Appendix E

Programming computer vision in C / C++

In this appendix we will describe one way of implementing two-dimensional (2D) images in the C and C++ languages. These languages are chosen because they enable the construction of large-scale, real-time, maintainable software. Our goal will be to strive for simplicity and generality.

E.1 A grayscale image in C

Let us begin by implementing an image in C. An image is simply a two-dimensional array of pixels, so the first thing we have to decide is the type of elements in the array. We will assume that the elements are unsigned characters, since that is the most common format for grayscale pixels. In its simplest form, an image consists of three things: the width of the image, the height of the image, and the pixel values themselves. We will use a `struct` to group these together:

```
typedef struct
{
  int width, height;
  unsigned char* data;
} image;
```

This code creates a struct called `image` that contains three member variables: `width` to hold the number of columns of the image, `height` to hold the number of rows of the image, and `data`, which is a pointer to the first data element (pixel). We will assume that pixel data are stored in row-major order, so that the one-dimensional array `data` contains the values for the first row of pixels, followed by the values for the second row of pixels, followed by the values for the third row of pixels, and so on.

Some libraries also include in the image definition additional member variables to keep track of things like the type of pixel, whether there are any zero-padded pixels at the end of each row, or whether the origin of the image is in the top-left or bottom-right corner. Zero-padded pixels, however, are usually not worth the trouble, since most standard image sizes are already divisible by 8. For all practical purposes, the origin can be safely assumed to be in the top-left corner, since that is by far the most common convention. The other fields are not widely used either. As a result, we prefer this minimalist arrangement which also simplifies the present discussion.

The next step is to create functions to initialize and destroy the image. We will create two initialization functions, one which creates an empty image, and another which creates an image of a specific size:

```
struct image init_image()
{
  struct image im;
  im.width = im.height = im.data = 0;
  return im;
}

struct image init_image(int w, int h)
{
  struct image im;
  im.width = w;
  im.height = h;
  im.data = (unsigned char*) malloc(w * h * sizeof(unsigned char));
  return im;
}

void destroy_image(struct image im)
{
  free( im.data );
}
```

In the two `init_image` functions, a local `image` variable is created on the stack, its member variables are initialized, and it is returned. Upon returning from the functions, a shallow copy is made of the member variables, but any pixel data will remain on the heap. The `destroy_image` function frees any memory allocated for the pixels, which is safe because `free(0)` does not do anything.

Another useful function is to be able to resize an image. For this, we will need to pass in a pointer to the image so that changes to its member variables will persist after the function call returns:

```
void resize_image(struct image* im, int w, int h)
{
  im->width = w;
  im->height = h;
  im->data = (unsigned char*) realloc(im->data, w * h * sizeof(unsigned char));
}
```

The `realloc` function is particularly easy to use, because it will either resize the heap block as necessary, or it will copy the data to a new heap block and delete the old one. Keep in mind, however, that our `resize` function only ensures that the appropriate memory has been allocated; it does *not* ensure that the pixel data has been preserved in any meaningful way.

Now let us add two functions to access the pixels:

```
unsigned char get_pixel(struct image im, int x, int y)
{
  return im.data[ y * im.width + x ];
}

void set_pixel(struct image im, int x, int y, unsigned char val)
{
  im.data[ y * im.width + x ] = val;
}
```

It is easy to verify that the formula `y * im.width + x` calculates the correct one-dimensional zero-based index, assuming that `x` and `y` are also zero-based indices. Note that `set_pixel` does not need to

take a pointer to the image, since the change is being made to an element in the array (which is stored on the heap) rather than a member variable of the struct itself (of which a copy is made on the stack).

The image struct is now ready to be used in code:

```
void main()
{
  struct image im1, im2;                // two images
  unsigned char val;                    // a pixel

  im1 = init_image( 640, 480 );         // allocate memory for a 640 x 480 image
  im2 = init_image( );                  // create an empty image
  resize_image( &im2, 640, 480 );       // allocate memory for a 640 x 480 image
  val = get_pixel( im1, 3, 2 );         // get the pixel value at (3, 2)
  set_pixel( im1, 3, 2, val + 10 );     // increase the value at (3, 2) by 10 gray levels
  destroy_image( im1 );                 // free the memory
  destroy_image( im2 );                 // free the memory
}
```

Of course, in a real system we would want to check to make sure that `(3,2)` is not out of bounds, and that adding 10 to the pixel value does not cause it to exceed the maximum value, which is 255. These details are very important, and we hope our omission of them here – for the sake of streamlining the presentation – does not leave the reader with the wrong impression. Great care in attending to such details is highly recommended, and the programmer who does so will be amply rewarded with less development and debugging time, fewer headaches, less frustruation, and happier customers.

E.2 A grayscale image in C++

Now let us turn our attention to C++. This language has been designed to make it much easier to program in an object oriented paradigm. In particular, while a struct in C can contain member variables (as we just saw), it cannot contain methods (i.e., functions) except by using function pointers, which lead to cumbersome and unnatural syntax. In contrast, a struct in C++ can contain both member variables and methods in a natural way. For example, in C++ we could program our image like this:

```
struct ImageGray
{
  int width, height;
  unsigned char* data;
  unsigned char GetPixel( int x, int y ) { return data[ y * width + x ]; }
  void SetPixel( int x, int y, unsigned char val ) { data[ y * width + x ] = val; }
}
```

Here we have capitalized the class and method names for the sake of convention. Note that the syntax in C++ is already simpler, because we do not need to use `typedef` when defining a struct, and the methods are now part of the struct rather than being outside.

Of course, when programming in C++ we usually use classes rather than structs. The reason for this is more a matter of semantics, however, rather than any real change, because both are identical to the compiler. In other words, to a C++ compiler there is absolutely no difference between a struct and a class, except that structs are public by default, while classes are private by default. Changing a struct to private, or a class to public, results in the exact same behavior, respectively, as a class or a struct. (By the way, we recommend as a matter of convention using a class whenever the data members are private, and a struct whenever they are public.) Therefore, the following code is identical to that above:

```
class ImageGray
{
public:
  int width, height;
  unsigned char* data;
  unsigned char GetPixel( int x, int y ) { return data[ y * width + x ]; }
  void SetPixel( int x, int y, unsigned char val ) { data[ y * width + x ] = val; }
}
```

The problem with this solution is that we are not taking advantage of the most powerful feature of C++, namely, *data hiding*, or *encapsulation*. To do this, we make the data members private. By convention, we will prefix the name of each private member variable with `m_` to prevent confusion with local variables and parameters. These changes lead to the following:

```
class ImageGray
{
public:
  unsigned char GetPixel( int x, int y ) { return m_data[ y * m_width + x ]; }
  void SetPixel( int x, int y, unsigned char val ) { m_data[ y * m_width + x ] = val; }
private:
  int m_width, m_height;
  unsigned char* m_data;
}
```

Remember our `init_image` and `destroy_image` functions? C++ includes mechanisms for automating the initialization and destruction of objects. These mechanisms are the *constructor* and *destructor*. These methods are distinguished because their name is exactly the same as the name of the class, with an extra tilde for the destructor:

```
class ImageGray
{
public:
  ImageGray() { m_width = m_height = m_data = 0; }  // constructor
  ImageGray(int w, int h)  // another constructor
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) malloc(w * h * sizeof(unsigned char));
  }
  ~ImageGray() { free( m_data ); }  // destructor
  void Resize(int w, int h)
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) realloc(m_data, w * h * sizeof(unsigned char));
  }
  unsigned char GetPixel( int x, int y ) { return m_data[ y * m_width + x ]; }
  void SetPixel( int x, int y, unsigned char val ) { m_data[ y * m_width + x ] = val; }
private:
  int m_width, m_height;
  unsigned char* m_data;
}
```

A class can have as many constructors as you want, with each one being distinguished only by the number and type of parameters. Here we have two constructors, one that takes no parameters (called the *default constructor*) and one that takes two parameters (the width and height). A class has exactly one destructor, and it always takes no parameters. In the implementations of these methods, it is true that we could replace the C functions `malloc`, `realloc`, and `free` with their C++ counterparts `new` and `delete`, but since C++ is backward compatible, these work just as well, so we see no advantage to doing so. We have also added a `Resize` method, which works the same as the `resize_image` function above.

Now we have a class that we can use:

```
void main()
{
  ImageGray im1(640, 480);   // allocate memory for a 640 x 480 image (by calling constructor)
  ImageGray im2;             // an empty image (by calling the default constructor)
  unsigned char val;         // a pixel

  im2.Resize( 640, 480 );         // allocate memory for a 640 x 480 image
  val = im1.GetPixel( 3, 2 );     // get the pixel value at (3, 2)
  im1.SetPixel( 3, 2, val + 10 ); // increase the value at (3, 2) by 10 gray levels
}                                 // automatically call destructors for im1 and im2
```

Notice how much C++ simplifies the syntax. We no longer have to use the keyword `struct`, and we no longer have to explicitly call `init_image`. Instead, the appropriate constructor is called whenever we define an `ImageGray` object. Moreover, the constructor is automatically called whenever the object falls out of scope, so that whether the memory is freed is not dependent on our vigilance as a programmer to remember to call `destroy_image`. By delegating this responsibility to the compiler, we greatly reduce the chance for bugs and memory leaks in our program.

If you have understood the presentation so far, you are well on your way to being able to program computer vision in C++. It has been estimated that 90% of the value in the C++ language is contained in 10% of the features. As a result, once you become comfortable with constructors, destructors, and encapsulation, you can easily write clean C++ code that benefits from its major features. In fact, according to the inventor of C++, Bjarne Stroustrup, simple classes that hide their data and provide a clean interface, as we have done here — sometimes called *concrete classes* — are the "foundation of elegant programming."

E.3 A cleaner C++ class for a grayscale image

Nevertheless, additional features of the C++ language, once understood, make our life even easier. Here we consider six features: operator overloading, references, the keyword `const`, the copy constructor, the assignment operator, and initializer lists:

- *Operator overloading.* Instead of saying, `im1.GetPixel( 3, 2 )`, wouldn't it be nice if we could simply say `im1( 3, 2 )`? Well, this is actually possible in C++ by creating a method with a name that at first glance may look a bit strange: `unsigned char operator()(int x, int y)`. With this method, called `operator()`, we can now call `im1.operator()( 3, 2 )` to invoke it just as if it were any ordinary method. However, because the C++ compiler recognizes `operator()` as a special keyword, it also allows us to call the method using the shorthand `im1( 3, 2 )`. The value of a pixel can now be retrieved by simply calling `val = im1(3, 2)`.

- *References.* A reference is an alias, i.e., it is another name for the object. A reference acts like an automatically dereferenced pointer, allowing us to access an object without copying it. References are very useful when passing parameters into methods, as well as when returning

values from methods. References use the ampersand (`&`) symbol, which the reader will no doubt recognize as also the symbol used in C to "take the address of." By adding this ampersand to the method above, we get `unsigned char& operator()(int x, int y)`, which is similar to what we saw before except that now the method returns a reference (i.e, an alias) to the pixel itself rather than a copy of the pixel's value. This seemingly minor distinction has huge implications, because it makes it possible to put the method call on the left side of an equation, as in `im1(3, 2) = val`.

- *`const`.* The `const` keyword means "read-only." It can be used for either variables on the stack, parameters passed into a method, the return value of a method, or a method itself. In the first three cases, the meaning is clear: The variable or parameter cannot be modified. In the latter case, the meaning is this: None of the member variables of the object on which the method is called can be modified. Since the `GetPixel` method does not change any of the member variables' values, it is natural to make it `const`. The syntax for making a method const is to add the keyword after the closing parenthesis: `unsigned char operator()(int x, int y) const`. The reason for making the method const is to protect the *maintainer* of the class from accidentally changing variables in the method, as well as to protect the *user* of the class from calling a "non-read-only" method on what is supposed to be a "read-only" object. Such compile-time checks greatly reduce chances for bugs.

- *Copy constructor.* The copy constructor is a special kind of constructor that takes a single argument of a special type, namely a const reference to the class itself. The copy constructor is called automatically whenever the compiler needs to make a copy of the object, such as when passing the object into a function (without using references or pointers), returning the object from a function (again, without references or pointers), and at construction time. The copy constructor can be made as complicated as you want (for example, to implement reference counting), but we prefer the simpler approach of making a deep copy of all the member variables.

- *Assignment operator.* The assignment operator is called whenever an object is followed by the equal (`=`) sign — except for the line in which it is defined. The implementation of the assignment operator is often the same as the copy constructor.

- *Initializer lists.* In a constructor, a colon (`:`) followed by the names of the parameters, may be inserted just before the opening curly brace. This enables member variables to be initialized when they are constructed, before any code of the constructor is actually called. It is a good habit to initialize all member variables in this manner.

If the reader feels overwhelmed by these concepts, which are hardly done justice by the cursory descriptions just given, consulting any good C++ tutorial or reference for further details is recommended. Once the reader is comfortable with these concepts, it will be worthwhile to examine the following C++ class, which incorporates them:

```
class ImageGray
{
public:
  ImageGray() : m_width(0), m_height(0), m_data(0) {}
  ImageGray(int w, int h)
    : m_width(0), m_height(0), m_data(0)   // not needed, but makes code cleaner
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) malloc(w * h * sizeof(unsigned char));
  }
  ImageGray(const ImageGray& other)
```

```
    : m_width(0), m_height(0), m_data(0)   // needed to initialize m_data
  {
    *this = other;  // call assignment operator to make copy
  }
  ~ImageGray() { free( m_data ); }
  void Resize(int w, int h)
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) realloc(m_data, w * h * sizeof(unsigned char));
  }
  ImageGray& operator=( const ImageGray& other )
  {
    // make a deep copy
    Resize( other.m_width, other.m_height );
    memcpy( m_data, other.m_data, m_width * m_height * sizeof(unsigned char));
    return *this;
  }
  unsigned char operator()( int x, int y ) const { return m_data[ y * m_width + x ]; }
  unsigned char& operator()( int x, int y ) { return m_data[ y * m_width + x ]; }
private:
  int m_width, m_height;
  unsigned char* m_data;
}
```

An example using the class is as follows:

```
void main()
{
  ImageGray im1(640, 480); // allocate memory for a 640 x 480 image (by calling constructor)
  ImageGray im2 = im1;     // call copy constructor
  unsigned char val;       // a pixel

  im2 = im1;               // call assignment operator
  val = im1( 3, 2 );       // get the pixel value at (3, 2)
  im1( 3, 2) = val + 10;   // increase the value at (3, 2) by 10 gray levels
}                          // automatically call destructors for im1 and im2
```

Note that operator overloading and the reference have allowed us to simplify the syntax, while `const` and the initializer list have not affected our use of the code here — although they have made the code cleaner for future use and maintenance. The copy constructor and assignment operator are important; neglecting them will cause the default implementations to be used, which will cause unstable behavior due to the class' use of pointers and the heap. Note that the equal sign, when it appears in the same line as the definition, actually calls the copy constructor rather than the assignment operator. This is one of the peculiarities of C++, but it doesn't make any difference to us since we have written both methods to perform the same.

E.4 A color image in C++

The simplest color image is RGB, which stores three bytes for each pixel: the red, green, and blue values. For some reason, graphics cards and JPEG images have for decades used the convention that the bytes are stored in reverse order: first the blue, then the green, then the red. We will adopt this

convention, if for no other reason to avoid unnecessary reordering of the pixels every time the image is displayed.

Two options exist. One option is to represent the three color channels as *planes*, that is, we store the blue values of all the pixels, followed by the red values of all the pixels, followed by the green values of all the pixels. In this approach we have three images concatenated, so that the three values of any given pixel are not contiguous in memory. The other option is to *interleave* the color channels, so that the blue, green, and red values of the first pixel are stored, followed by the blue, green, and red values of the second pixel, and so on. The advantage of this approach is that it is the format used by graphics cards, and it also keeps all the values for a given pixel together. For these reasons we will adopt this latter approach.

Let us create a struct to hold the three color values of a pixel:

```
struct Bgr
{
  unsigned char b, g, r;
}
```

Since there is no difference between a struct and a class, we could have used a class as well. But we use struct here to emphasize that the member variables themselves are public.

A color image class is the same as the grayscale version above except that we replace `unsigned char` with `Bgr`. We also change the return type of the `const` version of the parenthesis operator to `const Bgr&` to avoid unnecessary copying of the struct:

```
class ImageBgr
{
public:
  ImageBgr() : m_width(0), m_height(0), m_data(0) {}
  ImageBgr(int w, int h)
    : m_width(0), m_height(0), m_data(0)   // not needed, but makes code cleaner
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) malloc(w * h * sizeof(Bgr));
  }
  ImageBgr(const ImageBgr& other)
    : m_width(0), m_height(0), m_data(0)   // needed to initialize m_data
  {
    *this = other;  // call assignment operator to make copy
  }
  ~ImageBgr() { free( m_data ); }
  void Resize(int w, int h)
  {
    m_width = w;
    m_height = h;
    m_data = (Bgr*) realloc(m_data, w * h * sizeof(Bgr));
  }
  ImageBgr& operator=( const ImageBgr& other )
  {
    // make a deep copy
    Resize( other.m_width, other.m_height );
    memcpy( m_data, other.m_data, m_width * m_height * sizeof(Bgr));
    return *this;
```

```
  }
  const Bgr& operator()( int x, int y ) const { return m_data[ y * m_width + x ]; }
  Bgr& operator()( int x, int y ) { return m_data[ y * m_width + x ]; }
private:
  int m_width, m_height;
  Bgr* m_data;
}
```

The class is used in the same manner as the grayscale version:

```
void main()
{
  ImageBgr im1(640, 480);  // allocate memory for a 640 x 480 image (by calling constructor)
  ImageBgr im2 = im1;      // call copy constructor
  Bgr val;                 // a pixel

  im2 = im1;               // call assignment operator
  val = im1( 3, 2 );       // get the pixel value at (3, 2)
  val.b += 10;
  val.g += 10;
  val.r += 10;
  im1( 3, 2) = val;        // increase the value at (3, 2) by 10 for each color channel
}                          // automatically call destructors for im1 and im2
```

The C++ mechanism for handling multiple data types (at compile time, at least) is the template. The templated version of our image class is exactly the same the one above, replacing `Bgr` with a generic type `T`. This leads to our final class implementation:

```
template <classname T>
class Image
{
public:
  Image() : m_width(0), m_height(0), m_data(0) {}
  Image(int w, int h)
    : m_width(0), m_height(0), m_data(0)   // not needed, but makes code cleaner
  {
    m_width = w;
    m_height = h;
    m_data = (unsigned char*) malloc(w * h * sizeof(T));
  }
  Image(const Image& other)
    : m_width(0), m_height(0), m_data(0)   // needed to initialize m_data
  {
    *this = other;  // call assignment operator to make copy
  }
  ~Image() { free( m_data ); }
  void Resize(int w, int h)
  {
    m_width = w;
    m_height = h;
    m_data = (T*) realloc(m_data, w * h * sizeof(T));
  }
  Image& operator=( const Image& other )
```

```
  {
    // make a deep copy
    Resize( other.m_width, other.m_height );
    memcpy( m_data, other.m_data, m_width * m_height * sizeof(T));
    return *this;
  }
  const T& operator()( int x, int y ) const { return m_data[ y * m_width + x ]; }
  T& operator()( int x, int y ) { return m_data[ y * m_width + x ]; }
private:
  int m_width, m_height;
  T* m_data;
}
```

Now we can simply define our two classes as:

```
typedef Image<unsigned char> ImageGray;
typedef Image<Bgr> ImageBgr;
```

With these definitions, these classes can be used exactly as they were when we defined them independently. The advantage of using templates is that it simplifies code maintenance and allows us to add new data types in the future with minimal extra work.