# Generic Image Library Tutorial

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

**Version:**
1.0

**Date:**
May 25, 2006

The Generic Image Library (GIL) is a C++ library that abstracts image representations from algorithms and allows writing code that can work on a variety of images with performance similar to hand-writing for a specific image type.

This document will give you a jump-start in using GIL. It does not discuss the underlying design of the library and does not cover all aspects of it. You can find a detailed library design document on the main GIL web page at http://opensource.adobe.com/gil

## Installation

GIL is part of Adobe Software Libraries (ASL) which can be installed at http://opensource.adobe.com. GIL consists of header files only and can be used as stand-alone also. Define `GIL_NO_ASL` if you want to use GIL without ASL. GIL's only dependency is boost. (http://www.boost.org) There is no library to link against. Including gil/core/gil_all.hpp will be sufficient for most projects.

## Example - Computing the Image Gradient

This tutorial will walk through an example of using GIL to compute the image gradients. We will start with some very simple and non-generic code and make it more generic as we go along. Let us start with a horizontal gradient and use the simplest possible approximation to a gradient - central difference. The gradient at pixel x can be approximated with the half-difference of its two neighboring pixels: D[x] = (I[x-1] - I[x+1]) / 2

For simplicity, we will also ignore the boundary cases - the pixels along the edges of the image for which one of the neighbors is not defined. The focus of this document is how to use GIL, not how to create a good gradient generation algorithm.

### Interface and Glue Code

Let us first start with 8-bit unsigned grayscale image as the input and 8-bit signed grayscale image as the output. Here is how the interface to our algorithm looks like:

```
#include <gil/core/gil_all.hpp>
using namespace GIL;

void x_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    assert(src.dimensions() == dst.dimensions());
    ...    // compute the gradient
}
```

`gray8c_view_t` is the type of the source image view - an 8-bit grayscale view, whose pixels are read-only (denoted by the `"c"`). The output is a grayscale view with a 8-bit signed (denoted by the `"s"`) integer channel type. See Appendix 1 for the complete convension GIL uses to name concrete types.

GIL makes a distinction between an image and an image view. A GIL *image view*, is a shallow, lightweight view of a rectangular grid of pixels. It provides access to the pixels but does not own the pixels. Copy-constructing a view does not deep-copy the pixels. Image views do not propagate their constness to the pixels and should always be taken by a const reference. Whether a view is mutable or read-only (immutable) is a property of the view type.

A GIL *image*, on the other hand, is a view with associated ownership. It is a container of pixels; its constructor/destructor allocates/deallocates the pixels, its copy-constructor performs deep-copy of the pixels and its operator== performs deep-compare of the pixels. Images also propagate their constness to their pixels - a constant reference to an image will not allow modifying its pixels.

Most GIL algorithms operate on image views; images are rarely needed. GIL's design is very similar to that of STL. The STL equivalent of GIL's image is a container, like `std::vector`, whereas GIL's image view corresponds to STL's range, which is often represented with a pair of iterators. STL algorithms operate on ranges, just like GIL algorithms operate on image views.

GIL's image views can be constructed from raw data - the dimensions, the number of bytes per row and the pixels, which for chunky views are represented with one pointer. Here is how to provide the glue between your code and GIL:

```
void ComputeXGradientGray8(const unsigned char* src_pixels, ptrdiff_t src_row_bytes, int w, int h,
                                  signed short* dst_pixels, ptrdiff_t dst_row_bytes) {
    gray8c_view_t src = interleaved_view(w, h, (const gray8_pixel_t*)src_pixels,src_row_bytes);
    gray8s_view_t dst = interleaved_view(w, h, (      gray8s_pixel_t*)dst_pixels,dst_row_bytes);
    x_gradient(src,dst);
}
```

This glue code is very fast and views are lightweight - in the above example the views have a size of 16 bytes. They consist of a pointer to the top left pixel and three integers - the width, height, and number of bytes per row.

## First Implementation

Focusing on simplicity at the expense of speed, we can compute the horizontal gradient like this:

```
void x_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    for (int y=0; y<src.height(); ++y)
        for (int x=1; x<src.width()-1; ++x)
            dst(x,y) = (src(x-1,y) - src(x+1,y)) / 2;
}
```

We use image view's `operator(x,y)` to get a reference to the pixel at a given location and we set it to the half-difference of its left and right neighbors. operator() returns a reference to a grayscale pixel. A grayscale pixel is convertible to its channel type (`unsigned char` for `src`) and it can be copy-constructed from a channel. (This is only true for grayscale pixels). While the above code is easy to read, it is not very fast, because the binary `operator()` computes the location of the pixel in a 2D grid, which involves addition and multiplication. Here is a faster version of the above:

```
void x_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    for (int y=0; y<src.height(); ++y) {
        gray8c_view_t::x_iterator src_it = src.row_begin(y);
        gray8s_view_t::x_iterator dst_it = dst.row_begin(y);

        for (int x=1; x<src.width()-1; ++x)
            dst_it[x] = (src_it[x+1] - src_it[x-1]) / 2;
    }
}
```

We use pixel iterators initialized at the beginning of each row. GIL's iterators are STL Random Access iterators. If you are not familiar with random access iterators, think of them as if they were pointers. In fact, in the above example the two iterator types are raw C pointers and their `operator[]` is a fast pointer indexing operator.

The code to compute gradient in the vertical direction is very similar:

```
void y_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    for (int x=0; x<src.width(); ++x) {
        gray8c_view_t::y_iterator src_it = src.col_begin(x);
        gray8s_view_t::y_iterator dst_it = dst.col_begin(x);

        for (int y=1; y<src.height()-1; ++y)
            dst_it[y] = (src_it[y+1] - src_it[y-1])/2;
    }
}
```

Instead of looping over the rows, we loop over each column and create a `y_iterator`, an iterator moving vertically. In this case a simple pointer cannot be used because the distance between two adjacent pixels equals the number of bytes in each row of the image. GIL uses here a special step iterator class whose size is 8 bytes - it contains a raw C pointer and a step. Its `operator[]` multiplies the index by its step.

The above version of `y_gradient`, however, is much slower (easily an order of magnitude slower) than `x_gradient` because of the memory access pattern; traversing an image vertically results in lots of cache misses. A much more efficient and cache-friendly version will iterate over the columns in the inner loop:

```
void y_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    for (int y=1; y<src.height()-1; ++y) {
        gray8c_view_t::x_iterator src1_it = src.row_begin(y-1);
        gray8c_view_t::x_iterator src2_it = src.row_begin(y+1);
        gray8s_view_t::x_iterator dst_it = dst.row_begin(y);

        for (int x=0; x<src.width(); ++x) {
            *dst_it = ((*src2_it) - (*src1_it))/2;
            ++dst_it;
            ++src1_it;
            ++src2_it;
        }
    }
}
```

This sample code also shows an alternative way of using pixel iterators - instead of `operator[]` one could use increments and dereferences.

## Using Locators

Unfortunately this cache-friendly version requires the extra hassle of maintaining two separate iterators in the source view. For every pixel, we want to access its neighbors above and below it. Such relative access can be done with GIL locators:

```
void y_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    gray8c_view_t::xy_locator src_loc = src.xy_at(0,1);
    for (int y=1; y<src.height()-1; ++y) {
        gray8s_view_t::x_iterator dst_it  = dst.row_begin(y);
        for (int x=0; x<src.width(); ++x) {
            (*dst_it) = (src_loc(0,1) - src_loc(0,-1)) / 2;
            ++dst_it;
            ++src_loc.x();                  // each dimension can be advanced separately
        }
        src_loc+=int_point(-src.width(),1); // carriage return
    }
}
```

The first line creates a locator pointing to the first pixel of the second row of the source view. A GIL pixel locator is very similar to an iterator, except that it can move both horizontally and vertically. `src_loc.x()` and `src_loc.y()` return references to a horizontal and a vertical iterator respectively, which can be used to move the locator along the desired dimension, as shown above. Additionally, the locator can be advanced in both dimensions simultaneously using its `operator+=` and `operator-=`. Similar to image views, locators provide binary `operator()` which returns a reference to a pixel with a relative offset to the current locator position. For example, `src_loc(0,1)` returns a reference to the neighbor below the current pixel. Locators are very lightweight objects - in the above example the locator has a size of 8 bytes - it consists of a raw pointer to the current pixel and an int indicating the number of bytes from one row to the next (which is the step when moving vertically). The call to ++`src_loc`.x() corresponds to a single C pointer increment. However, the example above performs more computations than necessary. The code src_loc(0,1) has to compute the offset of the pixel in two dimensions, which is slow. Notice though that the offset of the two neighbors is the same, regardless of the pixel location. To improve the performance, GIL can cache and reuse this offset:

```
void y_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    gray8c_view_t::xy_locator src_loc = src.xy_at(0,1);
    gray8c_view_t::xy_locator::cached_location_t above = src_loc.cache_location(0,-1);
    gray8c_view_t::xy_locator::cached_location_t below = src_loc.cache_location(0, 1);

    for (int y=1; y<src.height()-1; ++y) {
        gray8s_view_t::x_iterator dst_it = dst.row_begin(y);

        for (int x=0; x<src.width(); ++x) {
            (*dst_it) = (src_loc[above] - src_loc[below])/2;
            ++dst_it;
            ++src_loc.x();
        }
        src_loc+=int_point(-src.width(),1);
    }
}
```

In this example "`src_loc[above]`" corresponds to a fast pointer indexing operation and the code is efficient.

## Creating a Generic Version of GIL Algorithms

Let us make our `x_gradient` more generic. It should work with any image views, as long as they have the same number of channels. The gradient operation is to be computed for each channel independently. Here is how the new interface looks like:

```
template <typename S_VIEW, typename D_VIEW>
void x_gradient(const S_VIEW& src, const D_VIEW& dst) {
    gil_function_requires<ImageViewConcept<S_VIEW> >();
    gil_function_requires<MutableImageViewConcept<D_VIEW> >();
    gil_function_requires<ColorSpacesCompatibleConcept<typename S_VIEW::color_space_t,
                                                       typename D_VIEW::color_space_t> >();

    ... // compute the gradient
}
```

The new algorithm now takes the types of the input and output image views as template parameters. That allows using both built-in GIL image views, as well as any user-defined image view classes. The first three lines are optional; they use `boost::concept_check` to ensure that the two arguments are valid GIL image views, that the second one is mutable and that their color spaces are compatible (i.e. have the same set of channels).

GIL does not require using its own built-in constructs. You are free to use your own channels, color spaces, iterators, locators, views and images. However, to work with the rest of GIL they have to satisfy a set of requirements; in other words, they have to *model* the corresponding GIL *concept*. GIL's concepts are defined in the user guide.

One of the biggest drawbacks of using templates and generic programming in C++ is that compile errors can be very difficult to comprehend. This is a side-effect of the lack of early type checking - a generic argument may not satisfy the requirements of a function, but the incompatibility may be triggered deep into a nested call, in code unfamiliar and hardly related to the problem. GIL uses `boost::concept_check` to mitigate this problem. The above three lines of code check whether the template parameters are valid models of their corresponding concepts. If a model is incorrect, the compile error will be inside `gil_function_requires`, which is much closer to the problem and easier to track. Furthermore, such checks get compiled out and have zero performance overhead. The disadvantage of using concept checks is the sometimes severe impact they have on compile time. This is why GIL performs concept checks only in debug mode, and only if `USE_GIL_CONCEPT_CHECK` is defined (off by default).

The body of the generic function is very similar to that of the concrete one. The biggest difference is that we need to loop over the channels of the pixel and compute the gradient for each channel:

```
template <typename S_VIEW, typename D_VIEW>
void x_gradient(const S_VIEW& src, const D_VIEW& dst) {
    for (int y=0; y<src.height(); ++y) {
        typename S_VIEW::x_iterator src_it = src.row_begin(y);
        typename D_VIEW::x_iterator dst_it = dst.row_begin(y);

        for (int x=1; x<src.width()-1; ++x)
            for (int c=0; c<S_VIEW::color_space_t::num_channels; ++c)
                dst_it[x][c] = (src_it[x-1][c]- src_it[x+1][c])/2;
    }
}
```

Having an explicit loop for each channel could be a performance problem. GIL allows us to abstract out such per-channel operations:

```
template <typename OUT>
struct halfdiff_cast_channels {
    template <typename T> OUT operator()(const T& in1, const T& in2) const {
        return OUT((in2-in1)/2);
    }
};

template <typename S_VIEW, typename D_VIEW>
void x_gradient(const S_VIEW& src, const D_VIEW& dst) {
    typedef typename D_VIEW::channel_t dst_channel_t;

    for (int y=0; y<src.height(); ++y) {
        typename S_VIEW::x_iterator src_it = src.row_begin(y);
        typename D_VIEW::x_iterator dst_it = dst.row_begin(y);

        for (int x=1; x<src.width()-1; ++x)
            transform_channels(src_it[x-1], src_it[x+1], dst_it[x],
                               halfdiff_cast_channels<dst_channel_t>());
    }
}
```

`transform_channels` is an example of a channel-level GIL algorithm. Other such algorithms are `fill_channels` and `for_each_channel`. They are the channel-level equivalents of STL's `transform`, `fill` and `for_each` respectively. GIL channel algorithms use static recursion to unroll the loops; they never loop over the channels explicitly. This is not so important because modern compilers (at least Visual Studio 8) already unroll most channel-level loops, such as the one above. However, another advantage of using GIL's channel-level algorithms is that they pair the channels semantically, not based on their order in memory. For example, the above example will properly match an RGB source with a BGR destination.

Here is how we can use our generic version with images of different types:

```
// Calling with 16-bit grayscale data
void XGradientGray16_Gray32(const unsigned short* src_pixels, ptrdiff_t src_row_bytes, int w, int h,
                                  signed int* dst_pixels, ptrdiff_t dst_row_bytes) {
    gray16c_view_t src=interleaved_view(w,h,(const gray16_pixel_t*)src_pixels,src_row_bytes);
    gray32s_view_t dst=interleaved_view(w,h,(     gray32s_pixel_t*)dst_pixels,dst_row_bytes);
    x_gradient(src,dst);
}

// Calling with 8-bit RGB data into 16-bit BGR
void XGradientRGB8_BGR16(const unsigned char* src_pixels, ptrdiff_t src_row_bytes, int w, int h,
                                 signed short* dst_pixels, ptrdiff_t dst_row_bytes) {
    rgb8c_view_t  src = interleaved_view(w,h,(const rgb8_pixel_t*)src_pixels,src_row_bytes);
    rgb16s_view_t dst = interleaved_view(w,h,(    rgb16s_pixel_t*)dst_pixels,dst_row_bytes);
    x_gradient(src,dst);
}

// Either or both the source and the destination could be planar - the gradient code does not change
void XGradientPlanarRGB8_RGB32(
           const unsigned short* src_r, const unsigned short* src_g, const unsigned short* src_b,
           ptrdiff_t src_row_bytes, int w, int h,
           signed int* dst_pixels, ptrdiff_t dst_row_bytes) {
    rgb16c_planar_view_t src=planar_rgb_view (w,h, src_r,src_g,src_b,         src_row_bytes);
    rgb32s_view_t        dst=interleaved_view(w,h,(rgb32s_pixel_t*)dst_pixels,dst_row_bytes);
    x_gradient(src,dst);
}
```

As these examples illustrate, both the source and the destination can be interleaved or planar, of any channel depth (assuming the destination channel is assignable to the source), and of any compatible color spaces.

### Image View Transformations

One way to compute the y-gradient is to rotate the image by 90 degrees, compute the x-gradient and rotate the result back. Here is how to do this in GIL:

```
template <typename SRC_V, typename DST_V>
void y_gradient(const SRC_V& src, const DST_V& dst) {
    x_gradient(rotated90cw_view(src), rotated90cw_view(dst));
}
```

`rotated90cw_view` takes an image view and returns an image view representing 90-degrees clockwise rotation of its input. It is an example of a GIL view transformation function. GIL provides a variety of transformation functions that can perform any axis-aligned rotation, transpose the view, flip it vertically or horizontally, extract a rectangular subimage, perform color conversion, subsample view, etc. The view transformation functions are fast and shallow - they don't copy the pixels, they just change the "coordinate system" of accessing the pixels. `rotated90cw_view`, for example, returns a view whose horizontal iterators are the vertical iterators of the original view. The above code to compute `y_gradient` is slow because of the memory access pattern; using `rotated90cw_view` does not make it any slower.

Another example: suppose we want to compute the gradient of the N-th channel of a color image. Here is how to do that:

```
template <typename SRC_V, typename DST_V>
void nth_channel_x_gradient(const SRC_V& src, int n, const DST_V& dst) {
    x_gradient(nth_channel_view(src, n), dst);
}
```

`nth_channel_view` is a view transformation function that takes any view and returns a single-channel (grayscale) view of its N-th channel. For interleaved RGB view, for example, the returned view is a step view - a view whose horizontal iterator skips over two channels when incremented. If applied on a planar RGB view, the returned type is a simple grayscale view whose horizontal iterator is a C pointer. Image view transformation functions can be piped together. For example, to compute the y gradient of the second channel of the even pixels in the view, use:

```
y_gradient(subsampled_view(nth_channel_view(src, 1), 2,2), dst);
```

GIL can sometimes simplify piped views. For example, two nested subsampled views (views that skip over pixels in X and in Y) can be represented as a single subsampled view whose step is the product of the steps of the two views.

### 1D pixel iterators

Let's go back to `x_gradient` one more time. Many image view algorithms apply the same operation for each pixel and GIL provides an abstraction to handle them. However, our algorithm has an unusual access pattern, as it skips the first and the last column. It would be nice and instructional to see how we can rewrite it in canonical form. The way to do that in GIL is to write a version that works for every pixel, but apply it only on the subimage that excludes the first and last column:

```
void x_gradient_unguarded(const gray8c_view_t& src, const gray8s_view_t& dst) {
    for (int y=0; y<src.height(); ++y) {
        gray8c_view_t::x_iterator src_it = src.row_begin(y);
        gray8s_view_t::x_iterator dst_it = dst.row_begin(y);

        for (int x=0; x<src.width(); ++x)
            dst_it[x] = (src_it[x+1] - src_it[x-1]) / 2;
    }
}

void x_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    assert(src.width()>=2);
    x_gradient_unguarded(subimage_view(src, 1, 0, src.width()-2, src.height()),
                         subimage_view(dst, 1, 0, src.width()-2, src.height()));
}
```

`subimage_view` is another example of a GIL view transformation function. It takes a source view and a rectangular region (in this case, defined as x_min,y_min,width,height) and returns a view operating on that region of the source view. The above implementation has no measurable performance degradation from the version that operates on the original views.

Now that `x_gradient_unguarded` operates on every pixel, we can rewrite it more compactly:

```
void x_gradient_unguarded(const gray8c_view_t& src, const gray8s_view_t& dst) {
    gray8c_view_t::iterator src_it = src.begin();
    for (gray8s_view_t::iterator dst_it = dst.begin(); dst_it!=dst.end(); ++dst_it, ++src_it)
        *dst_it = (src_it.x_it()[-1] - src_it.x_it()[1]) / 2;
}
```

GIL image views provide `begin()` and `end()` methods that return one dimensional pixel iterators called *pixel image iterators* which iterate over each pixel in the view, left to right and top to bottom. They do a proper "carriage return" - they skip any unused bytes at the end of a row. As such, they are slightly suboptimal, because they need to keep track of their current position with respect to the end of the row. Their increment operator performs one extra check (are we at the end of the row?), a check that is avoided if two nested loops are used instead. Pixel image iterators have a method `x_it()` which returns the more lightweight horizontal iterator that we used previously. Horizontal iterators have no notion of the end of rows. In this case, the horizontal iterators are raw C pointers. In our example, we must use the horizontal iterators to access the two neighbors properly, since they could reside outside the image view.

### STL Equivalent Algorithms

GIL provides STL equivalents of many algorithms. For example, `std::transform` is an STL algorithm that sets each element in a destination range the result of a generic function taking the corresponding element of the source range. In our example, we want to assign to each destination pixel the value of the half-difference of the horizontal neighbors of the corresponding source pixel. If we abstract that operation in a function object, we can use GIL's `transform_pixel_positions` to do that:

```
struct half_x_difference {
    int operator()(const gray8c_loc_t& src_loc) const {
        return (src_loc.x()[-1] - src_loc.x()[1]) / 2;
    }
};

void x_gradient(const gray8c_view_t& src, const gray8s_view_t& dst) {
    transform_pixel_positions(src, dst, half_x_difference());
}
```

GIL provides the algorithms `for_each_pixel` and `transform_pixels` which are image view equivalents of STL's `std::for_each` and `std::transform`. It also provides `for_each_pixel_position` and `transform_pixel_positions`, which instead of references to pixels, pass to the generic function pixel locators. This allows for more powerful functions, that can use the pixel neighbors through the passed locators. GIL algorithms iterate through the pixels using the more efficient two nested loops (as opposed to the single loop using 1-D iterators)

### Color Conversion

Instead of computing the gradient of each color plane of an image, we often want to compute the gradient of the luminosity. In other words, we want to convert the color image to grayscale and compute the gradient of the result. Here how to compute the luminosity gradient of a 32-bit float RGB image:

```
void x_gradient_rgb_luminosity(const rgb32fc_view_t& src, const gray8s_view_t& dst) {
    x_gradient(color_converted_view<gray8_pixel_t>(src), dst);
}
```

`color_converted_view` is a GIL view transformation function that takes any image view and returns a view in a target color space and channel depth (specified as template parameters). In our example, it constructs an 8-bit integer grayscale view over 32-bit float RGB pixels. Like all other view transformation functions, `color_converted_view` is very fast and shallow. It doesn't copy the data or perform any color conversion. Instead it returns a view that performs color conversion every time its pixels are accessed.

In the generic version of this algorithm we would like to convert the color space to grayscale, but keep the channel depth the same. We do that by constructing the type of a GIL grayscale pixel with the same channel as the source, and color convert to that pixel type:

```
template <typename S_VIEW, typename D_VIEW>
void x_luminosity_gradient(const S_VIEW& src, const D_VIEW& dst) {
    typedef pixel<typename S_VIEW::channel_t, gray_t> gray_pixel_t;
    x_gradient(color_converted_view<gray_pixel_t>(src), dst);
}
```

When the destination color space and channel type happens to be the same as the source one, color conversionis unnecessary. GIL detects this case and avoids calling the color conversion code at all - i.e. `color_converted_view` returns back the source view unchanged.

### Image

The above example has a performance problem - `x_gradient` dereferences most source pixels twice, which will cause the above code to perform color conversion twice. Sometimes it may be more efficient to copy the color converted image into a temporary buffer and use it to compute the gradient - that way color conversion is invoked once per pixel. Using our non-generic version we can do it like this:

```
void x_luminosity_gradient(const rgb32fc_view_t& src, const gray8_view_t& dst) {
    gray8_image_t ccv_image(src.dimensions());
    copy_pixels(color_converted_view<gray8_pixel_t>(src), view(ccv_image));

    x_gradient(const_view(ccv_image), dst);
}
```

First we construct an 8-bit grayscale image with the same dimensions as our source. Then we copy a color-converted view of the source into the temporary image. Finally we use a read-only view of the temporary image in our `x_gradient` algorithm. As the example shows, GIL provides global functions `view` and `const_view` that take an image and return a mutable or an immutable view of its pixels.

Creating a generic version of the above is a bit trickier:

```
template <typename S_VIEW, typename D_VIEW>
void x_luminosity_gradient(const S_VIEW& src, const D_VIEW& dst) {
    typedef typename S_VIEW::channel_t                     s_channel_t;
    typedef typename image_type<s_channel_t, gray_t>::type gray_image_t;
    typedef typename gray_image_t::pixel_t                 gray_pixel_t;

    gray_image_t ccv_image(src.dimensions());
    copy_pixels(color_converted_view<gray_pixel_t>(src), view(ccv_image));

    x_gradient(const_view(ccv_image), dst);
}
```

We get the channel type of the source view and use it to generate the type of a grayscale image with the same channel. GIL provides a set of metafunctions that generate GIL types - **`image_type`** is one such meta-function that constructs the type of an image from a given channel type, color space, and planar/interleaved option (the default is interleaved). There are also similar meta-functions to construct the types of pixel iterators, locators and image views. GIL also has metafunctions **`derived_image_type`** and **`derived_view_type`** that construct the type of an image or a view from a given source one by changing one or more properties of the type and keeping the rest.

## Run-Time Specified Images and Image Views

So far we have created a generic function that computes the image gradient of a templated image view. Sometimes, however, the properties of an image view, such as its color space and channel depth, may not be available at compile time. GIL's `dynamic_image` extension allows for working with GIL constructs that are specified at run time, also called *variants*. GIL provides models of a run-time instantiated image, **`any_image`**, and a run-time instantiated image view, **`any_image_view`**. The mechanisms are in place to create other variants, such as `any_pixel`, `any_pixel_iterator`, etc. Most of GIL's algorithms and all of the view transformation functions also work with run-time instantiated image views and binary algorithms, such as `copy_pixels` can have either or both arguments be variants.

Lets make our `x_luminosity_gradient` algorithm take a variant image view. For simplicity, let's assume that only the source view can be a variant. First, we need to make a function object that contains the templated destination view and has an application operator taking a templated source view:

```
#include <gil/extension/dynamic_image/dynamic_image_all.hpp>

template <typename D_VIEW>
struct x_gradient_obj {
    typedef void result_type;       // required typedef

    const D_VIEW& _dst;
    x_gradient_obj(const D_VIEW& dst) : _dst(dst) {}

    template <typename S_VIEW>
    void operator()(const S_VIEW& src) const { x_luminosity_gradient(src, _dst); }
};
```

The second step is to provide an overload of `x_luminosity_gradient` that takes image view variant and calls GIL's `apply_operation` passing it the function object:

```
template <typename S_VIEWS, typename D_VIEW>
void x_luminosity_gradient(const any_image_view<S_VIEWS>& src, const D_VIEW& dst) {
    apply_operation(src, x_gradient_obj<D_VIEW>(dst));
}
```

any_image_view<S_VIEWS> is the image view variant. It is templated over S_VIEWS, an enumeration of all possible view types the variant can take. src contains inside an index of the currently instantiated type, as well as a block of memory containing the instance. apply_operation goes through a switch statement over the index, each case of which casts the memory to the correct view type and invokes the function object with it. Invoking an algorithm on a variant has the overhead of one switch statement. Algorithms that perform an operation for each pixel in an image view have practically no performance degradation when used with a variant.

Here is how we can construct a variant and invoke the algorithm:

```
#include <boost/mpl/vector.hpp>
#include <gil/extension/dynamic_image/dynamic_image_all.hpp>
#include <gil/extension/io/image_io.hpp>

typedef mpl::vector<gray8_image_t, gray16_image_t, rgb8_image_t, rgb16_image_t> my_img_types;
any_image<my_img_types> runtime_image;
load_jpeg_image(runtime_image,"monkey.jpg");

gray8s_image_t gradient(get_dimensions(runtime_image));
x_gradient(const_view(runtime_image), view(gradient));
save_jpeg_view(const_view(gradient), "x_gradient.jpg");
```

In this example, we create an image variant that could be 8-bit or 16-bit RGB or grayscale image. We then use GIL's I/O extension to load the image from file in its native color space and channel depth. If none of the allowed image types fits the image on disk, an exception will be thrown. We then construct a 8 bit signed (i.e. char) image to store the gradient and invoke x_gradient on it. Finally we save the result into another file.

Note how methods such as load_jpeg_image, get_dimensions, view and const_view work on both templated and variant types. For templated images view(img) returns a templated view, whereas for image variants it returns a view variant. For example, the return type of view(runtime_image) is any_image_view<VIEWS> where VIEWS enumerates four views corresponding to the four image types. const_view(runtime_image) returns a **any_image_view** of the four read-only view types, etc.

Sometimes explicitly enumerating all possible types a variant can take can be cumbersome. GIL allows for specifying type vectors of images and image views implicitly, by specifying the set of allowed color spaces, channel types, etc. For example, here is how we can specify an image view variant that can be planar or interleaved, step or non-step, RGB or CMYK and 8 or 16 bits:

```
typedef cross_vector_image_view_types
    < mpl::vector<bits8, bits16>,
      mpl::vector<rgb_t, cmyk_t>,
      kInterleavedAndPlanar,
      kNonStepAndStep,
      false                         // false == mutable; true == read-only
    >::type my_views_t;
typedef any_image_view<my_views_t> my_any_image_view_t;
```

In the above example, my_any_image_view_t can take one of 16 possible image view types. GIL also allows combining type vectors each of which could be implicitly defined or enumerated. For example, here is how to add 8-bit and 32-bit grayscale image views to the above set:

```
typedef mpl::concat_vector<my_views_t, mpl::vector<gray8_view_t, gray32_view_t> > my_extended_views_t;
typedef any_image_view<my_extended_views_t> my_any_image_view2_t;
```

concat_vector lives in **boost::mpl** namespace but is provided by GIL. Instantiating an algorithm with a variant effectively instantiates it with every possible type the variant can take. For binary algorithms, the algorithm is instantiated with every possible combination of the two input types! This can take a toll on both the compile time and the executable size. GIL has some capabilities to detect instantiations that are identical at assembly level and remove duplicates thereby reducing the size of the executable. This mechanism can be enabled by defining `DO_REDUCE`.

## Appendix

### Naming convention for GIL concrete types

Concrete (non-generic) GIL types follow this naming convention:

*ColorSpace* + *BitDepth* + [`f` | `s`]+ [`c`] + [`_planar`] + [`_step`] + *ClassType* + `_t`

Where *ColorSpace* also indicates the ordering of components. Examples are `rgb`, `bgr`, `cmyk`, `rgba`. *BitDepth* indicates the bit depth of the color channel. Examples are `8`,`16`,`32`. By default the type of channel is unsigned integral; using `s` indicates signed integral and `f` - a floating point type, which is always signed. `c` indicates object operating over immutable pixels. `_planar` indicates planar organization (as opposed to interleaved). `_step` indicates special image views, locators and iterators which traverse the data in non-trivial way (for example, backwards or every other pixel). *ClassType* is `_image` (image), `_view` (image view), `_loc` (pixel 2D locator) `_ptr` (pixel iterator), `_ref` (pixel reference), `_pixel` (pixel value).

```
bgr8_image_t             a;    // 8-bit interleaved BGR image
cmyk16_pixel_t;          b;    // 16-bit CMYK pixel value;
cmyk16c_planar_ref_t     c(x); // const reference to a 16-bit planar CMYK pixel x.
rgb32_planar_step_ptr_t  d;    // step pointer to a 32-bit planar RGB pixel.
```