept**<T>
- **Point2DConcept**<T>

Implementation:

GIL only provides a model of **`Point2DConcept`**, `point2<T>` where `T` is the coordinate type.

---

5. Channel

A channel indicates the intensity of a color component (for example, the red channel in an RGB pixel). We typically want to get, compare and set the value of a channel. GIL channels must provide the following pseudo-interface:

```
template <typename T>
concept ChannelConcept : EqualityComparable<T>, LessThanComparable<T>, CopyConstructible<T> {
    typename value_type      = T;
    typename reference       = T&;
    typename pointer         = T*;
    typename const_reference = const T&;
    typename const_pointer   = const T*;

    static value_type T::max_value();
    static value_type T::min_value();

    value_type channel_multiply(T x, T y); // = x * y / T::max_value()
    value_type channel_invert(T x);        // = T::max_value() - x + T::min_value();
};

template <typename T>
concept MutableChannelConcept : ChannelConcept<T>, Assignable<T> {};

template <typename T>
concept ChannelValueConcept : ChannelConcept<T>, ValueType<T> {};

template <typename T>
concept MutableChannelValueConcept : ChannelValueConcept<T>, MutableChannelConcept<T> {};

// Two channel types are compatible if they are the same (ignoring constness and references).
template <ChannelConcept T1, ChannelConcept T2>
concept ChannelsCompatibleConcept {
    where T1::value_type == T2::value_type;
};

// For one channel to be convertible to another, channel_convert must be defined
template <ChannelConcept SRC_T, ChannelValueConcept DST_T>
concept ChannelConvertibleConcept {
```

```
    DST_T channel_convert(const SRC_T&);
};
```

Note the distinction between **ChannelConcept** and its refinement, **ChannelValueConcept**. Channel references are models of the former but not of the latter. Also operators -,+,+=,-= over compatible channels may be required by certain operations.

Abstracting the type of a channel reference and pointer allows one to provide proxy classes to implement them. This could be used, for example, to create a model of a channel whose size is not divisible by bytes (for example, a channel of a single bit).

Note that the associated typedefs and the static functions `min_value` and `max_value` are not necessarily member typedefs of the channel class. We use a traits class, **channel_traits**, to define them, so that built-in types may be models of channels. That means that to get the reference of a channel, for example, you must use:

```
typename channel_traits<T>::reference;
```

Related Concepts:

- **ChannelConcept**<T>
- **ChannelValueConcept**<T>
- **MutableChannelConcept**<T>
- **MutableChannelValueConcept**<T>
- **ChannelsCompatibleConcept**<T1,T2>
- **ChannelConvertibleConcept**<SRC_T,DST_T>

Implementation:

GIL provides default implementations for 8-bit, 16-bit and 32-bit channels using built-in types `boost::uint8_t`, `boost::uint16_t` and `float`, called `bits8`, `bits16` and `bits32` respectively. Each of them is convertible to each other.

---

6. Color Space

A color space captures the number, ordering and interpretation of channels comprising a pixel. Two color spaces are *compatible* if they have the same set of channels. Compatible color spaces may only differ in ordering of the channels. For example, RGB and BGR are compatible, while RGBA and RGB are not. Compatible color spaces have the same *base* color space. For example, RGBA is the base color space and ARGB and ABGR are derived color spaces. Color spaces must model the following pseudo-interface:

```
template <typename CS>
concept ColorSpaceTypeConcept {
    typename base; where ColorSpaceTypeConcept<base>;       // The base color space.
    static const int num_channels; // The number of channels.
};

template <typename CS1, typename CS2> where { ColorSpaceTypeConcept<CS1>, ColorSpaceTypeConcept<CS2> }
concept ColorSpacesCompatibleConcept {
    where CS1::base == CS2::base;
};
```

Related Concepts:

- **ColorSpaceTypeConcept**<CS>
- **ColorSpacesCompatibleConcept**<CS1,CS2>

Implementation:

GIL currently provides the following color spaces: **gray_t**, **rgb_t**, **bgr_t** (derived from **rgb_t**), **lab_t**, **rgba_t**, **argb_t** (derived from **rgba_t**) and **cmyk_t**.

---

7. Pixel

A pixel is a set of channels defining the color at a given point in an image. The number, ordering and interpretation of the channels are defined by the pixel's color space. In a general case each channel may be of a different type, in which case the pixel is *heterogeneous*. Pixel-level operations typically involve getting/setting the channels of a pixel. The representation of the pixel in memory (planar vs interleaved) is unimportant and is therefore not part of the pixel concept.

Two pixels are *compatible* if their color spaces and channels are compatible. Note that constness, memory organization and reference/value are ignored. For example, an 8-bit RGB planar reference is compatible to a constant 8-bit BGR interleaved pixel value. Most pairwise pixel operations (copy construction, assignment, equality, etc.) are only defined for compatible pixels.

Pixels provide channel accessors/modifiers in two ways - based on the *semantic* and on the *physical* index of a channel. Semantic indices are consistent across compatible color spaces. For example, the first semantic channel of RGB and of BGR is red. A physical index corresponds to the order of the channels in memory. The first physical channel index of RGB is red, while it is blue for BGR. Nearly all per-channel operations (equality, assignment, etc.) pair channels based on their semantic indices.

Pixels must model the following pseudo-interface:

```
template <typename P>
concept HeterogeneousPixelConcept : CopyConstructible<T>, EqualityComparable<T> {
    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
        bool operator==(const P&, const P2&);
    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
        bool operator!=(const P&, const P2&);

    typename color_space_t;    where ColorSpaceTypeConcept<color_space_t>;
    typename pixel_value_type; where HeterogeneousPixelValueConcept<pixel_value_type>;

    template <size_t D> struct kth_channel {
        typename type;         where ChannelConcept<type>; // the type of each channel.
    }

    static const size_t num_channels = color_space_t::num_channels;

    template <size_t D> kth_channel<D>::type::const_reference semantic_channel() const;
    template <size_t D> kth_channel<D>::type::const_reference channel() const;

    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
         pixel_value_type operator+(const P&,const P2&);
    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
         pixel_value_type operator-(const P&,const P2&);
    template <ScalarValue S> pixel_value_type operator*(const P&, const S& s);
    template <ScalarValue S> pixel_value_type operator/(const P&, const S& s);
};

// Mutable pixels are assignable and provide channel modifiers
template <typename P>
concept MutableHeterogeneousPixelConcept : HeterogeneousPixelConcept<P>, Assignable<P> {
    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
         P& operator=(const P2&);

    template <size_t D> kth_channel<D>::type::reference semantic_channel();
    template <size_t D> kth_channel<D>::type::reference channel();

    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
```

```
        P operator+=(const P2&);
    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
        P operator-=(const P2&);
    template <ScalarValue S> P operator*=(const S& s);
    template <ScalarValue S> P operator/=(const S& s);
};

// Pixel values are default and copy constructible from a compatible pixel type
template <typename P>
concept HeterogeneousPixelValueConcept : HeterogeneousPixelConcept<P>, ValueType<P> {
    where pixel_value_type==P;

    template <HeterogeneousPixelConcept P2> where { PixelsCompatibleConcept<P,P2> }
        P::P(const P2&);
};

template <typename P>
concept MutableHeterogeneousPixelValueConcept : HeterogeneousPixelValueConcept<P>,
MutableHeterogeneousPixelConcept<P> {
};

// Pixels are compatible if they have compatible color spaces and the same channel values
template <HeterogeneousPixelConcept P1, HeterogeneousPixelConcept P2>
concept PixelsCompatibleConcept {
    where ColorSpacesCompatibleConcept<P1::color_space_t, P2::color_space_t>;
    template <size_t D> where P1::kth_channel<D>::type::value_type == P2::kth_channel<D>::type::
value_type;
};
```

The refinement GIL provides assumes all channels have the same type:

```
template <typename P>
concept PixelConcept : HeterogeneousPixelConcept<P> {
    typename channel_t = P::kth_channel<0>::type::value_type;

    typename channel_reference       = channel_t::reference;
    typename channel_const_reference = channel_t::const_reference;

    channel_const_reference P::operator[](size_t i) const;  // shortcut for physical channel accessor
};

template <typename P>
concept MutablePixelConcept : PixelConcept<P>, MutableHeterogeneousPixelConcept<P> {
    channel_reference       P::operator[](size_t i);
};

template <typename P>
concept PixelValueConcept : PixelConcept<P>, HeterogeneousPixelValueConcept<P> {};

template <typename P>
concept MutablePixelValueConcept : MutablePixelConcept<P>, PixelValueConcept<P> {};
```

GIL provides methods that can invoke a generic unary/binary/ternary operation on each set of logical channels of compatible pixels:

```
template <typename OP, HeterogeneousPixelConcept P1>
    OP for_each_channel(const P1&, OP op);
template <typename OP, HeterogeneousPixelConcept P1, HeterogeneousPixelConcept P2>
    where { PixelsCompatibleConcept<P1,P2> }
    OP for_each_channel(const P1&, const P2&, OP op);
template <typename OP, HeterogeneousPixelConcept P1, HeterogeneousPixelConcept P2,
HeterogeneousPixelConcept P3>
```

```
    where { PixelsCompatibleConcept<P1,P2>, PixelsCompatibleConcept<P1,P3> }
    OP for_each_channel(const P1&, const P2&, const P3&, OP op);

template <typename OP, MutableHeterogeneousPixelConcept DST>
    OP tranform_channels(DST&, OP op);
template <typename OP, MutableHeterogeneousPixelConcept DST, HeterogeneousPixelConcept P1>
    where { PixelsCompatibleConcept<DST,P1> }
    OP tranform_channels(const P1&, DST&, OP op);
template <typename OP, MutableHeterogeneousPixelConcept DST, HeterogeneousPixelConcept P1,
HeterogeneousPixelConcept P2>
    where { PixelsCompatibleConcept<DST,P1>, PixelsCompatibleConcept<DST,P2> }
    OP tranform_channels(const P1&, const P2&, DST&, OP op);
```

GIL provides additional methods for homogeneous pixels that set all channels to a given value, return the minimum and maximum channel value and the alpha channel value. (If the color space does not have alpha, the maximum channel value is returned):

```
// P Models PixelConcept
template <MutablePixelConcept P> void         fill_channels(P&, P::channel_const_reference val);
template <PixelConcept P>        P::channel_t min_channel(const P&);
template <PixelConcept P>        P::channel_t max_channel(const P&);
template <PixelConcept P>        P::channel_t alpha(const P&);
```

A pixel is *convertible* to a second pixel if it is possible to approximate its color in the form of the second pixel. Conversion is an explicit, non-symmetric and often lossy operation (due to both channel and color space approximation). Convertability requires fulfilling the following concept:

```
template <HeterogeneousPixelConcept SRC_T, MutableHeterogeneousPixelConcept DST_T>
concept PixelConvertibleConcept {
    void color_convert(const SRC_T&, DST_T&);
};
```

The distinction between **`HeterogeneousPixelConcept`** and **`HeterogeneousPixelValueConcept`** is analogous to that for channels - pixel references model both, but only pixel values model the latter.

Pixels of a given color space also allow accessing their channels by name. For example, `p.red` is the red channel of a pixel `p`. Standard channel names include `p.red`, `p.green`, `p.blue`, `p.gray`, **`p.alpha`**, `p.cyan`, `p.magenta`, `p.yellow`, `p.black`, `p.luminance`, `p.a`, `p.b`.

Related Concepts:

- **HeterogeneousPixelConcept**<PX>
- **MutableHeterogeneousPixelConcept**<PX>
- **HeterogeneousPixelValueConcept**<PX>
- **MutableHeterogeneousPixelValueConcept**<PX>
- **PixelConcept**<PX>
- **MutablePixelConcept**<PX>
- **PixelValueConcept**<PX>
- **MutablePixelValueConcept**<PX>
- **PixelsCompatibleConcept**<PX1,PX2>
- **PixelConvertibleConcept**<SRC_PX,DST_PX>

Implementation:

Many pixel operations are color-space agnostic and apply to each channel of the pixel. They are implemented using compile-time recursion using the semantic channel accessors/modifiers:

```
namespace detail {
    template <int N> struct channel_recursion {
        template <typename P1,typename P2> static void plus_eq(P1& p1, const P2& p2) {
            channel_recursion<N-1>::plus_eq(p1,p2);
            p1.template semantic_channel<N-1>()+=p2.template semantic_channel<N-1>();
        }
        template <typename P1,typename P2> static bool equal_to(const P1& p1, const P2& p2) {
            return channel_recursion<N-1>::equal_to(p1,p2) &&
                   p1.template semantic_channel<N-1>()==p2.template semantic_channel<N-1>();
        }
        template <typename P1,typename OP>
        static void for_each_channel(const P1& p1, OP op) {
            channel_recursion<N-1>::for_each_channel(p1,op);
            op(p1.template semantic_channel<N-1>());
        }
        ...
    };

    template<> struct channel_recursion<0> {
        template <typename P1,typename P2> static void plus_eq(P1& p1, const P2& p2) {}
        template <typename P1,typename P2> static bool equal_to(const P1& p1, const P2& p2) { return
true; }
        template <typename P1,typename OP> static void for_each_channel(const P1&,OP){}
        ...
    };
}

// per-channel operations like for_each_channel, transform_channels, fill_channels, etc. are using the
static recursion
template <typename OP, typename P1>
OP for_each_channel(const P1& p1, OP op) { return detail::channel_recursion<P1::num_channels>::
for_each_channel(p1,op); }
```

The color-space-specific aspect of pixels is implemented inside `color_base`. Color base classes are partial implementation of the empty `color_base` struct:

```
namespace detail {
    template <typename T, typename C> struct color_base {};
}
```

They represent an ordered set of elements of type `T` of a particular color space tag of type `C`.

The elements are typically channel values, but as we will see later in the implementation of planar pointer and planar reference, they can be channel pointers and channel references. Here is a synopsis of the RGB and BGR color base classes and the associated channel accessors:

```
namespace detail {
    template <typename T>
    struct homogeneous_color_base {
        template <int N> struct kth_element_t {
            typedef ... reference;
            typedef ... const_reference;
        };
    };

    template <typename T>   // Models ChannelConcept
    struct color_base<T,rgb_t> : public homogeneous_color_base<T> {
        T red,green,blue;

        color_base() {}
        color_base(T v0, T v1, T v2) : red(v0), green(v1), blue(v2) {}
    };
```

```
    template <typename T>   // Models ChannelConcept
    struct color_base<T,bgr_t> : public homogeneous_color_base<T> {
        T blue,green,red;

        color_base() {}
        color_base(T v0, T v1, T v2) : blue(v0), green(v1), red(v2) {}
    };
}
```

Note that currently GIL provides a model only for homogeneous pixels. To access the physical channels by index we use channel accessors, specializations of the `channel_accessor` struct:

```
namespace detail {
    template <typename CS,int N> struct channel_accessor;

    // the first physical channel in RGB
    template <>
    struct channel_accessor<rgb_t,0> {
        template <typename P> P::kth_element_t<0>::reference       operator()(      P& p) const {return p.
red;}
        template <typename P> P::kth_element_t<0>::const_reference operator()(const P& p) const {return p.
red;}
    };

    // the first physical channel in BGR
    template <>
    struct channel_accessor<bgr_t,0> {
        template <typename P> P::kth_element_t<0>::reference       operator()(      P& p) const {return p.
blue;}
        template <typename P> P::kth_element_t<0>::const_reference operator()(const P& p) const {return p.
blue;}
    };
}
```

(Having special classes for the accessors (as opposed to templated methods of `color_base`) is necessary because full member specialization is not allowed in a partially specialized class.)

A pixel is templated over the channel type `T` and color space tag `C`. It subclasses `color_base` and provides color-space agnostic operations, such as equality, assignment, operator+=, etc. which use the compile-time recursion above:

```
// GIL's model of PixelValueConcept
template <typename T, typename C>
struct pixel : public detail::color_base<T,C> {
public:
    typedef C                                  color_space_t;
    typedef channel_traits<T>::value_type      channel_t;
    typedef channel_traits<T>::reference       channel_reference;
    typedef channel_traits<T>::const_reference channel_const_reference;
    typedef pixel<channel_t,color_space_t>     pixel_value_type;
    template <int N>                           struct kth_channel_t { typedef channel_t type; };

    pixel(){}
    explicit pixel(T v)  { fill_channels(*this,v); }
    pixel(T v0, T v1);
    pixel(T v0, T v1, T v2);
    pixel(T v0, T v1, T v2, T v3);
    pixel(T v0, T v1, T v2, T v3, T v4);

    pixel(const pixel& p);
    template <typename T1, typename C1> pixel(const pixel<T1,C1>& p);
```

```
    template <typename T1, typename C1> pixel(const planar_ref<T1,C1>& p);

    template <typename P> pixel&        operator=(const P& p);

    template <typename P> bool          operator==(const P& p) const { return detail::
channel_recursion<num_channels>::equal_to(*this,p); }
    template <typename P> bool          operator!=(const P& p) const;

    channel_reference                   operator[](std::size_t i);
    channel_const_reference             operator[](std::size_t i) const;

    template <int N> channel_reference  channel();
    template <int N> channel_reference  semantic_channel();

    template <int N> channel_const_reference channel()          const { return detail::channel_accessor<C,
N>()(*this); }
    template <int N> channel_const_reference semantic_channel() const { return detail::
channel_accessor<typename C::base,N>()(*this); }

    ...
};
```

Interleaved pixels have the built-in reference (`pixel<T,C>&`). Planar pixels, however, have channels that are not together in memory and therefore a proxy class is required to represent a planar reference. A reference to a planar pixel is defined as a set of references to each corresponding channel. It is implemented as a subclass of `color_base`. Synopsis:

```
template <typename TR, typename C>      // TR is a channel reference
struct planar_ref : public detail::color_base<TR,C> {
    typedef C                                       color_space_t;
    typedef TR                                      channel_reference;
    typedef typename channel_traits<TR>::value_type channel_t;          // must be a value and have no
constness
    typedef pixel<channel_t,color_space_t>          pixel_value_type;   // type to return when returning
by value
    template <int N>                                struct kth_channel_t { typedef channel_t type; };

    template <typename P>               planar_ref(P& p);
    template <typename P> planar_ref&   operator=(const P& p);

    template <typename P> bool          operator==(const P& p)    const;
    template <typename P> bool          operator!=(const P& p)    const;

    channel_reference                   operator[](std::size_t i) const;

    template <int N> channel_reference channel()                  const;
    template <int N> channel_reference semantic_channel()         const;
};
```

The 8-bit immutable planar RGB reference, for example, is represented as `planar_ref<const bits8&, rgb_t>` and contains three constant references to the three corresponding channels. Thus **`planar_ref`** models **`PixelConcept`**, but not **`PixelValueConcept`**. The value type of a planar pixel is the same as that of interleaved pixel - i.e. for 8-bit RGB image it is `pixel<bits8,rgb_t>`.

Algorithms typically don't care whether they operate on pixel values or references; all they care about is setting and getting the channels. Thus algorithms should typically take models of **`PixelConcept`**, which would make them work for both planar and interleaved references and values.

---

8. Pixel Iterator

Fundamental Iterator

Pixel iterators are random-access iterators whose `value_type` models **`PixelValueConcept`**:

```
template <typename IT>
concept PixelIteratorConcept : RandomAccessIteratorConcept<IT> {
    where PixelValueConcept<value_type>;
    typaneme pixel_t = value_type;
    typename color_space_t;   where ColorSpaceTypeConcept<color_space_t>;
    typename channel_t;       where ChannelValueConcept<channel_t>;
    typename const_reference; where PixelConcept<const_reference>;
    typename const_t          where PixelIteratorConcept<const_t>;
    typename dynamic_step_t;  where PixelStepIteratorConcept<dynamic_step_t>;
    static const bool  is_base;   // is it an iterator adaptor
    static const bool  is_planar; // is it over planar data
};

template <typename IT>
concept MutablePixelIteratorConcept : PixelIteratorConcept<IT>, MutableRandomAccessIteratorConcept<IT> {};
```

Related Concepts:

- **PixelIteratorConcept**<PX>
- **MutablePixelIteratorConcept**<PX>

Implementation:

A built-in pointer to pixel, `pixel<T,C>*`, is GIL's model for pixel iterator over interleaved pixels. Since built-in pointers cannot have member typedefs, a traits class, `pixel_iterator_traits`, is used to store the typedefs of pixel iterators. pixel_iterator_traits subclasses from `iterator_traits` and thus includes all STL iterator traits too.

For planar pixels, GIL provides a planar pointer class, specialized for each color space. It contains a bundle of pointers to each channel by subclassing from `color_base<T*,C>`. Here is a synopsis of the proxy class that represents a pointer to a planar RGB pixel:

```
template <typename IC>  // IC is a channel pointer, like bits8* or const bits8*
struct planar_ptr<IC,rgb_t> : public color_base<IC, rgb_t> {
    typedef pixel<IC,rgb_tag> reference;   // planar reference

    planar_ptr();
    planar_ptr(const IC& ir, const IC& ig, const IC& ib);

    template <typename T1,typename C1> planar_ptr(const planar_ptr<T1,C1>&);
    template <typename T1,typename C1> planar_ptr& operator=(const planar_ptr<T1,C1>&);

    // These allow constructs like pointer = &value or pointer = &reference_proxy
    template <typename P> planar_ptr(P* pix);            // P models PixelConcept
    template <typename P> planar_ptr& operator=(P* pix); // P models PixelConcept

    template <int N> IC&       channel()               { return detail::channel_accessor<C,N>()(*this);}
    template <int N> IC const& channel()         const { return detail::channel_accessor<C,N>()(*this);}
    template <int N> IC&       semantic_channel()      { return detail::channel_accessor<typename C::
base,N>()(*this);}
    template <int N> IC const& semantic_channel() const { return detail::channel_accessor<typename C::
base,N>()(*this);}

    reference operator*() const { return reference(red, green, blue); }
```

```
    // all typical random access iterator operators
    planar_ptr& operator++()    { transform_channels(*this,*this,inc<IC>()); return *this; }
    planar_ptr& operator+=(difference_type d);
    ...
    difference_type operator-(const planar_ptr&);

    ...
    bool operator<(const planar_ptr&) const;
    ...
};
```

(In reality common code is abstracted in **planar_ptr_base** class, which uses boost::iterator_facade). In the above example operator++ shows how the per-channel operations are used to manipulate (in this case, increment) the channel pointers. This is possible because the per-channel operations (transform_channels, for_each_channel, fill_channels, max_channel, etc.) require only presence of the semantic_channel() method of its arguments, which **planar_ptr** provides. Note that those operations use compile-time recursion and in the above example the generated amounts to three pointer increments.

Iterator Adaptor

Iterator adaptor is an iterator that wraps around another iterator. Its traits supply the type of the underlying iterator and a boolean indicating the iterator is an adaptor. The iterator also provides access to its base iterator:

```
template <ForwardIteratorConcept IT>
concept IteratorAdaptorConcept {
    typename base_t; where ForwardIteratorConcept<base_t>;
    static const bool is_base=true;

    const base_t& base() const;
};

template <Mutable_ForwardIteratorConcept IT>
concept MutableIteratorAdaptorConcept : IteratorAdaptorConcept<IT> {};
```

GIL uses an iterator adaptor whose value type models **PixelConcept**. GIL's iterator adaptor can also provide the type of an otherwise identical iterator adaptor, except its base type is a step iterator (see the **Step Iterator** section for more):

```
template <PixelIteratorConcept IT>
concept PixelIteratorAdaptorConcept : IteratorAdaptorConcept<IT> {};

template <MutablePixelIteratorConcept IT>
concept MutablePixelIteratorAdaptorConcept : MutableIteratorAdaptorConcept<IT> {};
```

Related Concepts:

- **IteratorAdaptorConcept**<IT>
- **MutableIteratorAdaptorConcept**<IT>
- **PixelIteratorAdaptorConcept**<IT>
- **MutablePixelIteratorAdaptorConcept**<IT>

Implementation:

GIL provides several models of **PixelIteratorAdaptorConcept**:

- step_iterator_adaptor<IT>: An iterator adaptor that changes the fundamental step of the base iterator (see **Step Iterator**)
- **dereference_iterator_adaptor**<IT,FN>: An iterator that applies a unary function FN upon dereferencing. It is used, for example, for on-the-fly color conversion. It can be used to construct a shallow image "view" that pretends to have a different

color space or channel depth. See **Creating Image Views from Other Image Views** for more.

Step Iterator

Sometimes we want to traverse pixels with a unit step other than the one provided by the fundamental pixel iterators. Examples where this would be useful:

- a single-channel view of the red channel of an RGB interleaved image
- left-to-right flipped image (step = -fundamental_step)
- subsampled view, taking every N-th pixel (step = N*fundamental_step)
- traversal in vertical direction (step = number of bytes per row)
- any combination of the above (steps are multiplied)

Step iterators are forward iterators that allow changing the step between adjacent values:

```
template <ForwardIteratorConcept IT>
concept StepIteratorConcept {
    template <typename D> void IT::set_step(D step);
};

template <Mutable_ForwardIteratorConcept IT>
concept MutableStepIteratorConcept : StepIteratorConcept<IT> {};
```

GIL currently provides a step iterator whose `value_type` models **`PixelConcept`**. In addition, the step is specified in bytes. This is necessary, for example, when implementing an iterator navigating along a column of pixels - the size of a row of pixels may sometimes not be divisible by the size of a pixel; for example rows may be word-aligned.

To advance in bytes, the base iterator must be *byte-advanceable*. It must supply functions returning the current step in bytes, the byte distance between two iterators, and a reference a given distance in bytes away. It must also supply a function that advances an iterator a given distance in bytes. `byte_advanced` and `byte_advanced_ref` have a default implementation but some iterators may supply a more efficient version:

```
template <typename IT>
concept ByteAdvanceableIteratorConcept : RandomAccessIteratorConcept<IT> {
    ptrdiff_t     byte_step(const IT&);
    ptrdiff_t     byte_distance(const IT& , const IT&);
    void          byte_advance(IT&, ptrdiff_t byteDiff);
    IT            byte_advanced(const IT& p, ptrdiff_t byteDiff) { IT tmp; byte_advance(tmp,pyteDiff);
return tmp; }
    IT::reference byte_advanced_ref(const IT& p, ptrdiff_t byteDiff) { return *byte_advanced(p,
byteDiff); }
};
```

GIL's pixel step iterators model the following concept:

```
template <StepIteratorConcept IT>
concept PixelStepIteratorConcept : PixelIteratorAdaptorConcept<IT> {
    where ByteAdvanceableIteratorConcept<IT::base_t>;

    // Must be constructible from a base iterator and a step
    IT::IT(const base_t&, ptrdiff_t);
};

template <MutableStepIteratorConcept IT>
concept MutablePixelStepIteratorConcept : MutablePixelIteratorAdaptorConcept<IT> {
    where ByteAdvanceableIteratorConcept<IT::base_t>;
};
```

Related Concepts:

- **StepIteratorConcept**<IT>
- **MutableStepIteratorConcept**<IT>
- ByteAdvanceableIteratorConcept<IT>
- **PixelStepIteratorConcept**<IT>
- **MutablePixelStepIteratorConcept**<IT>

Implementation:

All standard iterators GIL supplies model ByteAdvanceableIteratorConcept. GIL's model for **`PixelStepIteratorConcept`** is the class **`pixel_step_iterator`**. It takes the base iterator as a template parameter and allows changing the step dynamically. It models **`StepIteratorConcept`**. GIL's implementation contains the base iterator and a `ptrdiff_t` denoting the number of bytes to skip for a unit step. It may also be used with a negative number. GIL provides a function to create a step iterator from a base iterator and a step:

```
template <typename I>  // Models PixelIteratorConcept, ByteAdvanceableIteratorConcept
typename pixel_iterator_traits<I>::dynamic_step_t make_step_iterator(const I& it, ptrdiff_t step);
```

Pixel Locator

A Locator allows for navigation in two or more dimensions. Locators are N-dimensional iterators in spirit, but we use a different name because they don't satisfy all the requirements of iterators. For example, they don't supply increment and decrement operators because it is unclear which dimension the operators should advance along. N-dimensional locators model the following concept:

```
template <typename LOC>
concept RandomAccessNDLocatorConcept : ValueType<LOC> {
    typename value_type;        // value over which the locator navigates
    typename reference;         // result of dereferencing
    typename difference_type; where PointNDConcept<difference_type>; // return value of operator-.
    typename const_t;           // same as LOC, but operating over immutable values
    typename cached_location_t; // type to store relative location (for efficient repeated access)
    typename domain_t = point_t;

    static const size_t num_dimensions; // dimensionality of the locator
    where num_dimensions = point_t::num_dimensions;

    // The difference_type and iterator type along each dimension. The iterators may only differ in
    // difference_type. Their value_type must be the same as this_type::value_type
    template <size_t D> struct axis {
        typename coord_t = point_t::axis<D>::coord_t;
        typename iterator; where RandomAccessIteratorConcept<iterator>; // iterator along N-th axis.
    };

    LOC& operator+=(const difference_type&);
    LOC& operator-=(const difference_type&);
    LOC operator+(const difference_type&) const;
    LOC operator-(const difference_type&) const;

    reference operator*() const;
    reference operator[](const difference_type&) const;

    // Storing relative location for faster repeated access and accessing it
    cached_location_t LOC::cache_location(const difference_type&) const;
    reference operator[](const cached_location_t&) const;

    // Accessing iterators along a given dimension at the current location or at a given offset
    template <size_t D> axis<D>::iterator&       LOC::axis_iterator();
    template <size_t D> axis<D>::iterator const& LOC::axis_iterator() const;
```

```
    template <size_t D> axis<D>::iterator        LOC::axis_iterator(const difference_type&) const;
};

template <typename LOC>
concept MutableRandomAccessNDLocatorConcept : RandomAccessNDLocatorConcept<LOC> {
    where Mutable<value_type>;
};
```

Two-dimensional locators have additional requirements:

```
template <typename LOC>
concept RandomAccess2DLocatorConcept : RandomAccessNDLocatorConcept<LOC> {
    where num_dimensions==2;
    where Point2DConcept<point_t>;

    typename x_iterator = axis<0>::iterator;
    typename y_iterator = axis<1>::iterator;
    typename x_coord_t  = axis<0>::coord_t;
    typename y_coord_t  = axis<1>::coord_t;

    x_iterator&       LOC::x();
    x_iterator const& LOC::x() const;
    y_iterator&       LOC::y();
    y_iterator const& LOC::y() const;

    x_iterator LOC::x_at(const difference_type&) const;
    y_iterator LOC::y_at(const difference_type&) const;
    LOC xy_at(const difference_type&) const;

    // x/y versions of all methods that can take difference type
    x_iterator x_at(x_coord_t, y_coord_t) const;
    y_iterator y_at(x_coord_t, y_coord_t) const;
    LOC xy_at(x_coord_t, y_coord_t) const;
    reference operator()(x_coord_t, y_coord_t) const;
    cached_location_t cache_location(x_coord_t, y_coord_t) const;
};

template <typename LOC>
concept MutableRandomAccess2DLocatorConcept : RandomAccess2DLocatorConcept<LOC>,
MutableRandomAccessNDLocatorConcept<LOC> {};
```

The locators GIL uses operate over models of **`PixelConcept`**. Its x and y dimension types are the same. It models the following pseudo-interface:

```
template <typename LOC>
concept PixelLocatorConcept : RandomAccess2DLocatorConcept<LOC> {
    where PixelValueConcept<value_type>;
    where PixelIteratorConcept<x_iterator>;
    where PixelIteratorConcept<y_iterator>;
    where x_coord_t == y_coord_t;

    typename pixel_t       = value_type;
    typename channel_t     = pixel_t::channel_t;
    typename color_space_t = pixel_t::color_space_t;
    typename coord_t       = x_coord_t;
    typename dynamic_step_t; where PixelLocatorConcept<dynamic_step_t>; where
StepIteratorConcept<dynamic_step_t::x_iterator>;

    LOC::LOC(const x_iterator&, ptrdiff_t);  // construct from horizontal iterator and y-step in bytes
    ptrdiff_t LOC::row_bytes() const;        // number of bytes between vertically adjacent pixels
    ptrdiff_t LOC::pix_bytestep() const;     // number of bytes between horizontally adjacent pixels
```

```
};

template <typename LOC>
concept MutablePixelLocatorConcept : PixelLocatorConcept<LOC>, MutableRandomAccess2DLocatorConcept<LOC>
{};
```

Related Concepts:

- **RandomAccessNDLocatorConcept**<LOC>
- **MutableRandomAccessNDLocatorConcept**<LOC>
- **RandomAccess2DLocatorConcept**<LOC>
- **MutableRandomAccess2DLocatorConcept**<LOC>
- **PixelLocatorConcept**<IT>
- **MutablePixelLocatorConcept**<IT>

Implementation:

GIL provides a model of **`PixelLocatorConcept`** that takes a model of **`PixelStepIteratorConcept`** as a template parameter. (When instantiated with a model of **`MutableStepIteratorConcept`**, it models **`MutablePixelLocatorConcept`**).

```
template <typename S_IT>  // where { PixelStepIteratorConcept<S_IT> }
class pixel_2d_locator;
```

The step of `S_IT` must be the number of bytes per row. The class **`pixel_2d_locator`** is a wrapper around `S_IT` and uses it to navigate vertically, while its base iterator is used to navigate horizontally.

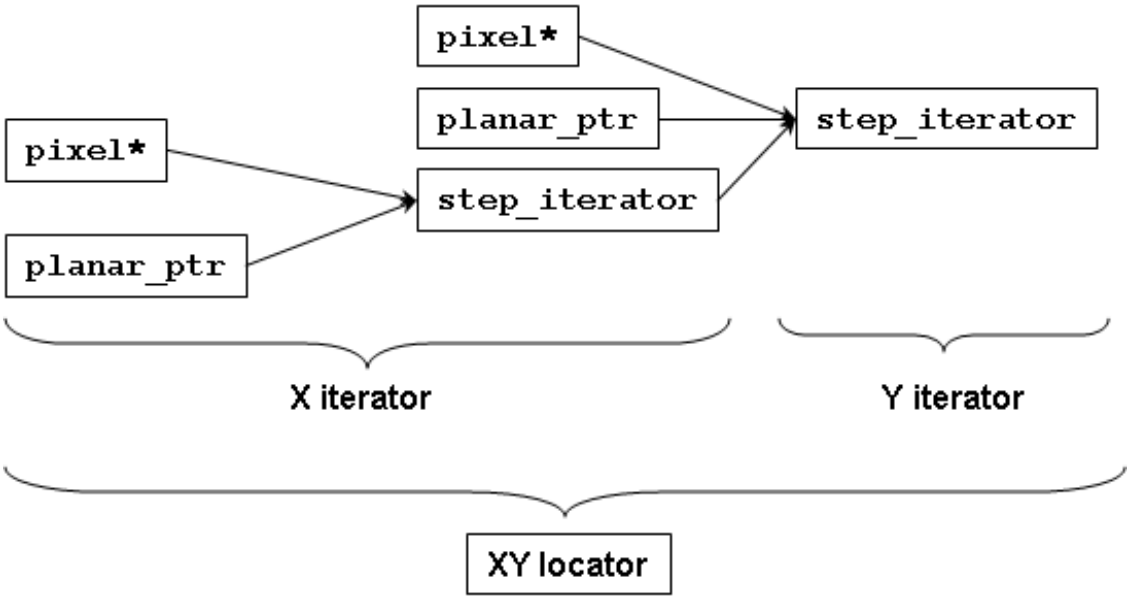
Combining fundamental and step iterators allows us to create locators that describe complex pixel memory organizations. First, we have a choice of iterator to use for horizontal direction, i.e. for iterating over the pixels on the same row. Using the fundamental and step iterators gives us four choices:

- `pixel<T,C>*` (for interleaved images)
- `planar_ptr<T*,C>` (for planar images)
- **`pixel_step_iterator`**`<pixel<T,C>*>` (for interleaved images with non-standard step)
- **`pixel_step_iterator`**`<planar_ptr<T*,C> >` (for planar images with non-standard step)

Of course, one could provide their own custom x-iterator. One such example described later is an iterator adaptor that performs color conversion when dereferenced.

Given a horizontal iterator `X_IT`, we could choose the *y-iterator*, the iterator that moves along a column, as `pixel_step_iterator<X_IT>` with a step equal to the number of bytes per row. Again, one is free to provide their own y-iterator.

Then we can instantiate **`pixel_2d_locator`**`<pixel_step_iterator<X_IT> >` to obtain a 2D pixel locator, as the diagram indicates:



Here is some sample code using locators:

```
loc=img.xy_at(10,10);            // start at pixel (x=10,y=10)
above=loc.cache_location(0,-1);  // remember relative locations of neighbors above and below
below=loc.cache_location(0, 1);
++loc.x();                       // move to (11,10)
loc.y()+=15;                     // move to (11,25)
loc-=pixel_type(1,1);            // move to (10,24)
*loc=(loc(0,-1)+loc(0,1))/2;     // set pixel (10,24) to the average of (10,23) and (10,25)
*loc=(loc[above]+loc[below])/2;  // the same, but faster using cached relative neighbor locations
```

The standard GIL locators are fast and lightweight objects. For example, the locator for a simple interleaved image consists of one raw pointer to the pixel location plus one integer for the row size in bytes, for a total of 8 bytes. `++loc.x()` amounts to incrementing a raw pointer (or N pointers for planar images). Computing 2D offsets is slower as it requires multiplication and addition. Filters, for example, need to access the same neighbors for every pixel in the image, in which case the relative positions can be cached into a raw byte difference using `cache_location`. In the above example `loc[above]` for simple interleaved images amounts to a raw array index operator.

Pixel Image Iterator

Pixel image iterator is a random access iterator over all pixels in an image, in the natural memory-friendly order left-to-right inside top-to-bottom. It satisfies the following pseudo-interface:

```
template <PixelIteratorConcept IT>
concept PixelImageIteratorConcept {
    size_t IT::width() const;  // the number of pixels in a row
    size_t IT::x()     const;  // the index of the current pixel horizontally

    bool   IT::is_contiguous() const; // returns true if there is no padding at the end of the rows
};

template <MutablePixelIteratorConcept IT>
concept MutablePixelImageIteratorConcept : PixelImageIteratorConcept<IT> {};
```

Related Concepts:

- **PixelImageIteratorConcept**<IT>
- **MutablePixelImageIteratorConcept**<IT>

Implementation:

GIL's model for **`PixelImageIteratorConcept`** is a class templated over a pixel locator:

```
template <typename LOC>  // Models PixelLocatorConcept
class pixel_image_iterator {
public:
    pixel_image_iterator(const LOC& loc, int x, int width);

    pixel_image_iterator& operator++(); // if (++_x<_width) ++_p.x(); else "carriage return"

    ...
private:
    size_t _x, _width;
    LOC _p;
};
```

It is a wrapper over the locator and keeps track of the current x-position and the width of the image, so that it can properly skip over potential padding at the end of rows.

Iterating through the pixels in an image using **`pixel_image_iterator`** is slower than going through all rows and using the x-iterator at each row. This is because two comparisons are done per iteration step - one for the end condition of the loop using the iterators, and one inside `pixel_image_iterator::operator++` to determine whether we are at the end of a row. For fast operations, such as pixel copy, this second check adds about 15% performance delay (measured for interleaved images on Intel platform). GIL overrides some STL algorithms, such as **`std::copy`** and **`std::fill`**, when invoked with `pixel_image_iterator-s`, to go through each row using their base x-iterators, and, if the image has no padding (i.e. `is_contiguous()` returns true) to simply iterate using the x-iterators directly.

---

9. Image View

An image view is a generalization of STL's range concept to multiple dimensions. Similar to ranges (and iterators), image views are shallow, don't own the underlying data and don't propagate their constness over the data. For example, a constant image view cannot be resized, but may allow modifying the pixels. For pixel-immutable operations, use constant-value image view (also called non-mutable image view). Most general N-dimensional views satisfy the following pseudo-interface:

```
template <typename VIEW>
concept RandomAccessNDImageViewConcept : ValueType<VIEW> {
    typename value_type;
    typename reference;       // result of dereferencing
    typename difference_type; // result of operator-(iterator,iterator) (1-dimensional!)
    typename const_t;  where RandomAccessNDImageViewConcept<const_t>; // same as VIEW, but over immutable
values
    typename domain_t = point_t;
    typename point_t;  where PointNDConcept<point_t>; // N-dimensional point
    typename locator;  where RandomAccessNDLocatorConcept<locator>; // N-dimensional locator.
    typename iterator; where RandomAccessIteratorConcept<iterator>; // 1-dimensional iterator over all
values
    typename reverse_iterator; where RandomAccessIteratorConcept<iterator>;
    typename size_type;       // the return value of size()

    // Equivalent to RandomAccessNDLocatorConcept::axis
    template <size_t D> struct axis {
        typename coord_t = point_t::axis<D>::coord_t;
        typename iterator; where RandomAccessIteratorConcept<iterator>;   // iterator along N-th axis.
        where coord_t = iterator::difference_type;
    };
```

```
    static const size_t num_dimensions = point_t::num_dimensions;

    // Create from a locator at the top-left corner and dimensions
    VIEW::VIEW(const locator&, const point_type&);

 // 1-dimensional operations
    size_type        VIEW::size()       const; // total number of elements
    reference        operator[](const difference_type&) const; // 1-dimensional reference
    iterator         VIEW::begin()      const;
    iterator         VIEW::end()        const;
    reverse_iterator VIEW::rbegin()     const;
    reverse_iterator VIEW::rend()       const;
    iterator         VIEW::at(const point_t&);

 // N-dimensional operations
    point_t          dimensions() const; // number of elements along each dimension

    // iterator along a given dimension starting at a given point
    template <size_t D> axis<D>::iterator axis_iterator(const point_t&) const;

    reference operator()(const point_t&) const;
};

template <typename VIEW>
concept MutableRandomAccessNDImageViewConcept : RandomAccessNDImageViewConcept<VIEW> {
    where Mutable<value_type>;
};
```

Two-dimensional image views have the following extra requirements:

```
template <typename VIEW>
concept RandomAccess2DImageViewConcept : RandomAccessNDImageViewConcept<VIEW> {
    typename x_iterator = axis<0>::iterator;
    typename y_iterator = axis<1>::iterator;
    typename x_coord_t  = axis<0>::coord_t;
    typename y_coord_t  = axis<1>::coord_t;
    typename xy_locator = locator;

    x_coord_t VIEW::width()  const;
    y_coord_t VIEW::height() const;

    // X-navigation
    x_iterator VIEW::x_at(const point_t&) const;
    x_iterator VIEW::row_begin(y_coord_t) const;
    x_iterator VIEW::row_end  (y_coord_t) const;

    // Y-navigation
    y_iterator VIEW::y_at(const point_t&) const;
    y_iterator VIEW::col_begin(x_coord_t) const;
    y_iterator VIEW::col_end  (x_coord_t) const;

    // navigating in 2D
    xy_locator VIEW::xy_at(const point_t&) const;

    // xy-versions of all methods taking point_type
    VIEW::VIEW(x_coord_t,y_coord_t,const locator&);
    iterator VIEW::at(x_coord_t,y_coord_t) const;
    reference operator()(x_coord_t,y_coord_t) const;
    xy_locator VIEW::xy_at(x_coord_t,y_coord_t) const;
    x_iterator VIEW::x_at(x_coord_t,y_coord_t) const;
    y_iterator VIEW::y_at(x_coord_t,y_coord_t) const;
};
```

```
template <RandomAccess2DImageViewConcept VIEW>
concept MutableRandomAccess2DImageViewConcept : MutableRandomAccessNDImageViewConcept<VIEW> {};
```

Finally, image views that GIL uses operate on value_types that model **`PixelConcept`** and have some additional requirements:

```
template <typename VIEW>
concept ImageViewConcept : RandomAccess2DImageViewConcept<VIEW> {
    where PixelValueConcept<value_type>;
    where PixelIteratorConcept<x_iterator>;
    where PixelIteratorConcept<y_iterator>;
    where x_coord_t == y_coord_t;

    typename pixel_t       = value_type;
    typename channel_t     = pixel_t::channel_t;
    typename color_space_t = pixel_t::color_space_t;
    typename coord_t       = x_coord_t;
    typename dynamic_step_t; where ImageViewConcept<dynamic_step_t>; where
StepIteratorConcept<dynamic_step_t::x_iterator>;

    // Create with a horizontal iterator at the top left corner, dimensions and number of bytes per row
    VIEW::VIEW(const point_t&,      const x_iterator&, ptrdiff_t row_bytes);
    VIEW::VIEW(x_coord_t,y_coord_t, const x_iterator&, ptrdiff_t row_bytes);

    ptrdiff_t VIEW::row_bytes()    const; // the number of bytes per row
    ptrdiff_t VIEW::pix_bytestep() const; // the number of bytes between two adjacent pixels on the same
row
};

template <typename VIEW>
concept MutableImageViewConcept : MutableRandomAccess2DImageViewConcept<VIEW>, ImageViewConcept<VIEW> {};
```

Related Concepts:

- **RandomAccessNDImageViewConcept**<V>
- **MutableRandomAccessNDImageViewConcept**<V>
- **RandomAccess2DImageViewConcept**<V>
- **MutableRandomAccess2DImageViewConcept**<V>
- **ImageViewConcept**<V>
- **MutableImageViewConcept**<V>

Implementation:

GIL provides a model for **`ImageViewConcept`**. It is templated over a model of **`PixelLocatorConcept`**. (If instantiated with a model of **`MutablePixelLocatorConcept`**, it models **`MutableImageViewConcept`**):

```
template <typename LOC>  // Models PixelLocatorConcept (or MutablePixelLocatorConcept)
class image_view {
public:
    typedef LOC xy_locator;
    typedef pixel_image_iterator<LOC> iterator;
    ...
private:
    xy_locator _pixels;     // 2D pixel locator at the top left corner of the image view range
    point_t    _dimensions; // width and height
};
```

Creating Views from Raw Pixels

Standard image views can be constructed from raw data of any supported color space, bit depth, channel ordering or planar vs. interleaved structure. Interleaved views are constructed using `interleaved_view`, supplying the image dimensions, number of bytes per row, and a pointer to the first pixel:

```
template <typename IT> // Models pixel iterator (like rgb8_ptr_t or rgb8c_ptr_t)
image_view<...> interleaved_view(ptrdiff_t width, ptrdiff_t height, IT pixels, ptrdiff_t rowsize_in_bytes)
```

Planar views are defined for every color space and take each plane separately. Here is the RGB one:

```
template <typename IC>  // Models channel iterator (like bits8* or const bits8*)
image_view<...> planar_rgb_view(ptrdiff_t width, ptrdiff_t height,
                                IC r, IC g, IC b, ptrdiff_t row_bytes);
```

Note that the supplied pixel/channel iterators could be `const`, in which case the returned view is a constant-value (immutable) view.

Creating Image Views from Other Image Views

It is possible to construct one image view from another by changing some policy of how image data is interpreted:

```
// flipped upside-down, left-to-right, transposed view
template <typename VIEW> VIEW                 flipped_up_down_view(const VIEW& src);
template <typename VIEW> VIEW::dynamic_step_t flipped_left_right_view(const VIEW& src);
template <typename VIEW> VIEW::dynamic_step_t transposed_view(const VIEW& src);

// rotations
template <typename VIEW> VIEW::dynamic_step_t rotated180_view(const VIEW& src);
template <typename VIEW> VIEW::dynamic_step_t rotated90cw_view(const VIEW& src);
template <typename VIEW> VIEW::dynamic_step_t rotated90ccw_view(const VIEW& src);

// view of an axis-aligned rectangular area within an image
template <typename VIEW> VIEW                 subimage_view(const VIEW& src,
            const VIEW::point_t& top_left, const VIEW::point_t& dimensions);

// subsampled view (skipping pixels in X and Y)
template <typename VIEW> VIEW::dynamic_step_t subsampled_view(const VIEW& src,
            const VIEW::point_t& step);

// color and bit depth converted view to match pixel type P
template <typename SRC_VIEW, // Models ImageViewConcept
          typename DST_P>    // Models PixelConcept
struct color_convert_types {
    typedef ... x_iterator; // dereference iterator adaptor that converts SRC_VIEW::value_type to DST_P
    typedef ... view_t;     // image view adaptor with value type DST_P, over SRC_VIEW
};
template <typename VIEW, typename P>
color_convert_types<VIEW,P>::view_t           color_converted_view(const VIEW& src);

// single-channel view of the N-th channel of a given view
template <typename SRC_VIEW>
struct nth_channel_types {
    typedef ... x_iterator; // iterator over a single-channel of the rows of SRC_VIEW
    typedef ... view_t;     // its corresponding image view type
};
template <typename VIEW>
nth_channel_types<VIEW>::view_t               nth_channel_view(const VIEW& view, int n);
```

The implementations of most of these view factory methods are straightforward. Here is, for example, how the flip views are implemented. The flip upside-down view creates an image whose first pixel is the bottom left pixel of the original image and whose

`row_bytes` is the negated `row_bytes` of the original:

```
// up-down flip
template <typename VIEW>
inline VIEW flipped_up_down_view(const VIEW& src) {
    gil_function_requires<ImageViewConcept<VIEW> >();  // VIEW must model ImageViewConcept
    return VIEW(src.dimensions(),src.row_begin(src.height()-1),-src.row_bytes());
}
```

The call to `gil_function_requires` ensures (at compile time) that the template parameter is a valid model of **`ImageViewConcept`**. Using it generates easier to track compile errors, creates no extra code and has no run-time performance impact. We are using the `boost::concept_check` library, but wrapping it in `gil_function_requires`, which performs the check if the `USE_GIL_CONCEPT_CHECK` is set. It is unset by default, because there is a significant increase in compile time when using concept checks. We will skip `gil_function_requires` in the code examples in this guide for the sake of succinctness.

Flipping left-to-right is a bit more complicated as we have to negate the unit step of the horizontal iterator (`pix_bytestep`), which requires that we use a step iterator. (If the original iterator was a step iterator, the two step iterators won't nest; `make_step_iterator` will return a single step iterator with the product of their steps.)

```
// left-to-right flip
template <typename VIEW>
inline typename VIEW::dynamic_step_t flipped_left_right_view(const VIEW& src) {
    return typename VIEW::dynamic_step_t(src.dimensions(),
          make_step_iterator(src.x_at(src.width()-1,0),-src.pix_bytestep()),src.row_bytes());
}
```

Image views can be freely composed (see section **13. Useful Metafunctions and Typedefs** for the typedefs `rgb16_image_t` and `gray16_step_view_t`):

```
rgb16_image_t img(100,100);    // an RGB interleaved image

// grayscale view over the green (index 1) channel of img
gray16_step_view_t green=nth_channel_view(view(img),1);

// 50x50 view of the green channel of img, upside down and taking every other pixel in X and in Y
gray16_step_view_t ud_fud=flipped_up_down_view(subsampled_view(green,2,2));
```

As previously stated, image views are fast, constant-time, shallow views over the pixel data. The above code does not copy any pixels; it operates on the pixel data allocated when `img` was created.

---

10. Image

An image is a container that owns the pixels of a given image view. It allocates them in its constructor and deletes them in the destructor. It has a deep assignment operator and copy constructor. Images are used rarely, just when data ownership is important. STL algorithms operate on ranges, not containers. Similarly GIL algorithms operate on image views (which images provide).

In the most general form images are N-dimensional and satisfy the following pseudo-interface:

```
template <boost::RandomAccessContainerConcept IMG>
concept RandomAccessNDImageConcept : DefaultConstructible<IMG> {
    typename const_view_t; where RandomAccessNDImageViewConcept<const_view_t>;
    typename point_t = const_view_t::point_t;

    IMG::IMG(const point_t&, int);   // construct with dimensions and alignment
```

```
    void resize_clobber_image(IMAGE& img, const IMG::point_t& new_size) {
        IMG tmp(new_size); swap(tmp,img);
    }

    const point_t&        get_dimensions(const IMG&);
    const const_view_t&   const_view(const IMG&);
};

template <boost::Mutable_RandomAccessContainerConcept IMG>
concept MutableRandomAccessNDImageConcept : RandomAccessNDImageConcept<IMG> {
    typename view_t; where MutableRandomAccessNDImageViewConcept<view_t>;

    const view_t& view(IMG&);
};
```

Note that `boost::RandomAccessContainerConcept` requires lots of typedefs (`value_type`, `difference_type`, etc.) and methods (`begin()`, `end()`, `rbegin()`, `size()`, `operator[]`, etc) that are not explicitly listed here.

Two-dimensional images have additional requirements:

```
template <typename IMG>
concept RandomAccess2DImageConcept : RandomAccessNDImageConcept<IMG> {
    typename x_coord_t = const_view_t::x_coord_t;
    typename y_coord_t = const_view_t::y_coord_t;

    x_coord_t get_width(const IMG&);
    y_coord_t get_height(const IMG&) const;

    void resize_clobber_image(IMAGE& img, const IMG::x_coord_t& width, const IMG::y_coord_t& height) {
        IMG tmp(width,height); swap(tmp,img);
    }
};

template <MutableRandomAccessNDImageConcept IMG>
concept MutableRandomAccess2DImageConcept : RandomAccess2DImageConcept<IMG> {};
```

GIL's images have views that model **`ImageViewConcept`**:

```
template <typename IMG>
concept ImageConcept : RandomAccess2DImageConcept<IMG> {
    where ImageViewConcept<const_view_t>;
};

template <MutableRandomAccess2DImageConcept IMG>
concept MutableImageConcept : ImageConcept<IMG> {
    where MutableImageViewConcept<view_t>;
};
```

Related Concepts:

- **RandomAccessNDImageConcept**<IMAGE>
- **MutableRandomAccessNDImageConcept**<IMAGE>
- **RandomAccess2DImageConcept**<IMAGE>
- **MutableRandomAccess2DImageConcept**<IMAGE>
- **ImageConcept**<IMAGE>
- **MutableImageConcept**<IMAGE>

Implementation:

GIL provides a class, `image`, templated over an image view, which models **`ImageConcept`** when instantiated with a model of **`ImageViewConcept`** and **`MutableImageConcept`** when instantiated with a model of **`MutableImageViewConcept`**.

```
template <typename V, \\ Models ImageViewConcept
          typename A=std::allocator<unsigned char> >
class image;
```

By default images have 1 byte (no) alignment - i.e. there is no padding at the end of rows. Many operations are faster using such images (because **`image_view::x_iterator`** can be used to traverse the pixels, instead of the more complicated **`image_view::iterator`**). Images can also be constructed from existing image views with special allocators that allow them to take over the ownership. This allows us to integrate images with 3rd party reference-counting mechanism.

---

11. Run-time specified images and image views

The color space, channel depth, channel ordering, and interleaved/planar structure of an image are defined by the type of its template argument, which makes them compile-time bound. Often some of these parameters are available only at run time. Consider, for example, writing a module that opens the image at a given file path, rotates it and saves it back in its original color space and channel depth. How can we possibly write this using our generic image? What type is the image loading code supposed to return?

GIL's dynamic_image extension allows for images, image views or any GIL constructs to have their parameters defined at run time. Here is an example:

```
#include <gil/extension/dynamic_image/any_image_factory.hpp>
#include <gil/extension/dynamic_image/any_image.hpp>
using namespace boost;

#define ASSERT_SAME(A,B) BOOST_STATIC_ASSERT((is_same< A,B >::value))

// Define the set of allowed images
typedef mpl::vector<rgb8_image_t, cmyk16_planar_image_t> my_images_t;

// Create any_image class (or any_image_view) class
typedef any_image<my_images_t> my_any_image_t;

// Associated view types are available (equivalent to the ones in image_t)
typedef any_image_view<mpl::vector2<rgb8_view_t,  cmyk16_planar_view_t > > AV;
ASSERT_SAME(my_any_image_t::view_t, AV);

typedef any_image_view<mpl::vector2<rgb8c_view_t, cmyk16c_planar_view_t> > CAV;
ASSERT_SAME(my_any_image_t::const_view_t, CAV);
ASSERT_SAME(my_any_image_t::const_view_t, my_any_image_t::view_t::const_t);

typedef any_image_view<mpl::vector2<rgb8_step_view_t, cmyk16_planar_step_view_t> > SAV;
ASSERT_SAME(my_any_image_t::view_t::dynamic_step_t, SAV);

// Assign it a concrete image at run time:
my_any_image_t myImg = my_any_image_t(rgb8_image_t(100,100));

// Change it to another at run time. The previous image gets destroyed
myImg = cmyk16_planar_image_t(200,100);

// Assigning to an image not in the allowed set throws an exception
myImg = gray8_image_t();        // will throw std::bad_cast
```

Explicit enumeration of all possible types is one way to define a GIL run-time instantiated object. Images and image views can also be defined implicitly - by defining the sets of allowed color spaces, channels, interleaved/planar and step/nonstep policy:

```
// Create [gray8_image_t, gray16_image_t, bgr8_image_t, bgr16_image_t]
typedef cross_vector_image_types
    < mpl::vector<bits8*, bits16*>,
      mpl::vector<gray_t, bgr_t>,
      kInterleavedOnly
    >::type my_images_t;
typedef any_image<my_images_t> my_any_image_t;

// Create [lab32c_view_t, lab32c_step_view_t, lab32c_planar_view_t, lab32c_planar_step_view_t]
typedef cross_vector_image_view_types
    < mpl::vector<const bits32*>,
      mpl::vector<lab_t>,
      kInterleavedAndPlanar,
      kNonStepAndStep
    >::type my_views_t;
typedef any_image_view<my_any_view_t> my_any_image_view_t;
```

Using GIL's `concat_vector` (which lives in mpl namespace), it is also possible to combine type sequences:

```
// Create [gray8_image_t, gray16_image_t, bgr8_image_t, bgr16_image_t, lab8_image_t, lab16_image_t]
typedef mpl::vector<lab8_image_t, lab16_image_t> extra_images_t;
typedef mpl::concat_vector<my_images_t, extra_images_t>::type my_extended_images_t;
typedef any_image<my_extended_images_t> my_extended_any_image_t;
```

Type sequence concatenation could be nested and result in complex unions of implicitly and explicitly defined types.

`any_image` and **`any_image_view`** subclass from GIL's `variant` class, which breaks down the instantiated type into a non-templated underlying base type and a unique instantiation type identifier. The underlying base instance is represented as a block of bytes. The block is large enough to hold the largest of the specified types.

GIL's variant is similar to `boost::variant` in spirit (hence we borrow the name from there) but it differs in several ways from the current boost implementation. Perhaps the biggest difference is that GIL's variant always takes a single argument, which is a model of MPL Random Access Sequence enumerating the allowed types. Having a single interface allows GIL's variant to be used easier in generic code. Synopsis:

```
template <typename TYPES>   // models MPL Random Access Container
class variant {
    ...           _bits;
    std::size_t   _index;
public:
    typedef TYPES   types_t;

    variant();
    variant(const variant& v);
    virtual ~variant();

    variant& operator=(const variant& v);
    bool operator==(const variant&) const;
    bool operator!=(const variant&) const;

    // Construct/assign to type T. Throws std::bad_cast if T is not in TYPES
    template <typename T> explicit variant(const T& obj);
    template <typename T> variant& operator=(const T& obj);

    // Construct/assign by swapping T with its current instance. Only possible if they are swappable
    template <typename T> explicit variant(T& obj, bool do_swap);
    template <typename T> void move_in(T& obj);

    template <typename T> static bool has_type();

    template <typename T> const T& _dynamic_cast() const;
```

```
    template <typename T>       T& _dynamic_cast();
};

template <typename UOP, typename TYPES>
   UOP::result_type apply_operation(variant<TYPES>& v, UOP op);
template <typename UOP, typename TYPES>
   UOP::result_type apply_operation(const variant<TYPES>& v, UOP op);

template <typename BOP, typename TYPES1, typename TYPES2>
   BOP::result_type apply_operation(      variant<TYPES1>& v1,       variant<TYPES2>& v2, UOP op);

template <typename BOP, typename TYPES1, typename TYPES2>
   BOP::result_type apply_operation(const variant<TYPES1>& v1,       variant<TYPES2>& v2, UOP op);

template <typename BOP, typename TYPES1, typename TYPES2>
   BOP::result_type apply_operation(const variant<TYPES1>& v1, const variant<TYPES2>& v2, UOP op);

template <typename VTYPES>  // Models MPL Random Access Sequence of image view types
class any_image_view : public variant<VTYPES> {
public:
    typedef ... const_t;        // == any_image_view of the immutable versions of VTYPES
    typedef ... dynamic_step_t; // == any_image_view of the step versions of VTYPES

    any_image_view();
    any_image_view(const any_image_view& v);
    any_image_view& operator=(const any_image_view& v);

    template <typename T> explicit any_image_view(const T& obj);
    template <typename T> any_image_view& operator=(const T& obj);
};

template <typename ITYPES> // Models MPL Random Access Sequence of image types
class any_image : public variant<ITYPES> {
public:
    typedef ... view_t;       // == any_image_view of the corresponding views of ITYPES
    typedef ... const_view_t; // == any_image_view of the corresponding const views of ITYPES

    any_image();
    any_image(const any_image& v);
    any_image& operator=(const any_image& v);

    template <typename T> explicit any_image(const T& obj);
    template <typename T> any_image& operator=(const T& obj);

    template <typename T> explicit any_image(T& obj, bool doSwap);
};
```

Variants behave like the underlying type. Their copy constructor will invoke the copy constructor of the underlying instance. Equality operator will check if the two instances are of the same type and then invoke their operator==, etc. The default constructor of a variant will default-construct the first type.

Operations are invoked on variants via `apply_operation` passing a function object to perform the operation. The code for every allowed type in the variant is instantiated and the appropriate instantiation is selected via a switch statement. Since image view algorithms typically have time complexity at least linear on the number of pixels, the single switch statement of image view variant adds practically no measurable performance overhead compared to templated image views. Note that run-time instantiated images and image views do not provide ways to access their pixels. There is no "any_pixel" or "any_pixel_iterator" in GIL. Such constructs could be provided via the `variant` mechanism, but doing so would result in inefficient algorithms, since the type resolution would have to be performed per pixel. Image-level algorithms should be implemented via `apply_operation`.

Most GIL algorithms have versions that can take image view variants in place of templated views:

```
rgb8_view_t v1(...);  // concrete image view
bgr8_view_t v2(...);  // concrete image view compatible with v1 and of the same size
any_image_view<TYPES>  av(...);  // run-time specified image view

// Copies the pixels from v1 into v2.
// If the pixels are incompatible triggers compile error
copy_pixels(v1,v2);

// The source or destination (or both) may be run-time instantiated.
// If they happen to be incompatible, throws std::bad_cast
copy_pixels(v1, av);
copy_pixels(av, v2);
copy_pixels(av, av);
```

GIL's I/O extension (not yet released) makes extensive use of run-time specified images. Here is how to make a 180-degree rotated copy of an image saved on disk, preserving its native color space and channel depth (the file formats are inferred automatically from the image extensions):

```
#include <gil\extension\dynamic_image\image_view_factory.hpp>
#include <gil\extension\io\image_io.hpp>

template <typename IMG_TYPES>
void save_180rot(const std::string& fileNameIn, const std::string& fileNameOut) {
    any_image<IMG_TYPES> img;
    load_image(img, fileNameIn);
    save_view(rotated180_view(view(img)), fileNameOut);
}
```

The above example works because the methods `view`, `rotated180_view` and `save_view` have overloaded versions that take image view variants. For example, here is how `rotated180_view` is implemented:

```
// implementation using templated view
template <typename VIEW>
typename VIEW::dynamic_step_t rotated180_view(const VIEW& src) { ... }

namespace detail {
    // the method, wrapped inside a function object
    template <typename RESULT> struct rotated180_view_fn {
        typedef RESULT result_type;
        template <typename VIEW> result_type operator()(const VIEW& src) const {
            return result_type(rotated180_view(src));
        }
    };
}

// overloading of the method using variant. Takes and returns run-time bound view.
// The returned view has a dynamic step
template <typename VTYPES> inline // Models ImageViewVectorConcept
typename any_image_view<VTYPES>::dynamic_step_t rotated180_view(const any_image_view<VTYPES>& src) {
    typedef typename any_image_view<VTYPES>::dynamic_step_t result_t;
    return apply_operation(src,detail::rotated180_view_fn<result_t>());
}
```

Variants should be used with caution (especially algorithms that take more than one variant) because they instantiate the algorithm for every possible model that the variant can take. This can take a toll on compile time and executable size. GIL uses algorithm-centric strategy for reducing code bloat, but it is still in early stages of development. Despite these limitations, `variant` is a powerful technique that allows us to combine the speed of compile-time resolution with the flexibility of run-time resolution. It allows us to treat images of different parameters uniformly as a collection and store them in the same container.

---

12. Algorithms

GIL overrides some STL algorithms with faster versions when used with pixel iterators. For example, **`std::copy`**:

- when used with raw pixel pointers (interleaved) of the same type, resolves to `memmove`.
- when used with planar pointers of the same type, resolves to `memmove` for each color plane
- when a source or destination range is delimited by `pixel_image_iterator-s` and their image happens to have no padding at the end of rows, their base x-iterators are used instead (which, in turn may resolve to `memmove`).
- if the image of **`pixel_image_iterator`** has padding, **`std::copy`** traverses each row using the base x-iterators along the row (which may resolve to `memmove` for each row).

Similarly, **`std::fill`** attempts to reduce **`pixel_image_iterator`**-s to their base iterators and will sometimes resolve to `memset`.

GIL provides versions of some standard STL algorithms that operate on image views and on variants of image views:

```
// Equivalent of std::copy
// V1 models ImageViewConcept; V2 Models MutableImageViewConcept.
// V1::value_type and V2::value_type must be compatible.
// V1 and V2 must have the same dimensions
template <typename V1, typename V2>
void copy_pixels(const V1& src, const V2& dst);
template <typename C1, typename V2>
void copy_pixels(const any_image_view<C1>& src, const V2& dst);
template <typename V1, typename C2>
void copy_pixels(const V1& src, const any_image_view<C2>& dst);
template <typename C1, typename C2>
void copy_pixels(const any_image_view<C1>& src, const any_image_view<C2>& dst);

// Equivalent of std::fill
// V Models MutableImageViewConcept. VAL models PixelConcept and is compatible with V::value_type
template <typename V, typename VAL>
void fill_pixels(const V& img_view, const VAL& val);
template <typename C, typename VAL>
void fill_pixels(const any_image_view<C>& img_view, const VAL& val);

// Equivalent of std::for_each
// V models ImageViewConcept and F models boost::UnaryFunctionConcept.
// V::reference must be compatible with F::argument_type
template <typename V, typename F>
F for_each_pixel(const V& img, F fun);

// Equivalent of std::transform
// V1 models ImageViewConcept; V2 Models MutableImageViewConcept.
// V1::value_type and V2::value_type must be compatible.
// V1 and V2 must have the same dimensions
// F models boost::UnaryFunctionConcept.
// V1::const_reference must be compatible with F::argument_type and
// V2::reference must be compatible with F::result_type
template <typename V1, typename V2, typename F>
F transform_pixels(const V1& src, const V2& dst, F fun);
template <typename V1, typename V2, typename F>
F transform_pixels(const V1& src1, const V1& src2, const V2& dst, F fun);

// Copies a view into another, color converting the pixels if necessary
// V1 models ImageViewConcept; V2 Models MutableImageViewConcept.
// V1::value_type must be convertible to V2::value_type.
// V1 and V2 must have the same dimensions
template <typename V1, typename V2>
void copy_and_convert_pixels(const V1& src, const V2& dst);
template <typename C1, typename V2>
void copy_and_convert_pixels(const any_image_view<C1>& src, const V2& dst);
template <typename V1, typename C2>
```

```
void copy_and_convert_pixels(const V1& src, const any_image_view<C2>& dst);
template <typename C1, typename C2>
void copy_and_convert_pixels(const any_image_view<C1>& src, const any_image_view<C2>& dst);

// Equivalent of std::equal
// V1 and V2 model ImageViewConcept
// V1::value_type and V2::value_type must be compatible.
// V1 and V2 must have the same dimensions
template <typename V1, typename V2>
bool equal_pixels(const V1& src, const V2& dst);
template <typename C1, typename V2>
bool equal_pixels(const any_image_view<C1>& src, const V2& dst);
template <typename V1, typename C2>
bool equal_pixels(const V1& src, const any_image_view<C2>& dst);
template <typename C1, typename C2>
bool equal_pixels(const any_image_view<C1>& src, const any_image_view<C2>& dst);
```

---

13. Useful Metafunctions and Typedefs

Flexibility comes at a price. GIL types can be very long and hard to read. Here is, for example, the type of a simple 8-bit grayscale image, perhaps the simplest possible image:

```
typedef image< image_view< pixel_2d_locator< pixel_step_iterator< pixel< bits8,gray_t >* > > >
             , std::allocator<unsigned char> > gray8_image_t;
```

To address this problem, GIL provides typedefs to refer to any standard image, pixel iterator, pixel locator, pixel reference or pixel value. They follow this pattern:

ColorSpace + *BitDepth* + ("c") + ("_planar") + ("_step") + *ClassType* + "_t"

Where *ColorSpace* also indicates the ordering of components. Examples are `rgb`, `bgr`, `cmyk`, `rgba`. *BitDepth* can be, for example, `8`,`16`,`32`. `c` indicates object whose `value_type` is immutable. `_planar` indicates planar organization (as opposed to interleaved). `_step` indicates the type has a dynamic step and *ClassType* is `_image` (image, using a standard allocator), `_view` (image view), `_loc` (pixel locator), `_ptr` (pixel iterator), `_ref` (pixel reference), `_pixel` (pixel value). Here are examples:

```
bgr8_image_t              i;     // 8-bit interleaved BGR image
cmyk16_pixel_t;           x;     // 16-bit CMYK pixel value;
cmyk16c_planar_ref_t      p(x);  // const reference to a 16-bit planar CMYK pixel x.
rgb32_planar_step_ptr_t   ii;    // step iterator to a 32-bit planar RGB pixel.
```

GIL provides the following metafunctions that return the types of standard GIL constructs:

```
// Returns the type of an iterator based on the types of its channel and color space,
// whether it needs to be planar/interleaved, and whether it requires a dynamic step
template <typename IC,  // Models iterator to ChannelConcept
          typename C,   // Models ColorSpaceTypeConcept
          bool IS_PLANAR=false, bool IS_STEP=false>
struct iterator_type {
    typedef ... type;  // Models PixelIteratorConcept
};

// Returns the type of the standard step iterator, locator and view
// corresponding to a given horizontal pixel iterator:
template <typename X_IT> // Models PixelIteratorConcept
struct type_from_x_iterator {
    typedef pixel_step_iterator<X_IT>         step_iterator_t;
```

```
    typedef pixel_2d_locator<step_iterator_t> xy_locator_t;
    typedef image_view<xy_locator_t>          view_t;
};

// Returns the step iterator, locator, view and image based on the types of its channel iterator and
color space,
// whether it needs to be planar/interleaved, and whether it requires a dynamic step
template <typename IC, typename C, bool IS_PLANAR=false, bool IS_STEP=false>
struct step_iterator_type {
    typedef ... type; // models StepIteratorConcept
};
template <typename IC, typename C, bool IS_PLANAR=false, bool IS_STEP=false>
struct pixel_2d_locator_type {
    typedef ... type; // models PixelLocatorConcept
};
template <typename IC, typename C, bool IS_PLANAR=false, bool IS_STEP=false>
struct view_type {
    typedef ... type; // models ImageViewConcept
};
// (Note that images cannot have a step)
template <typename IC, typename C, bool IS_PLANAR=false, typename ALLOC=std::allocator<unsigned char> >
struct image_type {
    typedef ... type; // models ImageConcept
};

// Returns whether a given iterator has a step that could be set dynamically
template <typename I>  // Models PixelIteratorConcept
struct has_dynamic_step {
    static const bool value=...;// true if I models PixelStepIteratorConcept
};
```

---

14. Sample Code

Pixel-level Sample Code

Here are some operations you can do with pixel values, pointers and references:

```
rgb8_pixel_t p1(255,0,0);       // make a red RGB pixel
bgr8_pixel_t p2 = p1;           // RGB and BGR are compatible and the channels will be properly mapped.
assert(p1==p2);               // p2 will also be red.
assert(p2[0]!=p1[0]);         // operator[] gives physical channel order (as laid down in memory)
assert(p1.v<0>()==p2.v<0>()); // this is how to compare the two red channels
p1.g = p2.b;                  // channels can also be accessed by name
p1 += p2*3;                   // channelwise operations treat pixels as vectors of channels (paired
semantically)

const unsigned char* r=...;
const unsigned char* g=...;
const unsigned char* b=...;
rgb8c_planar_ptr_t ptr(r,g,b); // constructing const planar pointer from const pointers to each plane

rgb8c_planar_ref_t ref=*ptr;   // just like built-in reference, dereferencing a planar pointer returns a
planar reference

p2 = ref+p1-ptr[7]+rgb8_pixel_t(1,2,3);    // planar/interleaved references and values to RGB/BGR can be
freely mixed

rgb8_planar_ref_t ref2;        // compile error: References have no default constructors
ref2=*ptr;                   // compile error: Cannot construct non-const reference by dereferencing
const pointer
```

```
ptr[3]=p1;                  // compile error: Cannot set the fourth pixel through a const pointer
p1 + pixel<float, rgb_tag>();// compile error: Incompatible channel depth
p1 = pixel<bits8, lab_tag>();// compile error: Incompatible color space (even though it has the same
number of channels)
p1== pixel<bits8,rgba_tag>();// compile error: Incompatible color space (even though it contains red,
green and blue channels)
```

Here is how to use pixels in generic code:

```
template <typename GRAY, typename RGB>
void gray_to_rgb(const GRAY& src, RGB& dst) {
    gil_function_requires<PixelConcept<GRAY> >();
    BOOST_STATIC_ASSERT((boost::is_same<typename GRAY::color_space_t::base,gray_t>::value));

    gil_function_requires<MutablePixelConcept<RGB> >();
    BOOST_STATIC_ASSERT((boost::is_same<typename RGB::color_space_t::base,rgb_t>::value));

    dst.r=dst.g=dst.b=channel_convert<typename RGB::channel_value_t>(src.gray);
}

// example use patterns:

// converting gray l-value to RGB and storing at (5,5) in a 16-bit BGR interleaved image:
bgr16_view_t b16(...);
gray_to_rgb(gray8_pixel_t(33), b16(5,5));

// storing the first pixel of an 8-bit grayscale image as the 5-th pixel of 32-bit planar RGB image:
rgb32_planar_view_t rpv32(...);
gray8_view_t gv8(...);
gray_to_rgb(*gv8.begin(), rpv32[5]);
```

As the example shows, both the source and the destination can be references or values, planar or interleaved, as long as they model **`PixelConcept`** and **`MutablePixelConcept`** respectively.

Creating a Copy of an Image with a Safe Buffer

Suppose we want to convolve an image with multiple kernels, the largest of which is 2K+1 x 2K+1 pixels. It may be worth creating a margin of K pixels around the image borders. Here is how to do it:

```
template <typename SRC_VIEW, // Models ImageViewConcept (the source view)
          typename DST_IMG>  // Models ImageConcept     (the returned image)
void create_with_margin(const SRC_VIEW& src, int k, DST_IMG& result) {
    gil_function_requires<ImageViewConcept<SRC_VIEW> >();
    gil_function_requires<ImageConcept<DST_IMG> >();

    result=DST_IMG(src.width()+2*k, src.height()+2*k);
    SRC_VIEW centerImg=subimage_view(view(result), k,k,src.width(),src.height());
    std::copy(src.begin(), src.end(), centerImg.begin());
}
```

We allocated a larger image, then we used `subimage_view` to create a shallow image of its center area of top left corner at (k,k) and of identical size as `src`, and finally we copied `src` into that center image. If the margin needs initialization, we could have done it with `fill_pixels`. Here is how to simplify this code using the `copy_pixels` algorithm:

```
template <typename SRC_VIEW, typename DST_IMG>
void create_with_margin(const SRC_VIEW& src, int k, DST_IMG& result) {
    resize_clobber_image(result, src.width()+2*k, src.height()+2*k);
    copy_pixels(src, subimage_view(view(result), k,k,src.width(),src.height()));
}
```

(Note also that `resize_clobber_image` is more efficient than `operator=`, as it uses **`std::swap`**). Not only does the above example work for planar and interleaved images of any color space and pixel depth; it is also quite fast. GIL overrides **`std::copy`** - when called on two identical interleaved images with no padding at the end of rows, it simply does a `memmove`. For planar images it does `memmove` for each channel. If one of the images has padding, (as in our case) it will try to do `memmove` for each row. When an image has no padding, it will use its lightweight horizontal iterator (as opposed to the more complex 1D image iterator that has to check for the end of rows). It choses the fastest method, taking into account both static and run-time parameters.

Histogram

The histogram can be computed by counting the number of pixel values that fall in each bin. The following method takes a grayscale (one-dimensional) image view, since only grayscale pixels are convertible to integers:

```
template <typename GRAY_VIEW, typename R>
void grayimage_histogram(const GRAY_VIEW& img, R& hist) {
    for (typename GRAY_VIEW::iterator it=img.begin(); it!=img.end(); ++it)
        ++hist[*it];
}
```

Using `boost::lambda` and GIL's `for_each_pixel` algorithm, we can write this more compactly:

```
template <typename GRAY_VIEW, typename R>
void grayimage_histogram(const GRAY_VIEW& img, R& hist) {
    for_each_pixel(img, ++var(hist)[_1]);
}
```

Where `for_each_pixel` invokes `std::for_each` and `var` and `_1` are `boost::lambda` constructs. To compute the luminosity histogram, we call the above method using the grayscale view of an image:

```
template <typename VIEW, typename R>
void luminosity_histogram(const VIEW& img, R& hist) {
    grayimage_histogram(color_converted_view<gray8_pixel_t>(img),hist);
}
```

This is how to invoke it:

```
unsigned char hist[255];
std::fill(hist,hist+255,0);
luminosity_histogram(img,hist);
```

If we want to view the histogram of the second channel of the image in the top left 100x100 area, we call:

```
grayimage_histogram(nth_channel_view(subimage_view(img,0,0,100,100),1),hist);
```

No pixels are copied and no extra memory is allocated - the code operates directly on the source pixels, which could be in any supported color space and channel depth. They could be either planar or interleaved.

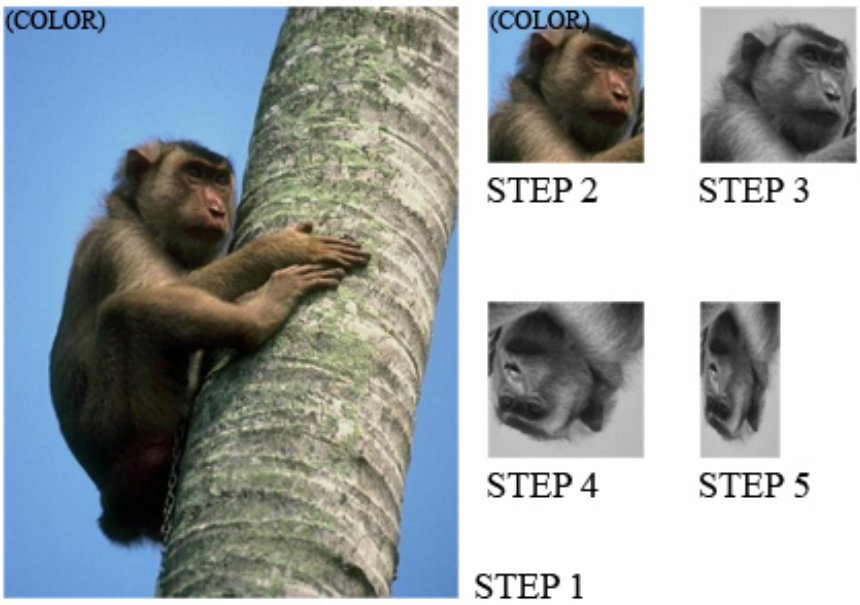
Using Image Views

The following code illustrates the power of using image views:

```
load_jpeg_image(img, "monkey.jpg");
step1=view(img);
step2=subimage_view(step1, 200,300, 150,150);
step3=color_converted_view<rgb8_view_t,gray8_pixel_t>(step2);
step4=rotated180_view(step3);
step5=subsampled_view(step4, 2,1);
```

```
save_jpeg_image(step5, "monkey_transform.jpg");
```

The intermediate images are shown here:



Notice that no pixels are ever copied. All the work is done inside `save_jpeg_image`. If we call our `luminosity_histogram` with `step5` it will do the right thing.

---

15. Extending the Generic Image Library

You can define your own pixel iterators, locators, image views, images, channel types, color spaces and algorithms. You can make virtual images that live on the disk, inside a jpeg file, somewhere on the internet, or even fully-synthetic images such as the Mandelbrot set. As long as they properly model the corresponding concepts, they will work with any existing GIL code. Most such extensions require no changes to the library and can thus be supplied in another module.

Defining New Color Spaces

Each color space is in a separate file. To add a new color space, just copy one of the existing ones (like **rgb.hpp**) and change it accordingly. If you want color conversion support, you will have to provide methods to convert between it and the existing color spaces (see color_convert.h). For convenience you may want to provide useful typedefs for pixels, pointers, references and images with the new color space (see typedefs.h).

Defining New Channel Types

Most of the time you don't need to do anything special to use a new channel type. You can just use it:

```
typedef pixel<double,rgb_tag>   rgb64_pixel;    // 64 bit RGB pixel
typedef rgb64_pixel*            rgb64_pixel_ptr;// pointer to 64-bit interleaved data
typedef image_type<double,rgb_tag>::type rgb64_image;    // 64-bit interleaved image
```

If you want to use your own channel class, you will need to provide a specialization of **`channel_traits`** for it (see **channel.hpp**). If you want to do color conversion, you will need to provide `channel_convert`, `channel_maxval` and `channel_multiply` methods for your channel type (see **color_convert.hpp**).

Defining New Pixel Iterators and Images

You can provide your own pixel iterators, overriding either the mechanism for getting from one pixel to the next or doing an arbitrary pixel transformation on dereference. For example, let's look at the implementation of `color_converted_view` (an image factory method that, given any image view, returns a new, otherwise identical view, except that color conversion is performed on pixel access). First we need to define a function object that will be called when we dereference a pixel iterator. It will dereference the base iterator, color-convert the pixel and return it:

```
template <typename DST_P>  // Models PixelConcept
struct color_convert_deref_fn {
    typedef DST_P result_type;

    template <typename I> result_type operator()(const I& srcIt) const {
        result_type dst;
        color_convert(*srcIt,dst);
        return dst;
    }
};
```

We then use GIL's `dereference_iterator_adaptor<I,FN>` - this is a class that wraps an existing iterator `I` and invokes a specified function object `FN` when the iterator is dereferenced. Putting the two together, we can define:

```
template <typename SRC_VIEW, typename DST_P>
struct color_convert_types {
    typedef dereference_iterator_adaptor<SRC_VIEW::x_iterator, color_convert_deref_fn<DST_P> > x_iterator;
    typedef type_from_x_iterator<x_iterator>::view_t                                           view_t;
};
```

If `I` is an iterator, `color_convert_types<I,rgb8_pixel_t>`, for example, provides the type of an iterator just like `I`, except that it looks through 8-bit RGB glasses and its corresponding view. Finally our `color_converted_view` code simply copy-constructs a color-converted view from the source view (copy construction from a compatible view is allowed):

```
template <typename DST_P, typename VIEW> inline
typename color_convert_types<VIEW,DST_P>::view_type color_convert_view(const VIEW& src) {
    return typename color_convert_types<VIEW,DST_P>::view_type(src);
}
```

---

16. Conclusion

The Generic Image Library is designed with the following five goals in mind:

- **Generality.** Abstracts image representations from algorithms on images. It allows for writing code once and have it work for any image type.
- **Performance.** Speed has been instrumental to the design of the library. The generic algorithms provided in the library are comparable in speed to hand-coding the algorithm for a specific image type.
- **Flexibility.** Compile-type parameter resolution results in faster code, but severely limits code flexibility. The library allows for any image parameter to be specified at run time, at a minor performance cost.
- **Extensibility.** It is easy to extend the library with new color spaces and image types. Library extensions typically require no changes to the library core; at the same time existing image algorithms apply to new color spaces and image types.
- **Compatibility.** The library is designed as an STL and Boost complement. Generic STL algorithms can be used for pixel manipulation, and they are specifically targeted for optimization. The library works with existing raw pixel data from another image library and can even be integrated into a third-party reference counting mechanism.

---

17. Acknowledgements

- **Jon Brandt**, **Mark Ruzon** and **Paul McJones** spent significant time reviewing the library and documentation and provided valuable feedback.
- **Alex Stepanov**'s class has directly inspired this project. Alex's "start from the inside" and "bottom up" principles have been applied throughout the design.
- **Sean Parent** and Alex Stepanov suggested splitting the image into a 'container' and 'range' classes
- **Bjarne Stroustrup** reviewed the GIL design