IN THIS CHAPTER

C++ Concepts
Memory Allocation Object
Singleton Object
Reference Counting
Template Primer
Handles and Rep objects
Prototyping Strategy

Image Framework Concepts
Image Object (Prototype 1)
Templated Image Object (Prototype 2)
Image Storage (Prototype 3)

3

Design Techniques

In this chapter, we lay the groundwork for extending our digital imaging framework. We begin by designing a memory allocation object, and then continue with a templates primer that provides a road map to subtleties of templates and their use. Finally, we apply C++ constructs to specific aspects of the design by creating detailed prototypes, and discussing the advantages and disadvantages of each technique.

3.1 Memory Allocation

In our test application's image class described in Section 2.3.1 on page 12, we use the operators **`new`** and **`delete`** to allocate and free storage for our image pixels in **`apImage::init()`** and **`apImage::cleanup()`**. Our test application employs this simple memory management scheme to demonstrate that it can be trivial to come up with a working solution. This simple mechanism, however, is very inefficient and breaks down quickly as you try to extend it. Managing memory is critical for a fully functional image framework. Therefore, before we delve into adding functionality, we will design an object that performs and manages memory allocation.

3.1.1 Why a Memory Allocation Object Is Needed

Images require a great deal of memory storage to hold the pixel data. It is very inefficient to copy these images, in terms of both memory storage and time, as the images are manipulated and processed. You can easily run out of memory if there are a large number of images. In addition, the heap could become fragmented if there isn't a large enough block of memory left after all of the allocations.

You really have to think about the purpose of an image before duplicating it. Duplication of image data should only happen when there is a good reason to retain a copy of the image (for example, you want to keep the original image and the filtered image result).

► **EXAMPLE**

A simple example illustrates the inefficiencies that can occur when manipulating images. Try adding two images together as follows:

```
apImage a (...);
apImage b (...);
apImage c = a + b;
```

The code allocates memory to store image `a`, allocates memory to store image `b`, allocates more memory to store the temporary image `(a+b)`, and finally allocates memory for the resulting image `c`. This simple example is bogged down with many memory allocations, and the time required for copying all the pixel data is excessive.

◄

We use this example as a simple way of showing how much memory and time a seemingly trivial operation can require. Note that some compilers can eliminate the temporary `(a+b)` storage allocation by employing a technique called return value optimization [Meyers96].

3.1.2 Memory Allocation Object Requirements

Instead of designing an object that works only for images, we create a generic object that is useful for any application requiring allocation and management of heap memory. We had to overcome the desire to produce the perfect object, because we do not have an unlimited budget or time. Commercial software is fluid; it is more important that we design it such that it can be adapted, modified, and extended in the future. The design we are presenting here is actually the third iteration.

Here's the list of requirements for our generic memory allocation object:

- Allocates memory off the heap, while also allowing custom memory allocators to be defined for allocating memory from other places, such as private memory heaps.
- Uses reference counting to share and automatically delete memory when it is no longer needed.
- Employs locking and unlocking as a way of managing objects in multithreaded applications (not shown here, but included in the software on the CD.-ROM For more information, see section *apLock* on page 128).
- Has very low overhead. For example, no memory initialization is done after allocation. This is left to the user to do, if needed.
- Uses templates so that the unit of allocation is arbitrary.
- Supports simple arrays, `[]`, as well as direct access to memory.
- Throws Standard Template Library (STL) exceptions when invalid attempts are made to access memory.
- Aligns the beginning of memory to a specified boundary. The need for this isn't obvious until you consider that certain image processing algorithms can take advantage of how the data is arranged in memory. Some compilers, such as Microsoft Visual C++,

can control the alignment when memory is allocated. We include this feature in a platform-independent way.

Before we move forward with our own memory allocation object, it is wise to see if any standard solutions exist.

Before designing your own solution, look to see if there is an existing solution that you can adapt or use directly.

The STL is always a good resource for solutions. You can imagine where **`std::vector`**, **`std::list`**, or even **`std::string`** could be used to manage memory. Each has its advantages, but none of these template classes offer reference counting. And even if reference counting were not an issue, there are performance issues to worry about. Each of these template objects provides fast random access to our pixel data, but they are also optimized for insertion and deletion of data, which is something we do not need.

Why Reference Counting Is Essential

Our solution is to create a generic object that uses *reference counting* to share and automatically delete memory when finished. Reference counting allows different objects to share the same information. Figure 3.3 shows three objects sharing image data from the same block of memory.

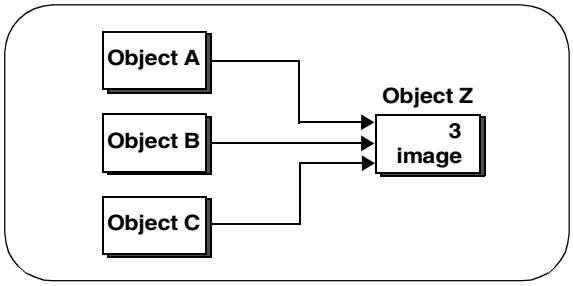


Figure 3.3: Objects Sharing Memory

In Figure 3.3, the object containing the image data, **`Object Z`**, also contains the reference count, **`3`**, which indicates how many objects are sharing the data. **`Objects A`**, **`B`**, and **`C`** all share the same image data.

Consider the following:

```
apImage image2 = image1
```

If reference counting is used in this example, **`image2`** will share the same storage as **`image1`**. If **`image1`** and **`image2`** point to identical memory, a little bookkeeping is necessary to make

sure this shared storage is valid while either image is in scope. That's where reference counting comes in.

Here's how it works in a nutshell. A memory allocation object allocates storage for an image. When subsequent images need to share that storage, the memory allocation object increments a variable, called a reference count, to keep track of the images sharing the storage; then it returns a pointer to the memory allocation object. When one of those images is deleted, the reference count is decremented. When the reference count decrements to zero, the memory used for storage is deleted. Let's look at an example using our memory allocation object, **apAlloc<>**.

```
apAlloc<int> array1 (100);
int i;
for (i=0; i<100; i++)
  array1[i] = i;
apAlloc<int> array2 = array1;
for (i=0; i<100; i++)
  array1[i] = i*2;
```

Once the **apAlloc<>** object is constructed, it is used much like any pointer. In this example, we create and populate an object **array1** with data. After assigning this object to **array2**, the code modifies the contents of **array1**. **array2** now contains the same contents as **array1**. Reference counting is not a new invention, and there are many in-depth discussions on the subject. See [Meyers96] and [Stroustrup00].

3.1.3 A Primer on Templates

The memory allocator objects use templates. The syntax can be confusing and tedious if you aren't used to it. Compilers are very finicky when it comes to handling templates, so the syntax becomes extremely important. Therefore, we provide a quick review of template syntax using a cookbook format. For a more detailed discussion, see [Stroustrup00] or [Vandevoorde03].

Converting a Class to Templates

Consider this simple image class that we want to convert to a template class:

```
class apImageTest
{
public:
  apImageTest (int width, int height, char* pixels);
  char* getPixel (int x, int y);
  void setPixel (int x, int y, char* pixel);
private:
  char* pixels_;
  int width_, height_;
};
```

The conversion is as easy as substituting a type name **T** for references to **char** as shown:

```
template<class T> class apImageTest
{
public:
```

```
    apImageTest (int width, int height, T* pixels);
    T* getPixel (int x, int y);
    void setPixel (int x, int y, T* pixel);
  private:
    T* pixels_;
    int width_, height_;
  };
```

To use this object, you would replace references of **apImageTest** with **apImageTest<char>**.

Be careful of datatypes when converting a function to a template function. When using common types, like int, and converting it to T, there may be places where int is still desired, such as in loop counters.

Type Argument Names

Any placeholder name can be used to represent the template arguments. We use a single letter (usually **T**), but you can use more descriptive names if you want. Consider the following:

```
template<class Pixel> class apImageTest;
```

Pixel is much more descriptive than **T** and may make your code more readable. There is no requirement to capitalize the first letter of the name, but we recommend doing so to avoid confusing argument names and variables. If you write a lot of template classes you will probably find that single-letter names are easier to use. Our final **apImage<>** class will have two arguments, **T** and **S,** that mean nothing out of context; however, when you are looking at the code, the parameters quickly take on meaning.

class Versus typename

The word **class** is used to define an argument name, but this argument does not have to be a class. In our **apImageTest** example shown earlier, we wrote **apImageTest<char>**, which expands the first line of the class declaration to:

```
template<class char> class apImageTest;
```

Although **char** is not a class, the compiler does not literally assume that the argument name **T** must be a class. You are also free to use the name **typename** instead of **class** when referring to templates. These are all valid examples of the same definition:

```
template<class T> class apImageTest;
template<typename T> class apImageTest;
template<class Pixel> class apImageTest;
template<typename Pixel> class apImageTest;
```

Is there ever a case where **class** is not valid? Yes, because there can be parsing ambiguities when dependent types are used. See [Meyers01]. Late-model compilers that have kept current with the C++ Standard, such as gcc v3.1, will generate warning messages if

typename is missing from these situations. For example, using gcc v3.1, the following line of code:

```
apImage<T1>::row_iterator i1 = src1.row_begin();
```

produces a warning:

```
warning: 'typename apImage<T1>::row_iterator' is implicitly a
typename
warning: implicit typename is deprecated, please see the
documentation for details
```

The compiler determines that **row_iterator** is a type rather than an instance of a variable or object, but warns of the ambiguity. To eliminate this warning, you must explicitly add **typename**, as shown:

```
typename apImage<T1>::row_iterator i1 = src1.row_begin();
```

Another case where **typename** is an issue is in template specialization, because the use of **typename** is forbidden. [Vandevoorde03] points out that the C++ standardization committee is considering relaxing some of the **typename** rules. For example, you might think that the previous example could be converted as follows:

```
typename apImage<apRGB>::row_iterator i1 = src1.row_begin();
```

where **apRGB** is the specialization. However, this generates an error:

```
In function '...': using 'typename' outside of template
```

To resolve the error, you must remove **typename**, as shown:

```
apImage<apRGB>::row_iterator i1 = src1.row_begin();
```

There is a growing movement to use **typename** instead of **class** because of the confusion some new programmers encounter when using templates. If you do not have a clear preference, we recommend that you use **typename**. The most important thing is that you are consistent in your choice.

Default Template Arguments

You can supply default template arguments much like you can with any function argument. These default arguments can even contain other template arguments, making them extremely powerful. For example, our **apAlloc<>** class that we design in Section 3.1.5 on page 31 is defined as:

```
template<class T, class A = apAllocator_<T> >
class apAlloc
```

As we will see in that section, anyone who does not mind memory being allocated on the heap can ignore the second argument, and the **apAllocator_<>** object will be used to allocate memory. Most clients can think of **apAlloc<>** as having only a single parameter, and be blissfully ignorant of the details.

The syntax we used by adding a space between the two '**>**' characters is significant. Defining the line as:

✗
```
template<class T, class A = apAllocator_<T>>
```

will produce an error or warning. The error message that the compiler produces in this case is extremely cryptic, and can take many iterations to locate and fix. Basically, the compiler is interpreting **>>** as **operator>>**. It is best to avoid this potential trap by adding a space between the two **>** characters.

Inline Versus Non-Inline Template Definitions

❖ INLINE DEFINITION

Many template developers put the implementation inside the class definition to save a lot of typing. For example, we could have given the **getPixel()** function in our **apImageTest<>** object an inline definition in the header file this way:

```
template<class T> class apImageTest
{
public:
  ...
  T* getPixel (int x, int y)
  { return pixels_[y*width_ + x];} // No error detection
  ...
};
```

❖ NON-INLINE DEFINITION

We can also define **getPixel()** after the class definition (non-inline) in the header file:

```
template <class T>
T* apImageTest<T>::getPixel (int x, int y)
{ return pixels_[y*width_ + x];}
```

Now you know why many developers specify the implementation inside the definition — it is much less typing. For more complex definitions, however, you may want to define the implementation after the definition for clarity, or even in a separate file. If the template file is large and included everywhere, putting the implementation in a separate file can speed compilation. The choice is yours.

The copy constructor makes a particularly interesting example of complex syntax:

```
template<class T>
apImageTest<T>::apImageTest (const apImageTest& src)
{...}
```

It is hard to get the syntax correct on an example like this one. The error messages generated by compilers in this case are not particularly helpful, so we recommend that you refer to a C++ reference book. See [Stroustrup00] or [Vandevoorde03].

Template Specialization

Templates define the behavior for all types (type **T** in our examples). But what happens if the definition for a generic type is slow and inefficient? Specialization is a method where

additional member function definitions can be defined for specific types. Consider an image class, **apImage<T>**. The parameter **T** can be anything, including some seldom used types like **double** or even **std::string**. But what if 90 percent of the images in your application are of a specific type, such as **unsigned char**? An inefficient algorithm is fine for a generic parameter, but we would certainly like the opportunity to tell the compiler what to do if the type is **unsigned char**.

To define a specialization, you first need the generic definition. It is good to write this first anyway to flesh out what each member function does. If you choose to write only the specialization, you should define the generic version to throw an error so that you will know if you ever call it unintentionally.

Once the generic version is defined, the specialization for **unsigned char** can be defined as shown:

```
template<> class apImageTest<unsigned char>
{
public:
  apImageTest (int width, int height, unsigned char* pixels);
  unsigned char* getPixel (int x, int y);
  void setPixel (int x, int y, unsigned char* pixel);
private:
  unsigned char* pixels_;
  int width_, height_;
};
```

We can now proceed with defining each specialized member function. Is it possible to define a specialization for just a single member function? The answer is yes. Consider a very generic implementation for our **apImageTest<>** constructor:

```
template<class T>
apImageTest<T>::apImageTest (int width, int height, T* pixels)
: width_ (width), height_ (height), pixels_ (0)
{
  pixels_ = new T [width_ * height_];
  T* p = pixels_;
  for (int y=0; y<height; y++) {
    for (int x=0; x<width; x++)
      *p++ = *pixels++;  // use assignment to copy pixels
  }
}
```

This definition is careful to use assignment to copy each pixel from the given array to the one controlled by the class. Now, let us define a specialization when **T** is an **unsigned char**:

```
apImageTest<unsigned char>::apImageTest
  (int width, int height, unsigned char* pixels)
  : width_ (width), height_ (height), pixels_ (0)
{
  pixels_ = new unsigned char [width_ * height_];
  memcpy (pixels_, pixels, width_ * height_);
}
```

We can safely use **memcpy()** to initialize our pixel data. You can see that the syntax for specialization is different if you are defining the complete specialization, or just a single member function.

Function Templates

C++ allows the use of templates to extend beyond objects to also include simple function definitions. Finally we have the ability to get rid of most macros. For example, we can replace the macro **min()**:

```
#ifndef min
#define min(a,b) (((a) < (b)) ? (a) : (b))
#endif
```

with a function template **min()**:

```
template<class T> const T& min (const T& a, const T& b)
{ return (a<b) ? a : b;}
```

❖ FUNCTION TEMPLATE SPECIALIZATION

We can even define function template specializations:

```
template<> const char& min<char> (const char& a, const char& b)
{ return (a<b) ? a : b;}
```

In this example, the specialization is unnecessary, but it does show the special syntax you will need to use for specialization.

Function templates can also have multiple template parameters, but don't be surprised if the compiler sometimes selects a function you don't expect. Here is an example of a function that we used in an early iteration of our image framework:

```
template<class T1, class T2, class T3>
void add2 (const T1& s1, const T2& s2, T3& d1)
{ d1 = s1 + s2;}
```

We needed such a function inside our image processing functions to control the behavior in overflow and underflow conditions. Specialized versions of this function test for overflow and clamp the output value at the maximum value for that data type, while other versions test for underflow and clamp the output value at the minimum value. The generic definition shown above just adds two source values and produces an output.

You must be careful with these mixed-type function templates. It is entirely possible that the compiler will not be able to determine which version to call. We could not use the above definition of **add2<>** with recent C++ compilers, such as Microsoft Visual Studio, because our numerous overrides make it ambiguous, according to the latest C++ standard, as to which version of **add2<>** to call. So, our solution is to define non-template versions for all our expected data types, such as:

```
void add2 (int s1, int s2, int& d1);
```

If you plan on using mixed-type function templates, you should definitely create prototypes and compile and run them with the compilers you expect to use. There are still many older

compilers that will compile the code correctly, but the code does not comply with the latest C++ standards. As compilers gain this compliance, you will need to implement solutions that conform to these standards.

Explicit Template Instantiation

C++ allows you to explicitly instantiate one or more template arguments. We can rewrite our **`add2()`** example to return the result, rather than pass it as a reference, as follows:

```
template<class R, class T1, class T2>
R add2 (const T1& s1, const T2& s2)
{ return static_cast<R> (s1 + s2);}
```

There is no way for the compiler to decide what the return type is without the type being specified explicitly. Whenever you use this form of **`add2()`**, you must explicitly specify the destination type for the compiler, as follows:

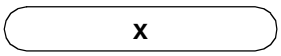
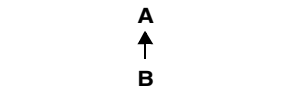
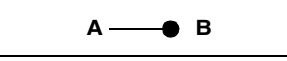
```
add2<double>(1.1, 2.2);
add2<int>(1.1, 2.2);
```

We will use explicit template instantiation later in the book to specify the data type for intermediate image processing calculations. For more information on explicit instantiation or other template issues, see [Vandevoorde03].

3.1.4 Notations Used in Class Diagrams

Table 3.1 shows a simple set of notations we use throughout the book to make the relationships clear in class diagrams. See [Lakos96].

Table 3.1: Notations Used in Class Diagrams

Notation	Meaning
	**X** is a class
	**B** is a kind of **A** (inheritance)
	**B**'s implementation uses **A**

3.1.5 Memory Allocator Object's Class Hierarchy

The class hierarchy for the memory allocator object is shown in Figure 3.4.

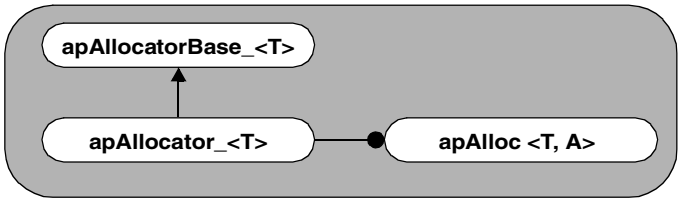


Figure 3.4: Memory Allocator Object Class Diagram

It consists of a base class, a derived class, and then the object class, which uses the derived class as one of its arguments. All three classes use templates. Note that we have appended an underscore character, _, to some class names to indicate that they are internal classes used in the API, but never called directly by its clients.

`apAllocatorBase_<>` is a base class that manages memory and contains all of the required functionality, except for the actual allocation and deallocation of memory. Its constructor basically initializes the object variables.

`apAllocator_<>` is derived from **`apAllocatorBase_<>`**. **`apAllocator_<>`** manages the allocation and deallocation of memory from the heap. You can use **`apAllocator_<>`** as a model for deriving the classes from **`apAllocatorBase_<>`** that use other allocation schemes.

`apAlloc<>` is a simple interface that the application uses to manage memory. **`apAlloc<>`** uses an **`apAllocator_<>`** object as one of its parameters to determine how to manage memory. By default, **`apAlloc<>`** allocates memory off the heap, because this is what our **`apAllocator_<>`** object does. However, if the application requires a different memory management scheme, a new derived allocator object can be easily created and passed to **`apAlloc<>`**.

apAllocatorBase_<> Class

The **`apAllocatorBase_<>`** base class contains the raw pointers and methods to access memory. It provides both access to the raw storage pointer, and access to the reference count pointing to shared storage, while also defining a reference counting mechanism. **`apAllocatorBase_<>`** takes a single template parameter that specifies the unit of memory to be allocated. The full base class definition is shown here.

```
template<class T> class apAllocatorBase_
{
public:
    apAllocatorBase_ (unsigned int n, unsigned int align)
    : pRaw_ (0), pData_ (0), ref_ (0), size_ (n), align_ (align)
    {}
```

```
    // Derived classes alloc memory; store details in base class

    virtual ~apAllocatorBase_ () {}
    // Derived classes will deallocate memory

    operator       T* ()       { return pData_;}
    operator const T* () const { return pData_;}
    // Conversion to pointer of allocated memory type

    unsigned int size  () const { return size_;}  // Number of elements
    unsigned int ref   () const { return ref_;}   // Number of references
    unsigned int align () const { return align_;} // Alignment

    void addRef () { ref_++; }
    void subRef ()
    {
      --ref_;
      if (ref_ == 0) delete this;
    }
    // Increment or decrement the reference count

protected:
    virtual void allocate   () = 0;
    virtual void deallocate () = 0;
    // Used to allocate/deallocate memory.

    T* alignPointer (void* raw);
    // Align the specified pointer to match our alignment

    apAllocatorBase_            (const apAllocatorBase_& src);
    apAllocatorBase_& operator= (const apAllocatorBase_& src);
    // No copy or assignment allowed.

    char*        pRaw_;  // Raw allocated pointer
    T*           pData_; // Aligned pointer to our memory
    unsigned int size_;  // Number of elements allocated
    unsigned int ref_;   // Reference count
    unsigned int align_; // Memory alignment (modulus)
};
```

❖ ALLOCATION AND DEALLOCATION

You'll notice that the constructor in **apAllocatorBase_** doesn't do anything other than initialize the object variables. We would have liked to have had the base class handle allocation and deallocation too, but doing so would have locked derived classes into heap allocation. This isn't obvious, until you consider how objects are constructed, as we will see in the next example.

► EXAMPLE

Suppose we designed the base class and included allocate and deallocate functions as shown:

✗

```
template<class T> class apAllocatorBase_
{
public:
    apAllocatorBase_ (unsigned int n) : size_ (n) { allocate ();}
    virtual ~apAllocatorBase_ ()                  { deallocate();}

    virtual void allocate () { pData_ = new T [size_];}
    virtual void deallocate () { delete [] pData_;}
};
```

It appears that we could derive an object, **apAllocatorCustom<>**, from **apAllocatorBase_<>** and override the **allocate()** and **deallocate()** methods. There is nothing to stop you from doing this, but you won't get the desired results. The reason is that when the **apAllocatorBase_<>** constructor runs, it obtains its definition for **allocate()** and **deallocate()** from the base class, because the derived versions are not available until the object is fully constructed.

Watch out for bugs introduced in the constructor of base and other parent classes. Make sure an object is fully constructed before calling any virtual function.

We found it cleaner to define a base class, **apAllocatorBase_<>**, and later derive the object **apAllocator_<>** from it. The derived object handles the actual allocation and deallocation. This makes creating custom memory management schemes very simple. **apAlloc<>** simply takes an **apAllocator_<>** object of the appropriate type and uses that memory management scheme.

We made **apAllocatorBase_<>** an abstract class by doing the following:

```
virtual void allocate   () = 0;
virtual void deallocate () = 0;
```

apAllocatorBase_<> never calls these functions directly, but by adding them we provide an obvious clue as to what functions need to be implemented in the derived class.

❖ CONVERSION OPERATORS

Let's take a look at the non-obvious accessor functions in **apAllocatorBase_<>**:

```
operator       T* ()       { return pData_;}
operator const T* () const { return pData_;}
```

We define two different types of **T*** conversion operators, one that is **const** and one that isn't. This is a hint to the compiler on how to convert from **apAlloc<>** to a **T*** without having to explicitly specify it. See Section 8.2.2 on page 283.

It isn't always the right choice to define these conversion operators. We chose to use **operator T*** because **apAlloc<>** is fairly simple and is little more than a wrapper around a memory pointer. By simple, we mean that there is little confusion if the compiler were to use **operator T*** to convert the object reference to a pointer.

► EXAMPLE

For more complex objects, we could have used a **data()** method for accessing memory, which would look like:

```
T* data     () { return pData_;}
```

This means that we would have to explicitly ask for a memory pointer by using the **data()** method as follows:

```
apAllocatorBase_ a (...);
T* p1 = a;   // Requires operator T* to be defined.
T* p2 = a.data();
```

Note that the STL **string** class also chooses not to define conversion operators, but rather uses the **c_str()** and **data()** methods for directly accessing the memory. The STL purposely does not include implicit conversion operators to prevent the misuse of raw string pointers.

❖ REFERENCE COUNTING

Next we'll look at the functions that manage our reference count.

```
void addRef () { ref_++;}
void subRef () { if (--ref_ == 0) delete this;}
```

These methods are defined in the base class for convenience but are only used in the derived class. Whenever an **apAllocatorBase_<>** object is constructed or copied, **addRef()** is called to increment our reference count variable. Likewise, whenever an object is deleted or otherwise detached from the object, **subRef()** is called. When the reference count becomes zero, the object is deleted. Note that later we will modify the definitions for **addRef()** and **subRef()** to handle multi-threaded synchronization issues.

❖ MEMORY ALIGNMENT

Memory alignment is important because some applications might want more control over the pointer returned after memory is allocated. Most applications prefer to leave memory alignment to the compiler, letting it return whatever address it wants. We provide memory alignment capability in **apAllocatorBase_<>** so that derived classes can allocate memory on a specific boundary. On some platforms, this technique can be used to optimize performance. This is especially useful for imaging applications, because image processing algorithms can be optimized for particular memory alignments.

When a derived class allocates memory, it stores a pointer to that memory in **pRaw_**, which is defined as a **char***. Once the memory is aligned, the **char*** is cast to type **T*** and stored in **pData_**, which serves as a pointer to the aligned memory.

alignPointer() is defined in **apAllocatorBase_<>** to force a pointer to have a certain alignment. The implementation presented here is acceptable for single-threaded applications and is sufficient for our current needs. Later it will be extended to handle multi-threaded applications. Here is the final version of this implementation:

```
T* alignPointer (void* raw)
{
  T* p = reinterpret_cast<T*>(
    (reinterpret_cast<uintptr_t>(raw) + align_ - 1)
    & ~(align_ - 1));
  return p;
}
```

Here is how we arrived at this implementation: in order to perform the alignment arithmetic, we need to change the data type to some type of numeric value. The following statement accomplishes this by casting our **raw** pointer to a **uintptr_t** (a type large enough to hold a pointer):

```
reinterpret_cast<uintptr_t>(raw)
```

By subsequently casting the **raw** pointer to an **uintptr_t**, we're able to do the actual alignment arithmetic, as follows:

```
(reinterpret_cast<uintptr_t>(raw) + align_ - 1) & ~(align_ - 1);
```

Basically, we want to round up the address, if necessary, so that the address is of the desired alignment. The only way to guarantee a certain alignment is to allocate more bytes than necessary (in our case, we must allocate **(align_-1)** additional bytes). **alignPointer()** works for any alignment that is a power of two.

For example, if **align_** has the value of 4 (4-byte alignment), then the code would operate as follows:

```
(reinterpret_cast<uintptr_t>(raw) + 3) & ~3);
```

It now becomes clear that we are trying to round the address up to the next multiple of 4. If the address is already aligned on this boundary, this operation does not change the memory address; it merely wastes 3 bytes of memory. The only thing left is to cast this pointer back to the desired type. We use **reinterpret_cast<>** to accomplish this, since we must force a cast between pointer and integer data types.

Conversely, this example uses old C-style casting:

✗
```
T* p = (T*)(((uintptr_t)(raw + align_ - 1)) & ~(align_ - 1));
```

This is still legal, but it doesn't make it obvious what our casts are doing. Note that for systems that do not define the symbol **uintptr_t**, you can usually define your own as:

```
typedef int uintptr_t;
```

Stop using C-style casting and start using the C++ casting operators. reinterpret_cast<> and static_cast<> allow you to perform arbitrary casting and the casting operators make your intent clear. It is also easier to search for using an editor.

❖ COPY CONSTRUCTOR AND ASSIGNMENT

The only thing we haven't discussed from **apAllocatorBase_<>** is the copy constructor and assignment operator:

```
apAllocatorBase_            (const apAllocatorBase_& src);
apAllocatorBase_& operator= (const apAllocatorBase_& src);
// No copy or assignment is allowed.
```

We include a copy constructor and assignment operator, but don't provide an implementation for them. This causes the compiler to generate an error if either of these functions is ever called. This is intentional. These functions are not necessary, and worse, will cause our reference counting to break if the default versions were to run. Once an **apAllocatorBase_<>** object or derived object is created, we only reference them using pointers.

apAllocator_<> Class

The **apAllocator_<>** class, which is derived from **apAllocatorBase_<>**, handles heap-based allocation and deallocation. Its definition is shown here

```
template<class T> class apAllocator_ : public apAllocatorBase_<T>
{
public:
    explicit apAllocator_ (unsigned int n, unsigned int align = 0)
    : apAllocatorBase_<T> (n, align)
    {
        allocate ();
        addRef ();
    }

  virtual ~apAllocator_ () { deallocate();}

private:
    virtual void allocate () ;
  // Allocate our memory for size_ elements of type T with the
  // alignment specified by align_. 0 and 1 specify no alignment,
  // 2 = word alignment, 4 = 4-byte alignment, ... This must
  // be a power of 2.

    virtual void deallocate ();

  apAllocator_            (const apAllocator_& src);
  apAllocator_& operator= (const apAllocator_& src);
  // No copy or assignment is allowed.
};
```

Constructor and Destructor

The **apAllocator_<>** constructor handles memory allocation, memory alignment, and setting the initial reference count value. The destructor deletes the memory when the object is destroyed.

The implementation of the constructor is:

```
public:
    explicit apAllocator_ (unsigned int n, unsigned int align = 0)
    : apAllocatorBase_<T> (n, align)
    {
      allocate ();
      addRef ();
    }

    virtual ~apAllocator_ () { deallocate();}

    ...
```

```
private:
    apAllocator_             (const apAllocator_& src);
    apAllocator_& operator=  (const apAllocator_& src);
    // No copy or assignment is allowed.
```

❖ MEMORY ALIGNMENT

The **apAllocator_<>** constructor takes two arguments, a size parameter and an alignment parameter.

Although the alignment parameter is an **unsigned int**, it can only take certain values. A value of 0 or 1 indicates alignment on a byte boundary; in other words, no special alignment is needed. A value of 2 means that memory must be aligned on a word (i.e., 2-byte) boundary. A value of 4 means that memory must be aligned on a 4-byte boundary.

► EXAMPLE

Suppose we use operator **new** to allocate memory and we receive a memory pointer, **0x87654325**. This hexidecimal value indicates where storage was allocated for us in memory. For most applications, this address is fine for our needs. The compiler makes sure that the address is appropriate for the type of object we are allocating. Different alignment values will alter this memory address, as shown in Table 3.2.

Table 3.2: Effect of Alignment on Memory Address

Alignment	Memory Address
0 or 1	0x87654325
2	0x87654326
4	0x87654328
8	0x87654328

◄

❖ REFERENCE COUNTING

The constructor also calls **addRef()** directly. This means that the client code does not have to explicitly touch the reference count when an **apAllocator_<>** object is created. The reference count is set to 1 when the object is constructed.

What would happen if the client does call **addRef()**? This would break the reference counting scheme because the reference count would be 2 instead of 1. When the object is no longer used and the final instance of **apAllocator_<>** calls **subRef()**, the reference count would be decremented to 1 instead of to 0. We would end up with an object in heap memory that would never be freed.

Similarly, if we decided to leave out the call to **addRef()** in the constructor and force the client to call it explicitly, it could also lead to problems. If the client forgets to call **addRef()**, the reference count stays at zero. Our strategy is to make it very clear through the comments embedded in the code about what is responsible for updating the reference count.

❖ EXPLICIT KEYWORD USAGE

We use the **explicit** keyword in the constructor. This keyword prevents the compiler from using the constructor to perform an implicit conversion from type **unsigned int** to type **apAllocator_<>**. The **explicit** keyword can only be used with constructors that take a single argument, and our constructor has two arguments. Or does it? Since most users do not care about memory alignment, the second constructor argument has a default value of 0 for alignment (i.e., perform no alignment). So, this constructor can look as if it has only a single argument (i.e., **apAllocator_<char> alloc (5);**).

Use the explicit keyword to eliminate the chance that single-argument constructors will be misused by the compiler for implicit conversions.

Allocation

The constructor calls the **allocate()** function to perform the actual memory allocation. The full implementation of this function is as follows:

```
protected:
  virtual void allocate ()
  {
    if (size_ == 0) {
      // Eliminate possibility of null pointers by allocating 1 item.
      pData_ = new T [1];
      pRaw_  = 0;
      return;
    }

    if (align_ < 2) {
      // Let the compiler worry about any alignment
      pData_ = new T [size_];
      pRaw_  = 0;
    }
    else {
      // Allocate additional bytes to guarantee alignment.
      // Then align and cast to our desired data type.
      pRaw_  = new char [sizeof(T) * size_ + (align_ - 1)];
      pData_ = alignPointer (pRaw_);
    }
  }
```

Our **allocate()** function has three cases it considers.

- The first is what to do when an allocation of zero bytes is requested. This is a very common programming concern, because we cannot allow the user to obtain a null pointer (or worse, an uninitialized pointer). We can eliminate all of this checking by simply allocating a single element of type **T**. By doing this, our definition for **operator T*** in the base class never has to check the pointer first. And, because, our **size()** method will return zero in this case, the client code can safely get a valid pointer that it presumably will never use.

- The next case that **allocate()** considers is when no memory alignment is desired. Since this is a common case, we bypass our alignment code and let the compiler decide how to allocate memory:

```
pData_ = new T [size_];
```

  Our function uses two pointers, **pData_** and **pRaw_**, to manage our heap allocation. **pData_** is a **T*** pointer, which references the aligned memory. **pRaw_** contains the pointer returned after calling **operator new**. Since we do not perform any alignment, we don't use the **pRaw_** pointer in this case, so we set this variable to **0**.

- Our final case in **allocate()** is when a specific kind of memory alignment is desired; for example, as it does when managing images using third-party imaging libraries. Many libraries require memory alignment to avoid the performance hit of copying images. We use the **pRaw_** pointer (defined as a **char***) for the allocation, and then align and coerce the pointer to be compatible with **pData_**:

```
pRaw_  = new char [sizeof(T) * size_ + (align_ - 1)];
pData_ = alignPointer (pRaw_);
```

  Our base class provides a function, **alignPointer()**, to handle the alignment and conversion to type **T***. We saw how this function can alter the memory address by up to **(align_-1)** bytes during alignment. For this reason, we must allocate an additional **(align_-1)** bytes when we make the allocation, to make sure we never access memory outside of our allocation.

Deallocation

Our **deallocate()** function must cope with the **pRaw_** and **pData_** pointers. Whenever we bypass performing our own memory alignment, the **pRaw_** variable will always be null. This is all our function needs to know to delete the appropriate pointer. The **deallocate()** definition is as follows:

```
virtual void deallocate ()
{
  // Decide which pointer we delete
  if (pRaw_)
    delete [] pRaw_;
  else
    delete [] pData_;
  pRaw_  = 0;
  pData_ = 0;
}
```

At the end of the **deallocate()** function, we set both the **pRaw_** and **pData_** variables to their initial state (**0**). Some developers will skip this step, because **deallocate()** is only called by the destructor, so in an attempt to be clever, a possible bug is introduced. Sometime in the future, the **deallocate()** function may also be used in other places, such

as during an assignment operation. In this case, the values of **pRaw_** and **pData_** will appear to point to valid memory.

Whenever memory is deallocated in any function other than the destructor, remember to set all memory pointers to 0.

apAlloc<> Class

apAlloc<> is our memory allocation object. This is the object that applications will use directly to allocate and manage memory. The definition is shown here.

```
template<class T, class A = apAllocator_<T> >
class apAlloc
{
public:
  static apAlloc& gNull ();
  // We return this object for any null allocations
  // It actually allocates 1 byte to make all the member
  // functions valid.

  apAlloc  ();
  // Null allocation. Returns pointer to gNull() memory

  explicit apAlloc  (unsigned int size, unsigned int align=0);
  ~apAlloc ();
  // Allocate the specified bytes, with the correct alignment.
  // 0 and 1 specify no alignment. 2 = word alignment,
  // 4 = 4-byte alignment. Must be a power of 2.

  apAlloc            (const apAlloc& src);
  apAlloc& operator= (const apAlloc& src);
  // We need our own copy constructor and assignment operator.

  unsigned int  size   () const { return pMem_->size ();}
  unsigned int  ref    () const { return pMem_->ref ();}
  bool          isNull () const { return (pMem_ == gNull().pMem_);}

  const T* data () const { return *pMem_;}
  T*       data ()       { return *pMem_;}
  // Access to the beginning of our memory region. Use sparingly

  const T& operator[] (unsigned int index) const;
  T&       operator[] (unsigned int index);
  // Access a specific element. Throws the STL range_error if
  // index is invalid.

  virtual A* clone ();
  // Duplicate the memory in the underlying apAllocator.

  void duplicate ();
  // Breaks any reference counting and forces this object to
  // have its own copy.

protected:
    A*   pMem_;   // Pointer to our allocated memory

  static apAlloc* sNull_; // Our null object
};
```

The syntax may look a little imposing because **apAlloc<>** has two template parameters: **template<class T, class A> class apAlloc**

The keywords class and typename are synonymous for template parameters. Use whichever keyword you are more comfortable with, but be consistent.

Parameter **T** specifies the unit of allocation. If we had only a single parameter, the meaning of it would be identical to the meaning for other STL types. For example,

```
vector<int> v;
apAlloc<int> a;
```

describe instances of a template whose unit of storage is an **int**.

Parameter **A** specifies how and where memory is allocated. It refers to another template object whose job is to allocate and delete memory, manage reference counting, and allow access to the underlying data. If we have an application that requires memory to be allocated differently, say a private memory heap, we would derive a specific **apAllocator_<>** object to do just that.

In our case, the second parameter, **A**, uses the default implementation **apAllocator_<>** to allocate memory from the heap. This allows us to write such statements as:

```
apAlloc<int> a1;          // Null allocation
apAlloc<int> a2 (100);    // 100 elements
apAlloc<int> a3 (100, 4); // 100 elements, 4-byte align
```

The null allocation, an allocation with no specified size, is of special interest because of how we implement it. We saw that the **apAllocator_<>** object supported null allocations by allocating one element. It is possible that many (even hundreds) of null **apAlloc<>** objects may be in existence. This wastes heap memory and causes heap fragmentation.

Our solution is to only ever have a single null object for each **apAlloc<>** instance. We do this in a manner similar to constructing a Singleton object. Singleton objects are typically used to create only a single instance of a given class. See [Gamma95] for a comprehensive description of the Singleton design pattern. We use a pointer, **sNull_**, and a **gNull()** method to accomplish this:

```
template<class T, class A>
apAlloc<T,A>* apAlloc<T, A>::sNull_ = 0;
```

This statement creates our **sNull_** pointer and sets it to null.

❖ GNULL() METHOD

The only way to access this pointer is through the **gNull()** method, whose implementation is shown here.

```
template<class T, class A>
apAlloc<T,A>& apAlloc<T, A>::gNull ()
```

```
{
  if (!sNull_)
    sNull_ = new apAlloc (0);
  return *sNull_;
}
```

The first time **gNull()** is called, **sNull_** is zero, so the single instance is allocated by calling **apAlloc()**. For this and subsequent calls, **gNull()** returns a reference to this object. A null **apAlloc<>** object is created by passing zero to the **apAlloc<>** constructor. This is a special case. When zero is passed to the **apAllocator_<>** object to do the allocation, a single element is actually created so that we never have to worry about null pointers. In this case, all null allocations refer to the same object. Note that this behavior differs from the C++ standard, which specifies that each null object is unique.

gNull() can be used directly, but its main use is to support the null constructor.

```
template<class T, class A>
apAlloc<T, A>::apAlloc () : pMem_ (0)
{
  pMem_ = gNull().pMem_;
  pMem_->addRef ();
}
```

apAlloc<> contains a pointer to our allocator object, **pMem_**. The constructor copies the pointer and tells the **pMem_** object to increase its reference count. The result is that any code that constructs a null **apAlloc<>** object will actually point to the same **gNull()** object. So, why didn't we just write the constructor as:

```
*this = gNull();
```

This statement assigns our object to use the same memory as that used by **gNull()**. We will discuss the copy constructor and assignment operator in a moment. The problem is that we are inside our constructor and the assignment assumes that the object is already constructed. On the surface it may look like a valid thing to do, but the assignment operator needs to access the object pointed to by **pMem_**, and this pointer is null.

Assignment Operator and Copy Constructor

The assignment operator and copy constructor are similar, so we will only look at the assignment operator here:

```
template<class T, class A>
apAlloc<T, A>& apAlloc<T, A>::operator= (const apAlloc& src)
{
  // Make sure we don't copy ourself!
  if (pMem_ == src.pMem_) return *this;

  // Remove reference from existing object. addRef() and subRef()
  // do not throw so we don't have to worry about catching an error
  pMem_->subRef ();  // Remove reference from existing object
  pMem_ = src.pMem_;
  pMem_->addRef ();  // Add reference to our new object

  return *this;
}
```

First, we must detach from whatever memory allocation we were using by calling **subRef()** on our allocated object. If this was the only object using the **apAllocator_<>** object, it would be deleted at this time. Next, we point to the same **pMem_** object that our **src** object is using, and increase its reference count. Because we never have to allocate new memory and copy the data, these operations are very fast.

Memory Access

Accessing memory is handled in two different ways. We provide both **const** and non-**const** versions of each to satisfy the needs of our clients. To access the pointer at the beginning of memory, we use:

```
T* data () { return *pMem_;}
```

When we discussed the **apAllocatorBase_<>** object, we talked about when it is appropriate to use **operator T*** and when a function like **data()** should instead be used. In **apAllocatorBase_<>**, we chose to use the operator syntax so that ***pMem_** will return a pointer to the start of our memory. In this **apAlloc<>** object, we use the **data()** method to grant access to memory, because we want clients to explicitly state their intention. **operator[]** prevents the user from accessing invalid data, as shown:

```
template<class T, class A>
T& apAlloc<T, A>::operator[] (unsigned int index)
{
  if (index >= size())
    throw std::range_error ("Index out of range");

  return *(data() + index);
}
```

Instead of creating a new exception type to throw, we reuse the **range_error** exception defined by the STL.

Object Duplication

The **duplicate()** method gives us the ability to duplicate an object, while letting both objects have separate copies of the underlying data. Suppose we have the following code:

```
apAlloc<int> alloc1(10);
apAlloc<int> alloc2 = alloc1;
```

Right now, these two objects point to the same underlying data. But what if we want to force them to use separate copies? This is the purpose of **duplicate()**, whose implementation is shown here:

```
template<class T, class A>
void apAlloc<T, A>::duplicate ()
{
  if (ref() == 1) return;  // No need to duplicate

  A* copy = clone ();
  pMem_->subRef ();  // Remove reference from existing object
  pMem_ = copy;      // Replace it with our duplicated data
}
```

Notice that this is very similar to our assignment operator, except that we use our **`clone()`** method to duplicate the actual memory. Instead of putting the copy functionality inside **`duplicate()`**, we place it inside **`clone()`** to allow clients to specify a custom **`clone()`** method for classes derived from **`apAlloc<>`**. This version of **`clone()`** does a shallow copy (meaning a bit-wise copy) on the underlying data:

```
template<class T, class A>
A* apAlloc<T, A>::clone ()
{
  A* copy = new A (pMem_->size(), pMem_->align());

  // Shallow copy
  T* src = *pMem_;
  T* dst = *copy;
  std::copy (src, &(src[pMem_->size()]), dst);
  return copy;
}
```

Under most conditions, a shallow copy is appropriate. We run into problems if **`T`** is a complex object or structure that includes pointers that cannot be copied by means of a shallow copy. Our image class is not affected by this issue because our images are constructed from basic types. Instead of using **`memcpy()`** to duplicate the data, we are using **`std::copy`** to demonstrate how it performs the same function.

3.2 Prototyping

Our strong recommendation is that any development plan should include some amount of time for prototyping. Prototyping has a number of advantages and can directly affect the success of a commercial software development effort. Our rules of thumb for prototyping are shown in Figure 3.5.

Prototyping Rules

- ✔ **Explore the hardest parts of the problem first in your prototypes.**
- ✔ **Use good coding practices in your prototypes.**
- ✔ **Document your prototypes even more heavily than actual production code, so that your design ideas are recorded.**
- ✔ **Experiment with different language elements to see how they affect the design and implementation.**
- ✔ **Take what you learn in one prototype and apply it to hone the next prototype.**
- ✔ **Write unit tests for your prototypes.**
- ✔ **Make sure that the compilers on all your platforms can predictably handle the prototypes and language constructs.**
- ✔ **Never distribute your prototypes as finished software products.**

Figure 3.5: Prototyping Rules

3.2.1 Why Prototyping Works

In our own commercial development efforts, we have consistently shown that by including prototyping as part of the process, the product is completed on time with full functionality. Why? Here are some of the reasons:

- **Early Visibility to Problems**. Prototypes help refine the design to produce the desired final product. More than that, prototyping is a necessary and important step in the design process. Errors can be caught during the prototyping stage instead of during actual development. Prototyping allows you to modify the design to avoid mistakes and it provides better visibility of what is required to complete the final product. Had you discovered the mistake during the development phase, it could negatively affect both the content of the product and the schedule for releasing the product.
- **Measurement of Performance and Code Size**. The intent of the prototype isn't to develop the product, but to develop ideas and a framework for the design. Good coding practices are as important here as they are for any other part of the design, including documentation and unit tests. Yes, unit tests. Otherwise, how else can you tell if the prototype is performing correctly? The unit test framework is also a good way to measure performance and code size. If a prototype is successful, it might be used as the basis for the real design. Prototypes only need to implement a small portion of the overall solution. By keeping the problem space limited, one or more features can be developed and tested in a very structured environment.
- **Assurance of Cross-Platform Compatibility**. Prototypes are also useful when the product must run on multiple platforms or on an embedded system. It might be obvious how something is designed on one platform, but the design may not work as well on others. This is especially true in the embedded platform world, because the problem is constrained by execution time and hardware resources (i.e., processor speed, memory, and the file system). Prototypes can also help decide which compiler(s) and version to use. The C++ standard library has evolved in recent years and compiler vendors are still trying to catch up. You should learn at the prototyping stage that your desired compiler will or will not work as planned. Once the compiler is chosen, you still must see if the included standard library will work, or whether a third-party version is needed.
- **Test Bed for Language Features**. Prototypes are also great for trying out new concepts or language features. This is especially true with the somewhat complex nature of templates. It is not always obvious how a final template object might look, or how it might interact with other objects. This is often found during the design phase, and some small prototypes can help guide the implementor. In our image class design, the use of templates is not necessarily obvious from the beginning.

3.2.2 Common Fears

There are a number of common fears and misconceptions about prototyping. We discuss some of the prevalent ones here.

One of the most common fears is that prototypes will be turned into the actual released software to save time and effort. This is especially true if your management is shown a prototype that looks like the desired final product. It can give the erroneous impression that the product is closer to completion than is actually the case. The problem with this scenario isn't that the prototype gave an incorrect impression, but rather that the expectations of management weren't properly set. Part of the development manager's role is to clearly set the expectations for any demonstration. Clearly explaining exactly what is being shown is part of that responsibility. If this is done well, management need not get a false impression.

Another common misconception about prototyping is that it will delay the actual design and implementation phases and result in making the product late. In actuality, prototyping is an iterative process with the design and implementation phases. By clarifying the most difficult aspects of the design, prototyping can actually result in avoiding costly mistakes and bringing the product in on time.

3.2.3 Our Image Framework Prototyping Strategy

In Chapter 2, we showed our first attempt to design an image object to handle the simple problem of generating a thumbnail image. We presented this test application to show how easy it is to design an object that solves a simple problem. Not unexpectedly, this application is totally inadequate for solving real-world problems and certainly unsuitable for our image framework. In this section, we use the prototyping strategy shown in Figure 3.6 to look at various aspects of image objects, to ensure that our image framework meets the standards of commercial software.

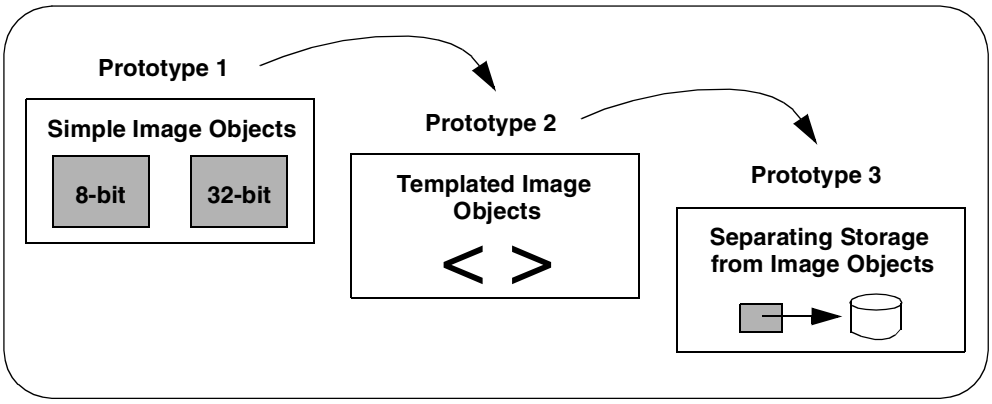


Figure 3.6: Image Framework Prototyping Strategy

We have chosen a prototyping strategy that lets us investigate three different aspects of the problem. In Prototype 1, we look at the common elements of images with different pixel types (8-bit versus 32-bit) to help us create a cleaner design. In Prototype 2, we explore whether using templates is a better way to handle the similarities between images with different pixel types. Once we started exploring templates, it became clear that there are

both image data and image storage components. In Prototype 3, we investigate the feasibility of separating these components in our design.

3.2.4 Prototype 1: Simple Image Objects

Prototype 1 is designed to explore the similarities between images of different pixel types. Remember that our test application defined a pixel as an **`unsigned char`** to specify monochrome images with 256 levels of gray (this is frequently called an 8-bit image, or an 8-bit grayscale image). 8-bit images are still very popular in applications for security, medical imaging, and machine vision, but they represent just one of many image formats. Other popular monochrome formats define pixels with depths of 16 bits and 32 bits per pixel. The larger the pixel depth, the more grayscale information is contained. And while 8 bits may be sufficient for a security application, 16 bits might be necessary for an astronomical application. In some cases, a monochrome image sensor may produce images that contain 10 bits or 12 bits of information. However, images of these odd depths are treated as 16-bit images with the higher-order bits set to zero.

Prototype 1 explores two types of monochrome images: an 8-bit image like our test application, and a 32-bit image, as shown in Figure 3.7.

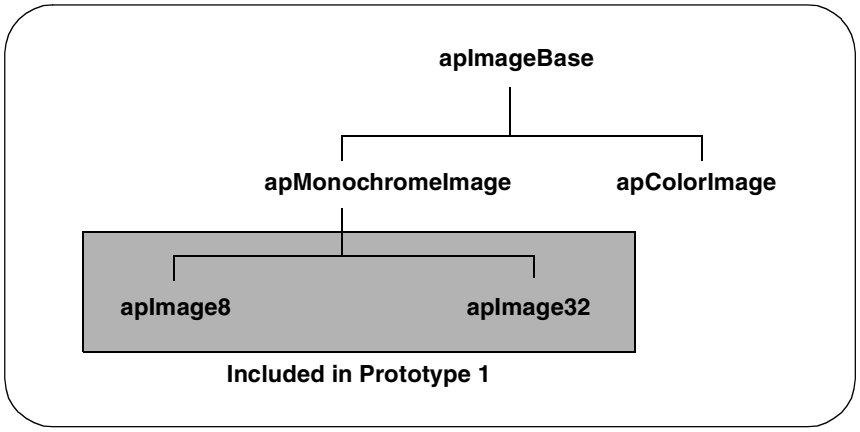


Figure 3.7: Image Object Strategy Used in Prototype 1

To be precise, our 8-bit image is represented by an **`unsigned char`**, and our 32-bit image is represented by an **`unsigned int`**. Each prototype object defines a simple class to create an image and supports one image processing operation, such as creating a thumbnail image.

Like any good prototyping strategy, we keep features from the test application that worked and add new features, as shown in Table 3.3.

Table 3.3: Features Reused and Added to Prototype 1

Reused Features	New Features
Simple construction	Uses **`apAlloc<>`**
Access to pixel data via **`getPixel()`** and **`setPixel()`**	Access to pixel data via **`pixels()`** for faster access
	`thumbnail` is a member function instead of a global function

The definition for **`apImage8`** is shown here.

```
typedef unsigned char Pel8;

class apImage8
{
public:
  apImage8 ();
  apImage8 (int width, int height);
  // Creates a null image, or the specified size

  virtual ~apImage8 ();

  int width  () const { return width_;}
  int height () const { return height_;}

  const Pel8* pixels () const { return pixels_.data();}
  Pel8*       pixels ()       { return pixels_.data();}
  // Return pointer to start of pixel data

  Pel8 getPixel (int x, int y) const;
  void setPixel (int x, int y, Pel8 pixel);
  // Get/set a single pixel

  // Image operations (only one for the prototype)
  virtual apImage8 thumbnail (int reduction) const;

  // Default copy ctor and assignment are ok
protected:
  apAlloc<Pel8> pixels_; // Pixel data
  int           width_;  // Image dimensions
  int           height_;
};
```

The definition for **`apImage32`** is shown here.

```
typedef unsigned int Pel32;

class apImage32
{
public:
  apImage32 ();
  apImage32 (int width, int height);
  // Creates a null image, or the specified size

  virtual ~apImage32 ();
```

```
  int width  () const { return width_;}
  int height () const { return height_;}

  const Pel32* pixels () const { return pixels_.data();}
  Pel32*       pixels ()       { return pixels_.data();}
  // Return pointer to start of pixel data

  Pel32 getPixel (int x, int y) const;
  void setPixel (int x, int y, Pel32 pixel);
  // Get/set a single pixel

  // Image operations (only one for the prototype)
  virtual apImage32 thumbnail (int reduction) const;

  // Default copy ctor and assignment are ok
protected:
  apAlloc<Pel32> pixels_; // Pixel data
  int            width_;  // Image dimensions
  int            height_;
};
```

The first thing to notice about the **apImage8** and **apImage32** objects is that a **typedef** is used to define the pixel type. It not only offers a convenient shorthand, but it is clear in our code when we deal with pixels. And before anyone asks, we did think of calling them **apPel8** and **apPel32** instead of just **Pel8** and **Pel32**, but we resisted. You also might notice that we are using an **int** to represent the width and height of the image. Some might argue that this should be an **unsigned int**, but for a prototype, we feel it is acceptable to make things simpler.

When you ignore the image references, **apImage8** and **apImage32** are little more than a wrapper around **apAlloc<>**. This is clear when you look at a few functions:

```
apImage8::apImage8 () : width_ (0), height_ (0) {}

apImage8::apImage8 (int width, int height)
: width_ (width), height_ (height)
{ pixels_ = apAlloc<Pel8> (width*height);}

apImage8::~apImage8 () {}

Pel8 apImage8::getPixel (int x, int y) const
{ return pixels_[y*width_ + x];}

void apImage8::setPixel (int x, int y, Pel8 pixel)
{ pixels_[y*width_ + x] = pixel;}
```

Our prototype may not break any new ground when it comes to image processing, but it already shows the benefits of using our memory allocation object. Our **apImage8** constructor makes **apAlloc<>** do all the work and our calls to **getPixel()** and **setPixel()** use **operator[]** of **apAlloc<>**.

❖ COPY CONSTRUCTOR

It gets even better. We don't define a copy constructor or assignment operator, because the default version works fine. If this isn't clear, look at what our copy constructor would look

like if we wrote one:

```
apImage8::apImage8 (const apImage8& src)
{
  pixels_ = src.pixels_;
  width_  = src.width_;
  height_ = src.height_;
}
```

pixels_ is an instance of **apAlloc<>** and **width_** and **height_** are just simple types. Since **apAlloc<>** has its own copy constructor we just let the compiler take care of this for us.

The **thumbnail()** method performs the same function as in our test application; however, its implementation is much cleaner. The output thumbnail image is created like any local variable, and returned at the end of the function. We saw how simple the copy constructor is, so even if the compiler creates some temporary copies, the overhead is extremely low. When we wrote the **thumbnail()** definition this time, we were careful with our naming convention. Variables **x** and **y** refer to coordinates in the original image and **tx** and **ty** refer to coordinates in the thumbnail image. So even though there are 4 nested loops, the code is still fairly easy to follow. The **thumbnail()** method is as follows:

❖ THUMBNAIL() METHOD

```
apImage8 apImage8::thumbnail (unsigned int reduction) const
{
  apImage8 output (width()/reduction, height()/reduction);

  for (unsigned int ty=0; ty<output.height(); ty++) {
    for (unsigned int tx=0; tx<output.width(); tx++) {
      unsigned int sum = 0;
      for (unsigned int y=0; y<reduction; y++) {
        for (unsigned int x=0; x<reduction; x++)
          sum += getPixel (tx*reduction+x, ty*reduction+y);
      }
      output.setPixel (tx, ty, sum / (reduction*reduction));
    }
  }

  return output;
}
```

If you compare the code for **apImage8** and **apImage32**, you find that they are almost identical. This is no great surprise, but the prototype shows this very clearly. This similarity leads to two thoughts. The first (historically) is to see how derivation can help simplify our design and maximize code reuse. The second is to see how templates can remove all of this duplicate code.

► EXAMPLE

Before templates were readily available, the image design could have been (and often actually was) handled by deriving each image from a common base class. As Figure 3.7 on page 47 indicates, we could derive our **apImage8** object from an **apMonochromeImage** object, which itself could be derived from **apImageBase**. Color images could also be handled by this framework by deriving from an **apColorImage** class. Although we aren't

going to present a full solution for this type of framework, let's look at one of the issues that would arise by taking a look at the **thumbnail()** definition:

```
class apImageBase
{
public:
  ...
  virtual apImageBase thumbnail (unsigned int reduction) const;
  ...
protected:
  apAlloc<Pel8> pixels_;
  int           type_;
  int           width_;
  int           height_;
};
```

You can see there is a new variable, **type_**, which is used to track what kind of image this is. This might be just the pixel depth of the image, an enumeration, or any other unique value to specify the image. The variables **width_** and **height_** have the same definition as in our prototype, but now **pixels_** is always defined as a buffer of **Pel8**s. This is not necessarily bad, although it means that **pixels_** must be cast to different types in derived classes. And these casts should exist in only a single place to keep everything maintainable. Our **thumbnail()** function returns an **apImageBase** object, not the type of the actual image computed in the derived class. This is a common issue and is discussed at length in other books. See [Meyers98]. As this example illustrates, it takes a bit of work, but you can construct a framework that does hold together.

Moving forward, we want to investigate the use of templates in our next prototypes to figure out how the final image class should be implemented.

Summary of Lessons Learned

These are the things that worked well:

- Using **apAlloc<>** helped us eliminate our copy constructor and assignment operator, made worrying about temporary images unimportant, and greatly improved the readability of the code.
- By extending the test application to explore two different image classes, we observed that the implementations are very similar. This similarity seems to lend itself to a derivation class design, or perhaps the use of templates. It is something we need to explore in future prototypes.

3.2.5 Prototype 2: Templated Image Objects

In Prototype 1, we extended our test application to show that image objects of different types are actually very similar. Derivation is one possible option for handling this similarity, but it forces all images into a single class hierarchy. Another way to handle this similarity is by the use of templates, with the goal of simplifying the design and maximizing the amount of code reuse.

We use Prototype 2 to investigate a number of new features:

- We use templates and rewrite the image class, **apImage**, to take a template parameter **T** that represents the pixel type.
- We introduce a handle class idiom so that many **apImage<>** handle objects can share the same underlying **apImageRep<>** representation object.
- We verify that our design works with more complex image types, such as an RGB image.

Use of Templates

Foremost in this prototype is the need to verify that a template object is the correct representation to solve our problem. Our test application only handled an 8-bit monochrome image (i.e., **unsigned char**), and Prototype 1 added a 32-bit monochrome image (i.e., **unsigned int**). Due to the similarity of the **apImage8** and **apImage32** objects, it makes sense to turn it directly into a template object, as shown in Figure 3.8:

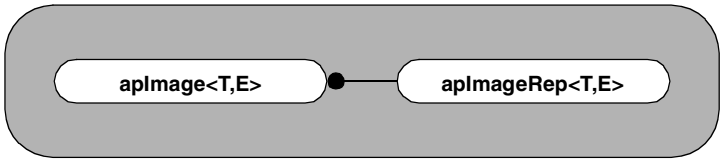


Figure 3.8: Templated Image Object Design

In Prototype 2, we introduce the *handle class idiom*, where there is a representation class that contains the data and performs all the operations, and a handle class that is a pointer to the representation class. In our prototype, **apImageRep<>** is the representation class to which the handle class **apImage<>** points.

We begin Prototype 2 by looking at the relevant parts of our **apImage<>** object from Prototype 1. It is not always clear from the outset how the prototype will be completed. Converting the **apImage<>** object into a template object gives us:

```
template<class T> class apImage
{
public:
  apImage  ();
  apImage  (unsigned int width, unsigned int height);
  ~apImage ();

  const T* pixels () const;
  T*       pixels ();

  T getPixel (unsigned int x, unsigned int y) const;
  void setPixel (unsigned int x, unsigned int y, T pixel);

  apImage<T> thumbnail (int reduction) const;
protected:
  apAlloc<T>   pixels_;
  unsigned int width_;
  unsigned int height_;
};
```

This certainly is very tidy, and with a small addition, we can use this object as a replacement for both **apImage8** and **apImage32**:

```
typedef apImage<unsigned char> apImage8;
typedef apImage<unsigned int>  apImage32;
```

Are we done? Unfortunately, templates are not always this simple. The implementation does not work correctly for **apImage8**. Let's look at the following example to understand why.

► EXAMPLE

Here is the original definition of **thumbnail()** from **apImage8**:

```
apImage8 apImage8::thumbnail (unsigned int reduction) const
{
  apImage8 output (width()/reduction, height()/reduction);

  for (unsigned int ty=0; ty<output.height(); ty++) {
    for (unsigned int tx=0; tx<output.width(); tx++) {
      unsigned int sum = 0;
      for (unsigned int y=0; y<reduction; y++) {
        for (unsigned int x=0; x<reduction; x++)
          sum += getPixel (tx*reduction+x, ty*reduction+y);
      }
      output.setPixel (tx, ty, sum / (reduction*reduction));
    }
  }
  return output;
}
```

❖ TEMPLATE CONVERSION

This is easily converted to a template function as shown:

✗
```
template<class T>
apImage<T> apImage<T>::thumbnail (unsigned int reduction) const
{
  apImage<T> output (width()/reduction, height()/reduction);

  for (unsigned int ty=0; ty<output.height(); ty++) {
    for (unsigned int tx=0; tx<output.width(); tx++) {
      T sum = 0;
      for (unsigned int y=0; y<reduction; y++) {
        for (unsigned int x=0; x<reduction; x++)
          sum += getPixel (tx*reduction+x, ty*reduction+y);
      }
      output.setPixel (tx, ty, sum / (reduction*reduction));
    }
  }
  return output;
}
```

It is still not obvious why it won't work correctly, until you study two lines from this function:

```
T sum = 0;
...
sum += getPixel (tx*reduction+x, ty*reduction+y);
```

If **T** is an **unsigned char**, the compiler sees this:

```
unsigned char sum = 0;
...
sum += getPixel (tx*reduction+x, ty*reduction+y);
```

The variable **sum** must have enough precision to contain the sum of many pixels. For example, if the reduction factor is 2, **sum** must be able to hold the summation of four pixels without overflowing. This condition is not true if **sum** is only an **unsigned char**. Worse, the compiler will happily accept this code without generating any errors. It is up to you and your unit tests to catch them. You might be tempted to write our first line as:

```
unsigned int sum = 0;
```

This fixes the problem for **unsigned char**, and probably works well with **unsigned int** (since images that are represented with 32 bits most likely have fewer significant bits). This fix (really more of a hack), however, does not work for many other pixel types, like **float** or **RGB**.

As shown in Figure 3.8 on page 52, Prototype 2 defines **apImage<T,E>**, which has two template arguments; the second argument is how we solve the thorny issue we just discussed. The first argument, **T**, is still the pixel type, but **E** now represents the internal pixel type to use during computation. For example, the definition **apImage<unsigned char, unsigned int>** describes an image of 8-bit pixels, but uses a 32-bit pixel for internal computations when necessary. Here are some other examples:

```
typedef unsigned char Pel8;
typedef unsigned int  Pel32;

apImage<Pel8, Pel8>;  // Watch out, round off is a real possibility
apImage<Pel8, Pel32>;
apImage<Pel32, Pel32>;
```

Now for an interesting design issue. Templates do support default arguments, so we can define our prototype class as either:

```
template<class T, class E> class apImage
```

or

```
template<class T, class E = T> class apImage
```

The difference may look small, but you must consider what happens when people forget the second argument. Default arguments are best used when the developer can predict what the argument is most of the time. One would expect that the template **apImage<Pel8,Pel32>** is used more often than **apImage<Pel8,Pel8>**. But if someone writes **apImage<Pel8>** they are getting the less commonly used object. For this reason, we do not supply a default argument.

Make sure the default argument is what should be used most of the time when deciding whether to supply one for your template class.

Handle Class Idiom

The handle class idiom has been used as long as C++ has been around. See [Coplien92]. This is little more than reference counting attached to an object. It is commonplace to call the shared object the *representation object*, and to call the objects that point to them the *handle objects*. The representation class (or rep class, as it is sometimes called) contains the implementation, does all the work, and contains all the data, while the handle class is little more than a pointer to a rep class. A more in-depth discussion can be found in Stroustrup's *The C++ Programming Language, Special Edition*, *Section 25.7*. See [Stroustrup00].

❖ HANDLE OBJECT

In Prototype 2, **`apImage<T,E>`** is our handle object and points to an instance of **`apImageRep<T,E>`**, as shown here.

```
template<class T, class E> class apImageRep; // Forward declaration

template<class T, class E> class apImage
{
public:
  friend class apImageRep<T, E>;

  apImage (); // A null image, suitable for later assignment
  apImage (unsigned int width, unsigned int height);
  ~apImage () { image_->subRef ();}

  apImage            (const apImage& src);
  apImage& operator= (const apImage& src);
  // We need our own copy constructor and assignment operator.

  const apImageRep<T, E>* operator -> () const { return image_;}
  apImageRep<T, E>*       operator -> ()       { return image_;}
  // Allow access to the rep object

protected:
  apImage (apImageRep<T, E>* rep);
  // Construct an image from a rep instance

  apImageRep<T, E>* image_; // The actual image data
};
```

The implementation of **`apImage<T,E>`** is similar to our **`apAlloc<T>`** object, since both use reference counting. If you study the implementation, you will see it all comes down to carefully calling **`addRef()`** and **`subRef()`** in the **`apImageRep<T,E>`** object. Besides the obvious constructor/destructor definitions, the most important function is **`operator->`**. This is the crux of the object, and is how you access a method in the rep class. This operator returns a pointer to the **`apImageRep<T,E>`** object, so any public method can be accessed.

❖ REP CLASS OBJECT

The rep class **apImageRep<T,E>** object, which is shown here, is very similar to the templated image object shown earlier on page 52.

```
template<class T, class E> class apImageRep
{
public:
  static apImageRep* gNull ();  // A null image

  apImageRep () : width_ (0), height_ (0), ref_ (0) {}
  // Creates a null image, suitable for later assignment

  apImageRep (unsigned int width, unsigned int height);
  ~apImageRep () {}

  unsigned int width  () const { return width_;}
  unsigned int height () const { return height_;}

  const T* pixels () const { return pixels_.data();}
  T*       pixels ()       { return pixels_.data();}

  const T& getPixel (unsigned int x, unsigned int y) const;
  void     setPixel (unsigned int x, unsigned int y, const T& pixel);

  // Reference counting related
  unsigned int ref () const { return ref_;}   // Number of references
  void addRef () { ref_++;}
  void subRef () { if (--ref_ == 0) delete this;}

  apImage<T, E> thumbnail (int reduction) const;

  // Default copy ctor and assignment are ok
protected:
  apAlloc<T>   pixels_; // Pixel data
  unsigned int width_;  // Image dimensions
  unsigned int height_;
  unsigned int ref_;    // Reference count

  static apImageRep* sNull_;   // Our null image object
};
```

The first difference between the rep object and the image object (from page 52) is in the definition of the null image. In the rep object, we use a Singleton method, **gNull()**, to be our null object. We define **gNull()** as:

```
template<class T, class E>
apImageRep<T,E>* apImageRep<T,E>::gNull ()
{
  if (!sNull_) {
    sNull_ = new apImageRep (0, 0);
    sNull_->addRef ();  // We never want to delete the null image
  }
  return sNull_;
}
```

This is the same behavior as our **apAlloc<T>** object. When we attempt to allocate an object with zero elements, we actually return an object that allocates a single element. Look at

what could happen if we did not define a **gNull()** object:

```
apImage<Pel8,Pel32> image;
if (image->width() == 0)
  // Null object
```

This would fail if **apImage<T,E>** contained a null pointer to the **apImageRep<T,E>** object, and we dereferenced it to get the width. An alternate, and less desirable, approach is to define an **isNull()** method to test if the pointer is null before using it, as shown:

```
apImage<Pel8,Pel32> image;
if (!image.isNull())
  // OK to use operator->
```

Null images are commonplace in applications. For example, an image operation that cannot produce a resulting image returns a null image. To eliminate the need to create many null rep images, we only need to allocate a single **gNull()**. By calling **addRef()** when the null image is created, we ensure that this object never gets deleted.

The complete source for **apImage<T,E>** can be found on the CD-ROM. Let's look at one of the constructors to reinforce that this object is little more than a wrapper:

```
template<class T, class E>
apImage<T,E>::apImage (unsigned int width, unsigned int height)
  : image_ (0)
{
  image_ = new apImageRep<T,E> (width, height);
  image_->addRef ();
}
```

❖ THUMBNAIL() METHOD

The **thumbnail()** method of **apImageRep<T,E>** now looks like this:

```
template<class T, class E>
apImage<T,E> apImageRep<T,E>::thumbnail (unsigned int reduction)
   const
{
  apImageRep<T,E>* output =
    new apImageRep<T,E> (width()/reduction,
                         height()/reduction);

  for (unsigned int ty=0; ty<output->height(); ty++) {
    for (unsigned int tx=0; tx<output->width(); tx++) {
      E sum = 0;
      for (unsigned int y=0; y<reduction; y++) {
        for (unsigned int x=0; x<reduction; x++)
          sum += getPixel (tx*reduction+x, ty*reduction+y);
      }
      output->setPixel (tx, ty, sum / (reduction*reduction));
    }
  }
  // Convert to apImage via the protected constructor
  return output;
}
```

This approach differs from our first attempt at converting **thumbnail()** to a template function (as shown on page 53). Rep classes are allocated on the heap and our handle classes

can be allocated anywhere. For this reason, we must use **new** to allocate our resulting object **output**. Although our **thumbnail()** method returns an **apImage<T,E>** object, there is no explicit reference to one in the function. We did this to avoid mixing references to **apImage<T,E>** and **apImageRep<T,E>**. We end the function by executing:

```
return output;
```

The compiler converts this object into an **apImage<T,E>** object, using the protected constructor:

```
apImage (apImageRep<T,E>* rep);
```

Our handle definition is looser than some implementations, although it is sufficient for our prototype. For example, there is nothing to stop someone from creating **apImageRep<T,E>** objects directly. It is a matter of opinion as to whether this is a feature or a detriment. It highlights the fact that it is not always clear what functionality is needed when prototyping.

RGB Images

So far our prototypes have dealt with monochrome images. Our design has gotten sufficiently complex that we should look at other image types to make sure our implementation and design are appropriate. We will do that now by looking at Red-Green-Blue (RGB) images. Depending on the application, color images may be more prevalent than monochrome images. Regardless of the file format used to store color images, they are usually represented by three independent values in memory. RGB is the most common, and uses the three colors red, green, and blue to describe a color pixel. There are many other representations that can be used, but each uses three independent values to describe an image.

An RGB image can be defined as:

✗
```
typedef unsigned char Pel8;
struct RGB {
  Pel8 red;
  Pel8 green;
  Pel8 blue;
};
```

With this simple definition, will a statement like the following work?

```
apImage<RGB> image;
```

The answer is no. Although we defined an RGB image, we did not define any operations for it. For example, the **thumbnail()** method needs to be able to perform the following operations with an RGB image:

- Construction from a constant (**E sum = 0**)
- Summing many pixel values (**sum += getPixel (...)**)
- Computing the output pixel (**sum / (reduction*reduction)**)

Adding support for RGB images entails defining these operations. The compiler will always tell you when you are missing a function, although some of the error messages are somewhat cryptic.

While we are adding functions for an **RGB** data type, we also need to define an **RGBPel32** type so that we don't have the same overflow issue we discussed earlier. **RGBPel32** is identical to **RGB**, except that it contains three 32-bit values, rather than three 8-bit values. At a minimum, we need to define these functions:

```
// Our basic color data type (8:8:8 format)
struct RGB {
  Pel8 red;
  Pel8 green;
  Pel8 blue;

  RGB (Pel8 b=0) : red (b), green (b), blue (b) {}
};

// Internal definition during computation (32:32:32 format)
struct RGBPel32 {
  Pel32 red;
  Pel32 green;
  Pel32 blue;

  RGBPel32 (Pel32 l=0) : red (l), green (l), blue (l) {}
};

RGBPel32& operator += (RGBPel32& s1, const RGB& s2)
{
  s1.red += s2.red;
  s1.green += s2.green;
  s1.blue += s2.blue;
  return s1;
}

RGB operator/ (const RGBPel32& s1, int den)
{
  RGB div;
  div.red = s1.red / den;
  div.green = s1.green / den;
  div.blue = s1.blue / den;
  return div;
}
```

Now, we are able to define an RGB image:

```
apImage<RGB,RGBPel32> image;
```

and even write a simple application:

```
apImage<RGB, RGBPel32> p (32, 32);

// Initialize the image with some data
RGB pel;
pel.red = pel.green = pel.blue = 0;
for (y=0; y<p->height(); y++)
  for (x=0; x<p->width(); x++) {
    p->setPixel (x, y, pel);
```

```
        pel.red++;
        pel.green++;
        pel.blue++;
      }
  apImage<RGB, RGBPel32> thumbnail = p->thumbnail (2);
```

To run any real applications, we also need to define additional functions that operate on **RGB** and **RGBPel32** images. For example, the unit test we wrote for this prototype adds a few more functions to initialize and test the value of RGB pixels:

```
RGBPel32 operator+ (const RGB& s1, const RGB& s2);
RGBPel32 operator+ (const RGBPel32& s1, const RGB& s2);
bool operator==    (const RGB& s1, const RGB& s2);
```

Summary of Lessons Learned

These are the things that worked well:

- Using templates to handle different types of images took advantage of the implementation similarities and resulted in an efficient design. The use of templates is something we will keep in the next prototype when we explore separating image storage from the image object .
- Defining an RGB image type was a good way to validate that the design is flexible and can handle many image types cleanly.

Here is what did not work well:

- Using the handle idiom did not provide any obvious advantages for the design. We had hoped that reference counting in our **apAlloc<T>** class, in conjunction with our **apImageRep<T,E>** class, would simplify the design, but it didn't. We are going to reuse the handle idiom in our next prototype, to see if it makes a difference when we separate storage from the image object.

3.2.6 Prototype 3: Separating Storage from Image Objects

In Prototype 2, we introduced a handle class idiom. We put all the functionality in the rep class and used a simple handle class to access the methods in the rep class. The light-weight handle, **apImage<>**, has simple semantics, but we still ended up with a fairly complicated rep class. Since the handle class cannot be used without understanding the functionality in the rep class, we didn't meet our goal of simplifying our code.

We need to intelligently manage large blocks of memory and allow access to a potentially large set of image processing functions. Strictly speaking, the use of the **apAlloc<>** class is what manages our image memory.

So what are the advantages of keeping our handle class? It does offer an insulation layer between our client object and the object that does all the work. However, we also need another construct to hold all of the data for storing and describing the image data. The

image data, such as the pixels and image dimensions, is contained within the **apImageRep<>** object, as shown:

```
apAlloc<T>   pixels_; // Pixel data
unsigned int width_;  // Image dimensions
unsigned int height_;
```

Given that our final image class will contain even more data, such as the image origin, creating an object that combines all of this data makes even more sense. Another advantage of this strategy is that, given the appropriate design, this object can be used by many different image classes. This allows applications to customize the front end and reuse the image storage object. We use Prototype 3 to separate the image storage from the actual image object by extending the handle idiom we introduced in Prototype 2.

Design for Partitioning Image Storage

The purpose of Prototype 3 is to separate the image storage from the image processing. To accomplish this, we create a class, **apStorageRep**, to encapsulate the image storage, and we define **apImage<>** to be the object that performs the image processing. We connect the two using a handle class, **apImageStorage**. The final design for Prototype 3 is shown in Figure 3.9.

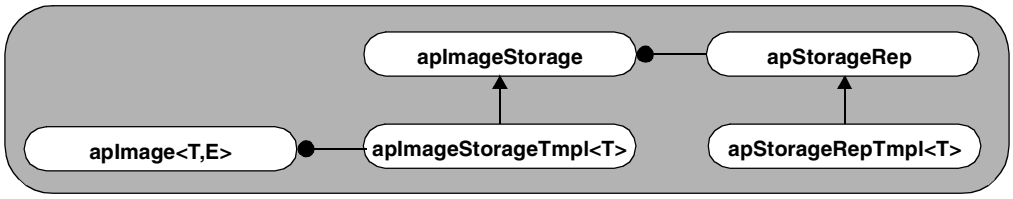


Figure 3.9: Image Object and Storage Separation Design

Note that we have introduced a new naming convention here to make it clear how these objects are related. Finding good names for closely related objects is not always easy. The C++ language does not support a class named **apImageStorage** and one called **apImageStorage<>**. The **Tmpl** suffix is added, renaming **apImageStorage<>** to **apImageStorageTmpl<T>**, to make it clear this is a templated class.

Evolution of the Design

Let's look at how we arrived at this design. Our first attempt to show how our objects are related is by extending Prototype 2, as shown in Figure 3.10.

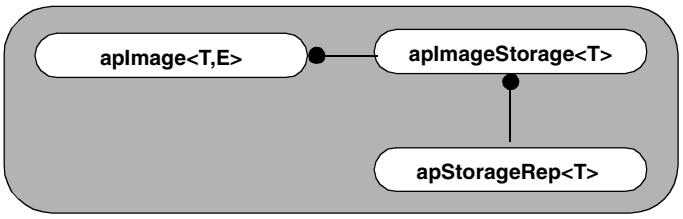


Figure 3.10: Evolution of Design (Step 1)

Although there is no inheritance in this example, the handle (**apImageStorage<T>**) and rep (**apStorageRep<T>**) classes are very closely related. Stacking them vertically in Figure 3.10 helps to show this relationship and clarifies that **apImage<T,E>** is related to the other two classes.

If you decide to graphically depict your object relationships, take advantage of both axes to represent them.

❖ COMMON BASE CLASS

Before we start writing code, let us take one more step in improving our design. The compiler will instantiate a template object for each pixel type. Even if your users only need limited types, the image processing routines may need additional types for temporary images and the like. We have not worried about code bloat issues in our prototype, but now we need to consider how to handle them.

Memory is nothing more than a series of bytes that the user's code can access and treat as other data types. This is pretty much what **new** does. It retrieves memory from the heap (or elsewhere, if you define your own **new** operator) and returns it to the user. When coding with C, it was customary to call **malloc()**, then cast the returned pointer to the desired data type.

There is no reason why we cannot take a similar approach and handle allocation with a generic object. Our code will perform all the allocation in a base class, **apStorageRep**, and perform all other operations through a derived template class, **apStorageRepTmpl<T>**, as shown in Figure 3.11.

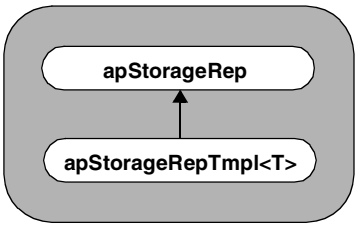


Figure 3.11: Evolution of Design (Step 2)

Image Storage Rep Objects

Given the handle idiom we have decided to use, our rep object, **apStorageRep,** will contain generic definitions for image storage, as well as the handle-specific functionality, as shown:

```
class apStorageRep
{

public:
  static apStorageRep* gNull ();
  // Representation of a null image storage.

  apStorageRep ();
  apStorageRep (unsigned int width, unsigned int height,
                unsigned int bytesPerPixel);

  virtual ~apStorageRep ();

  const unsigned char* base () const { return storage_.data();}
  unsigned char*       base ()       { return storage_.data();}
  // Access to base of memory

  unsigned int width         () const { return width_;}
  unsigned int height        () const { return height_;}
  unsigned int bytesPerPixel () const
    { return bytesPerPixel_;}
  unsigned int ref           () const { return ref_;}

  void addRef () { ref_++;}
  void subRef () { if (--ref_ == 0) delete this;}
  // Increment or decrement the reference count

  // Default copy constructor and assignment operators ok.
protected:
  apAlloc<unsigned char> storage_; // Pixel storage

  unsigned int bytesPerPixel_; // Bytes per pixel
  unsigned int width_;
  unsigned int height_;
  unsigned int ref_;           // Current reference count

  static apStorageRep* sNull_;
};
```

If you compare **apStorageRep** above with **apImageRep<>** from Prototype 2, you will find that they both offer the same functionality. The main difference is that **apImageRep<>** allocates memory as **apAlloc<T>**, where **T** is the data type, and **apStorageRep** allocates memory as **apAlloc<unsigned char>**.

Since **apStorageRep** is not a template class, let's look specifically at some of the differences, as shown:

```
apAlloc<unsigned char> storage_;
unsigned int bytesPerPixel_;
```

storage_ is defined in terms of bytes. When an object derived from **apStorageRep** wants to allocate storage, it must supply not only the width and height of the image, but also its *depth*. By depth, we mean the number of bytes it takes to store each pixel. Note that our definition does not support packed data. For example, a binary image where each pixel can be represented by a single bit still consumes the same amount of memory as an 8-bit image. This limitation is not very severe, since images often are aligned on byte boundaries by definition. And those that are not, for example 12-bit images, most likely require 16 bits of

storage and fall on byte boundaries anyway. In any event, we are not going to worry about this special case now.

Once an image is constructed, we allow access to all the parameters of the object, as well as the memory pointer (as an **unsigned char**). We chose to leave the **base()** method public to allow for future functionality. We continue to use the **gNull()** definition, so we only have one instance of a null image (remember, this is an image that has a valid pointer, but has a **width()** and **height()** of zero).

Since **apStorageRep** is not a templated object, we put its definition in a header file and put much of its implementation in a source file (**.cpp** in our case). Not having it be a templated object has the advantage of giving us control of how the object will be compiled, and lets us control code bloat. For example, if this were a templated object, compilers would expect classes to be defined in a single translation unit (or by means of nested include files). This gives the decision of what to inline, and what not to, to the compiler.

By putting most of the functionality into the base class, we can have a very simple templated class, **apStorageRepTmpl<>**, that redefines **base()** to return a **T*** pointer (which matches the pixel type) instead of an **unsigned char***, as shown here.

```
template<class T>
class apStorageRepTmpl : public apStorageRep
{
public:
  apStorageRepTmpl () {}
  apStorageRepTmpl (unsigned int width, unsigned int height)
    : apStorageRep (width, height, sizeof (T)) {}

  virtual ~apStorageRepTmpl () {}

  const T* base () const
  { return reinterpret_cast<const T*> (apStorageRep::base());}
  T* base ()
  { return reinterpret_cast<T*> (apStorageRep::base());}
  // This cast is safe
};
```

Defining our **base()** method this way is an effective means of hiding the base class version of this method. As you would expect, **base()** is nothing but a safe cast of the **base()** pointer from **apStorageRep**. The base class uses **sizeof(T)** to specify the storage size of each pixel. This is passed to **apStorageRep** when it is constructed, as:

```
apStorageRepTmpl (unsigned int width, unsigned int height)
    : apStorageRep (width, height, sizeof (T)) {}
```

At this point, we are fairly satisfied with Prototype 3's design of the **apStorageRep** and **apStorageRepTmpl<>** objects. The base classes do all the real work, and the derived object handles the conversions. Unlike Prototype 2, where we let operator **new** (via **apAlloc<>**) create our pixels, in Prototype 3 we use an explicit cast to change our pointer from **unsigned char*** to **T***. In this context, casting is an efficient way to maximize code reuse. This cast is completely safe and contained within two methods.

Image Storage Handle Objects

Our handle class, **apImageStorage**, looks very similar to the **apImage<>** handle object we designed in Prototype 2. A portion of the object is shown here.

```
class apImageStorage
{
public:
  apImageStorage (); // A null image storage
  apImageStorage (apStorageRep* rep);
  virtual ~apImageStorage ();

  apImageStorage            (const apImageStorage& src);
  apImageStorage& operator= (const apImageStorage& src);
  // We need our own copy constructor and assignment operator.

  const apStorageRep* operator -> () const { return storage_;}
  apStorageRep*       operator -> ()       { return storage_;}

protected:
  apStorageRep* storage_;
};
```

The one major difference is that our handle object, **apImageStorage**, is not a template. That is because this object is a handle to an **apStorageRep** object and not an **apStorageRepTempl<>** object. The complete definition for the **apImageStorage** object can be found on the CD-ROM.

The **apImageStorage** object is not of much use to us because **operator->** allows us to access the **apStorageRep** rep object, and we really want to access the derived class, **apStorageRepTempl<>**. To accomplish this, we add an **apImageStorageTmpl<>** class that derives from **apImageStorage** as shown here.

```
template<class T>
class apImageStorageTmpl : public apImageStorage
{
public:
  apImageStorageTmpl () {}
  apImageStorageTmpl (unsigned int width, unsigned int height)
    : apImageStorage (new apStorageRepTmpl<T> (width, height))
  {}

  virtual ~apImageStorageTmpl () {}

  const apStorageRepTmpl<T>* operator -> () const
  { return static_cast<apStorageRepTmpl<T>*> (storage_);}
  apStorageRepTmpl<T>*       operator -> ()
  { return static_cast<apStorageRepTmpl<T>*> (storage_);}
};
```

Prototype 3 is a balanced design; we see that each base class has a corresponding derived template class. This symmetry is an indicator that we are getting closer to the final design.

Like our rep class, we also have to make a cast to get our **operator->** to work correctly. **static_cast<>** will safely cast the rep class (**apStorageRep**) pointer, kept by our base class, to an **apStorageRepTmpl<>** object. These casts may look complicated, but they

really aren't. Two casts are needed for an **apImageStorageTmpl<T>** object to access **apStorageRepTmpl<T>**. And because of these casts, we can put most of our functionality inside base classes that are reused by all templates. Figure 3.12 shows the relationship between these two objects.

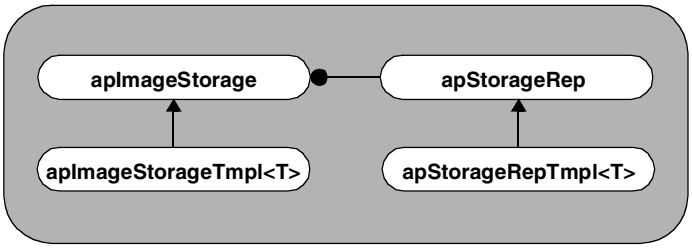


Figure 3.12: Image Object and Storage Separation Design

We are missing just one piece from this prototype. We need an object that actually performs the image processing operations on our image storage.

We chose to use **apImage<>** for this task with two template parameters:

```
template<class T, class E> class apImage
{
public:
  apImage ();
  apImage (unsigned int width, unsigned int height)
    : pixels_ (width, height) {}
  ~apImage () {}

  unsigned int width  () const { return pixels_->width();}
  unsigned int height () const { return pixels_->height();}

  const T* pixels () const { return pixels_->base();}
  T*       pixels ()       { return pixels_->base();}

  const T& getPixel (unsigned int x, unsigned int y) const;
  void     setPixel (unsigned int x, unsigned int y,
                     const T& pixel);

  // Image operations (only one for the prototype)
  apImage<T, E> thumbnail (unsigned int reduction) const;

  // Default copy ctor and assignment are ok
protected:
  apImage (apImageStorageTmpl<T>& storage);
  // Construct an image from the storage

  apImageStorageTmpl<T> pixels_;  // The actual image data
};
```

Other than its name, this object is not similar to **apImage<>** from Prototype 2. In that earlier prototype, **apImage<>** was nothing more than a handle class to the rep object. It was the rep object that did all the work. In Prototype 3, our **apImage<>** object is responsible for all of the image processing routines.

In this **apImage<>** object, all aspects of storing and maintaining the pixels are part of the storage object **pixels_**. **apImage<>** exposes only that part of the **apImageStorageTmpl<>** interface that it needs to for providing access to **width()**, **height()**, and **pixels()**. The definitions for **getPixel()** and **setPixel()** as they appear in Prototype 3 are as follows:

```
template<class T, class E>
const T& apImage<T,E>::getPixel (unsigned int x, unsigned int y)
   const
{ return (pixels_->base())[y*width() + x];}

template<class T, class E>
void apImage<T,E>::setPixel (unsigned int x, unsigned int y,
                             const T& pixel)
{ (pixels_->base())[y*width() + x] = pixel;}
```

This class structure is sufficiently complex that trying to define **operator T*** to avoid the explicit **base()** reference adds unwarranted complication. For example, we chose to define this operator in our **apAlloc<>** object because there was little confusion. But in this case, it would only spread the image functionality across three objects, giving rise to potential errors.

When we say:

```
pixels_->base()
```

it is clear we are calling the **base()** method of our rep class by means of the handle. Then, the following statement:

```
(pixels_->base())[y*width() + x]
```

becomes the lookup of a pixel, given its coordinates.

❖ THUMBNAIL() METHOD

The last method we need to look at is **thumbnail()**, and how it appears in Prototype 3 as:

```
template<class T, class E>
apImage<T,E> apImage<T,E>::thumbnail (unsigned int reduction) const
{
  apImage<T,E> output(width()/reduction, height()/reduction);

  for (unsigned int ty=0; ty<output.height(); ty++) {
    for (unsigned int tx=0; tx<output.width(); tx++) {
      E sum = 0;
      for (unsigned int y=0; y<reduction; y++) {
        for (unsigned int x=0; x<reduction; x++)
          sum += getPixel (tx*reduction+x, ty*reduction+y);
      }
      output.setPixel (tx, ty, sum / (reduction*reduction));
    }
  }

  return output;
}
```

If you removed the template references, this function is very similar to the **thumbnail()** method we designed in Prototype 1. This similarity to our very simple example means that we have a nice clean design.

Let's contrast Prototype 3's simpler version of **thumbnail()** with that in Prototype 2 to highlight the differences:

Prototype 2

```
apImageRep<T,E>* output =
  new apImageRep<T,E> (width()/reduction,
                       height()/reduction);
...
output->setPixel (tx, ty, sum / (reduction*reduction));
```

Prototype 3

```
apImage<T,E> output(width()/reduction,
                    height()/reduction);
...
output.setPixel (tx, ty, sum / (reduction*reduction));
```

The simple handle object in Prototype 2 meant our image processing routines had to access the computed images using pointers. We were able to access pixels in the current image using a normal method call, but access to a new image required access by means of the handle. In Prototype 3, access is consistent, regardless of which image we are trying to access.

Summary of Lessons Learned

These are the things that worked well:

- Dividing the image into storage and processing pieces enhances the design. It makes accessing the **apImage<>** object very direct using the **.** operator, even though the implementation of the object is slightly more complicated.
- Accessing image data from inside image processing routines is very clean.
- Writing custom versions of **apImage<>** with access to the same underlying image data is very simple using the current design. This element works extremely well, and we will keep it as an integral component of the final design.

Here is what did not work well:

- Using handles in our prototype has not shown a significant benefit to the design. **apAlloc<>** already allows the raw memory to be shared, avoiding the need to make needless copies of the data. Based on our prototypes, we have decided not to use handles in the final design.

3.3 Summary

In this chapter, we designed and implemented an object to efficiently manage memory allocation. We outlined the requirements of such an object in terms of a commercial-quality image framework. Then we designed and implemented the solution, using reference counting and memory alignment techniques to achieve those requirements. Because the solution involved heavy use of templates, we also provided a review of some of the syntactic issues with templates, including class conversion to templates, template specialization, function templates, and function template specialization.

With our memory allocation object complete, we began prototyping different aspects of the image framework design. Our first prototype explored two different types of images, 8-bit and 32-bit images, to determine if there were enough similarities to influence the design. We found that the classes were indeed very similar, and we felt that the design should be extended to include inheritance and/or templates to take advantage of the commonality.

In our second prototype, we extended our first prototype by using templates to handle the similarity among image objects. In addition, we introduced the handle class idiom, so that image objects of different types could share the same underlying representation object. Then we took the prototype one step further by exploring more complex image types, such as RGB images. We found that the use of templates resulted in an efficient design that leveraged the similarities; however, the handle idiom did not provide any obvious advantages.

In our final prototype, we explored separating the image storage component from the image object, because of the amount of data our image classes contained. We felt this strategy might allow the eventual reuse of the image storage object by various image objects. Once again, we tried to apply the handle idiom in our solution. We found that dividing the image into storage and processing pieces enhanced the design, but the handle idiom did not provide any obvious benefits.

In Chapter 4, we consider other issues for the final design of our image framework, including: coding guidelines and practices, object creation with the goal of reusability, and the integration of debugging support into the framework's design.