Old Possum’s Book of Practical Objects

by Shane D. Looker

Object-oriented programming has been promoted as a way to increase code reusability, code reliability, and get your socks whiter. The potential is there, but nobody seems to be sharing or selling classes that are truly reusable. In this paper, several small but powerful C++ base objects are described that can be taken and used as is, or, by overriding a few methods extended to fit a specific need in a product. Design techniques are development benefits are discussed. Classes shown are a FIFO queue, a simple binary tree class, a Boyer-Moore search object, and a keyword class.

Object-oriented programming has been promoted as a way to increase code reusability, code reliability, and get your socks whiter. The potential is there, but nobody seems to be sharing or selling classes that are truly reusable. Brad Cox expounded the concept of the Software IC back in the mid-1980’s. The Software IC is a standard object that can be taken off the shelf and plugged into new code with no changes. It can be extended to meet a specific need, but the core is unchanged. It is a good idea, but I haven’t seen it actually applied, at least not in the Macintosh world. The closest that I have seen are the class libraries such as MacApp or TCL which have only a few reusable classes which are not totally dependent on the entire class library.

One of the goals of this paper is to actually publish classes that are reusable in practical situations. Hopefully this will save you development time and give you more time to work on pieces of code that are unique to your program.

Making It Reusable

What is a reusable object, or more correctly a reusable class? It is a core class that can be used in multiple projects for different purposes without needing modification. The class is structured in such a way that internal data is not specific to a given purpose, and can be changed by overriding the internal data creation methods. The converse of the “what is” question is often easier to answer and can be used to identify non-reusable classes. Let’s examine properties of a class that prevent it from being reusable.

First: Is the class tied to a particular project? This often can be seen by inspecting the instance variables and methods of a class. Do they have names that reflect a particular project, or structures in a project? This is a warning flag that reusability was not stressed during the coding phase of the class, and perhaps not during the design phase.

Second: Is the class inflexible? Is it written in such a way that behaviors can’t be overridden to change or add functionality? For instance, are assumptions made about major internal data structures, so they can't be easily changed?

Third: Are data members and methods declared as private? Doing so prevents access and / or overrides from child classes, causing either new accessor methods to be written in the parent class, or changing the items to protected instead. (Protected is my preferred way to handle this, since it maintains the data hiding from the instantiating code, but allows legitimate descendants access to important internal structures.)

```
class BufferedData
{
   Ptr   fDataBuffer;
   long  fItemCount;

public:
   …  // Constructors, etc.

   AddNewItem(short anItem);
   short GetAnItem(void);

private:
   Ptr CreateTheBuffer(
            long defaultItemCount);
};
```

Snippet 1 – Non-Reusable Object

Snippet 1 illustrates both the second and third problems. By making a few simple changes (as seen in Snippet 2), the class can be converted into a reusable class.

Snippet 1 makes a number of assumptions that prevent the code from being reused. (Yes, this is blatant. It's an example.) The data buffer can only hold 2 byte values. The buffer is always stored in a Pointer. The method `CreateTheBuffer()` can't be overridden by any child class (both private and non-virtual). All in all, it works great for storing 2 byte values in the heap somewhere, but it can't be extended to do anything beyond that. What if, for instance, the values being stored were a list of used sectors on a disk? If the file system changes (say to support more than 64K sectors per disk) the values to be stored might need to be changed to 4 byte integers. The entire class would need to be modified, possibly leading to hours of debugging when one of the internal assumptions is missed during the code changes.

```
class NewBufferedData
{
   void* fDataBuffer;
   long  fItemCount;
   long  fItemSize;

public:
   …  // Constructors, etc.

   void  AddNewItem(void* anItemPtr);
   void  GetAnItem(void* outItemPtr);
   long  GetBufferItemSize(void);

protected:
   virtual  void* CreateTheBuffer(
            long defaultItemCount);

   virtual  void* GetBufferPoint(
               long whichItem);
   long  GetBufferItemSize(void);
};
```

Snippet 2 – Reusable Object

Now consider Snippet 2. If this object were used to hold 2-byte integers, a new sub-class could be created by overriding `CreateTheBuffer()`, `GetBufferPoint()`, and the constructor (to set fItemSize). The code for `AddNewItem()` and `GetAnItem()` would have been written to use `GetBufferPoint()` when an index was needed, so the only further change required (regarding this object) would be to change the object instantiation to use the new child class instead of `NewBufferedData`.

It is assumed in this object that all items being inserted into the buffer are of the same size. Without this assumption, the underlying code would be much more complex.

There are two primary arguments against the code in Snippet 2: It avoids type checking which is very useful when writing code and you also end up having to coerce data from one format to another. Both of these concerns can be addressed when writing a descendant class. The type checking can be moved into the new child class by overloading the methods `AddNewItem()` and `GetAnItem()` to accept the data of the correct type for that class. This also allows the actual data coercion to be hidden inside the child class.

The benefits of the code for `NewBufferedData` far outweigh the two arguments against using it, in my opinion. With minimal effort, I can now create a buffer for any data type, store and retrieve that data at will, and even change the storage from a pointer to a handle if I want, all in a simple override.

Arguments can be made both for and against flexibility. On one hand, flexibility is usually useful at some point of an objects lifespan. On the other hand, flexibility adds complexity which usually results in slower execution and more implementation and debugging effort. A balance will often need to be struck when designing and implementing a reusable class.

Digging In

Since the requirements for a good reusable base class are fairly simple, we should look at several classes that are useful to keep around. While not every project needs any one of these classes, they do come in handy from time to time.

A FIFO Queue

The first item I want to discuss is a FIFO (First In, First Out) queue. This is a basic element in computer science which is pretty useful. Items are added to the end of the queue, and are only accessed / removed from the front of the queue (hence first in, first out).

There are a few restrictions on this object which allow it to be kept very simple. Since this isn’t a general purpose queue, there are no accessors for arbitrary queue elements. More importantly, every element of the queue must be the same size. Snippet 3 shows the class definition for this queue.[1]

While a simple queue is actually conceptually easy to write, there are added considerations to make it flexible and reusable. There are also a few items about the actual code for this object which should be discussed.

I have actually written two separate classes for FIFO queuing. The first (shown here) depends on using a handle to store the queue. The second (not shown) added the flexibility to override the standard queue behavior of using a handle for the data storage. This class is the simpler and for most purposes the more practical to use, since it doesn’t have the added overhead needed for flexible storage. It simply uses a handle that can be extended on demand. It also doesn’t shrink the handle unless the queue is forced empty by a call to `EmptyQueue()`.

Notice that even though you will probably never need to override this class, most of the functionality is defined in virtual methods. Also protected has been used for the one internal function that doesn’t need to be referenced directly by the user of this class.

The data handling internally uses the toolbox function `::BlockMoveData` to shift data around, which is extremely inefficient for small data blocks. This is partially offset by the ability to tell the object that the data is the size of a long (for Handles, Pointers, or just plain integer values) so they an be directly manipulated.

While this object is very simple, it can be used in multiple ways. In drag and drop applications I often use this class to hold the list of files to be processed. If you had a multiply threaded application, different classes of threads could be given different priorities, which each priority level having its own task queue. The actual data supplied to the object is flexible,

[1] The headers shown here will usually be condensed from the full header. Full source code for the objects described in this paper will be available on the MacHack ’96 CD-ROM.

but the queue behavior is integral to the class.

A Binary Tree Object

Now let's take a look at the second object I want to present: a binary tree. Binary trees are simple and have been used for years. So why do people reimplement them so often when they go to write code? Because other versions of their binary trees are usually tied to specific data structures. The non-object version of a binary tree usually uses a structure to hold the data for each node of the tree. It might look like:

```
struct BinaryNode {
   long     key;
   Ptr      someData;
   BinaryNode  *leftNode, *rightNode;
} BinaryNode;
```

The key and data for the node are tied into the full structure and are not easily separated.

Binary trees are also often implemented procedurally using recursive functions. A basic binary tree is just a collection of nodes (objects) that are linked to peers of the same type. Each node can have up to two children and a single parent. The node is defined as containing some sort of key value and data associated with that key. Normally the key value is used to determine the insertion location. The key value must be unique with the tree.

We can now take a quick look at the functionality we might want from a binary tree:

- Flexible key and data types
- Insert and delete nodes
- Find a node based on a key
- Tree traversal (in order, reverse order)
- Node count
- Maximum tree depth

There are also a few advanced functions we might want:

- Balancing of the tree
- Finding a key based on data

Most of the implementation of a binary tree is straightforward and easy to implement. By doing the implementation using a reusable object, we gain greater flexibility. The important point is to note that the key and data for any given tree are the only items that are not determined when the object is written. Simple default behaviors can be put in the base class (instead of making this a pure virtual class), and to customize the code for a project, only two methods must be overridden. Those two methods are the comparison methods `CompareToNodeKey()` and `CompareToNodeData()`. Once those are overridden, any new type of data can be tested for insertion or searching.

In this class I've decided to take the easy way out and make the key (fNodeKey) and data (fNodeData) of type void*, letting me put arbitrary data or pointers or even up to 4 bytes of raw data (another abuse of C++) into each.

One last detail of the design is whether any arbitrary node within the tree represents the entire tree, or whether a node can be considered the independent root of it's own subtree. Since each key is required to be unique (in the basic tree object), I have decided that each node must be able to represent the entire tree. This means I can tell any node to insert the new data into the tree, and it will insert in the correct spot for the entire tree. This is very useful, since it means I can use any node available as my entry to the tree, not just the root of the tree.

In the header file included with this paper you might note that most methods are declared as virtual, even though there will probably never been a need to override them. While this does generate slightly slower code (an additional branch instruction to the method code), it may make it slightly easier to adapt the class for other purposes.

Boyer-Moore Search Object

Boyer-Moore is a nice, fast searching algorithm for finding one string within another. It depends on the ability to back-up in the search

```
class CQueue
{
   Handle   fQueueHead;
   long     fQueueMaxElements;
   long     fQueueElements;
   long     fQueueElementSize;
   Boolean  fOnlyLongs; // This flag means we are cheating and taking the
                        // void* from AddElement as a long, not a pointer to
                        // data. This can make us MUCH faster for Ptrs, etc.
   Boolean  fAutoExpand;      // Can we grow the queue on need?

public:
         CQueue(Boolean autoExpand = true, Boolean onlyLongs = false);
         CQueue(long elementSize, long maxElements = kDefaultQueueSize,
               Boolean autoExpand = true, Boolean onlyLongs = false);
   virtual  ~CQueue(void);

   // SetMaxQueueSize could return an error (-108, or QueueTooBig)
   OSErr SetMaxQueueSize(long maxElements);
   long  GetMaxQueueSize(void);

   long  GetQueueElementSize(void);

   // these functions return FALSE if they failed in some way.
   // GetFrontElement could fail if the queue is empty.
   virtual  Boolean  AddElement(void *theData);
   virtual  Boolean  GetFrontElement(void* theData);
   virtual  void     EmptyQueue(void);

   long  GetQueueLength(void);

protected:
   virtual  void  AddOneElementToQueue(void *theData);
};
```

Snippet 3 – CQueue Class Definition

string and restart a search, which means it is best suited for data contained completely in memory. Boyer-Moore works by comparing a search string against a target string, working from left to right for the overall search, but comparing bytes in a right to left fashion working from the last character of the search string. If the characters being compared don’t match, the target string pointer is advanced by a distance defined in an internal skip table (based on the search string) and comparing begins again.[2]

With additional work, it would be possible to read data from a file, backing up if necessary by resetting the file pointer and reading again. As you will see from this object, with proper overriding, this can be done, although the results may be less than satisfactory.

The primary data used in the Boyer-Moore search are a search string, and the data to search in. Both of these need to be supplied to the object. Internally, it builds the required search table, and can start searching from any arbitrary offset into the data.

From the implementation perspective, this object shows how to mask the real data type

[2] R. Sedgewick, Algorithms in C, Addison-Wesley, Reading, MA, 1990

being supplied by a child class. It uses two overrideable methods (`PrepareSearchData()` and `CleanupSearchData()`) to setup the addressability of the text to be searched. The data search area is limited to a long (2 Gigabytes) which should not be a major limitation for the foreseeable future.

Conversely, this object is built in such a way that it can't search for a target string more than 256 characters long. To implement that change would require updating the internal skip table to a larger data type (currently unsigned char), and redefining the way the search string is stored (currently an array of unsigned char.) Changing this would require either dynamic storage for the search string, or a reduction in the flexibility of data types which can be fed into this search object, due to the requirement that the caller hold the data for the object. (Not to mention the fact that the purpose of an object is to hide implementation details from the caller.)

I have also added one other feature to this search object, the ability to be case insensitive. By setting this flag, all comparisons on data are done after calling `tolower` (from the standard ANSI library) on the data. The side effect of this is that only ANSI text can be compared. A more correct way to handle this is to move the comparison of the two characters into a separate method which returns a Boolean result.

The reason I have chosen not to do this is because of the extremely high overhead such a call would create. I also don't want to use an inline method, since that would defeat the entire purpose of creating a separate method.

Designing an Object–Keywords

Although many of you reading this paper already are familiar with designing an object and coding it, I thought it would be a good exercise to follow the thought/design process I used for this final reusable class.

I have had occasion in the past to want an easy way to attach keywords to objects, but I've never gotten around to writing a consistent way to do it. Last year I actually had to write a keyword object for an image browser. They class worked, but because of the structure of the project and time deadlines, the classes I wrote were not as flexible as they could have been.

The original solution I wrote dealt with keywords for a collection of images. Images were collected in albums, and albums could have child albums. A fair amount of work was done to map keywords from different albums to the same keyword in a master list. There was also some code that is best left undescribed due to some of the internal document structures.

Now I have taken the lessons I learned and created a fairly easy to use class that allows arbitrary keywords to be maintained in a single object (multiple instantiations can hold multiple lists, of course) with user defined references attached to each keyword. I've also supplied methods for searching they keyword list by keyword or by reference information. These methods were not integral to my original solution and caused extra work during the building of the search engine.

The initial design process I tend to use is an internal dialog: I answer internal questions about the functionality of a class and how it is implemented as the questions occur to me. The questions usually follow from the previous answer, so there is a natural progression and refinement to the class which happens before I actually start to even write the header. What follows is a compressed version of my internal dialog.

One final point before my discussion: By the time I write a reusable class, I have already either thought about or written a different version of the class. As a result, I am not starting with an entirely clean slate. This can give a good boost to the design process, since pitfalls of previous code are known and can hopefully be avoided.

The Dialog

What should the object do? It should hold keywords that can be attached to other objects or items. I should also be able to search for anything that uses a given keyword.

How can you search for uses of a keyword? Each keyword should have a reference to anything that uses that keyword.

What is that reference? Undetermined. It could be a pointer, or a handle, or some user-defined token.

So how do you define a token? It should be at least a long word of storage. In the past I've needed more than a single long though, so let's assume that at least two longs of data are going to be used.

How long can a keyword be? Virtually unlimited would be nice. It should be able to at least be as long as a Str255.

How do I identify a keyword? Each keyword should have a unique token that can be used to find the text of the keyword and any references it holds. I also need to be able to find a keyword by name.

How do I want to store the keywords? I could store each keyword in its own storage block, but that would take a lot of Handles or Pointers, and that would slow down the Memory Manager if I had a lot of keywords. If I used something like `malloc()`, I would still need to keep track of the address for each one. I could cluster blocks into large blocks and then link those larger blocks together for easy indexing.

How do I prevent wasting a lot of storage if the keyword is short? Don't assume a fixed length for keyword storage. Try using basic blocks that can be extended. Then I could use multiple storage blocks for longer keywords.[3]

[3] This is where implementing something previously helps. I had already successfully used this technique in a previous implementation.

How do I generate an ID for each keyword? Easy, the ID is the addressing mode to get to the start of the keyword.

If I delete a keyword, should I compress the structures to remove the hole? No. If you do that, all the IDs that have been assigned will get screwed up. That would be bad if some of the references are stored in a file somewhere.

If I delete a keyword what happens to the ID? Well, the block holding the keyword will be freed. It could be reassigned, so make sure to use the references for the keyword to remove all traces of it first.

How are references stored? I could attach the references to each keyword, but then I am back to the major problem of having a lot of handles, or whatever, to keep track of and possible slogging through thousands of multiple handle de-references to search through the references.

Starting the Code

At this point I have the basic idea for the class laid out on in front of me with some major implementation details left. As I code this class, further things may cause me to reexamine some of my decisions, but for now I think they are sound.

At this point, it looks like there may be a fair amount of work to making this class work properly, especially the references. Why not just write a class for what I need and not bother? The reason it is useful to write this flexibly, is that the next time I need to do something similar, I don't want to have to start from scratch, or take the time to modify and debug a different version of the code. Also, with a common piece of source code, any bugs only need to be fixed once. Code changes don't have to be propagated to five or ten different projects with the possibility that a new bug is introduced in any of those projects. I think you would agree that this could make life much easier.

Implementation Details

Now we need to turn our attention to the implementation details. How do long keywords get stored efficiently and as cleanly as possible? How are references stored? And how do I search for items using this object?

There are two major portions to the object, the actual data storage, and the data searching. Attempting to pre-allocate enough storage for all the keywords can lead to grabbing large amounts of memory at once, which may not be needed. Conversely, storing each item individually leads to excessive use of pointers or handles. It is possible to allocate each keyword from a block using `malloc()` or `new()`, but again, that leads to a large number of pointers that must be maintained. Instead I elected to create "pages" of keywords. Each page consists of header information and data in much the same way that virtual memory uses pages. In this implementation I've chosen to allocate keyword pages as Handles so they can be moved in memory when they are not directly in use.

Keyword storage is a trade-off between allocating each storage chunks dynamically, and allocating fixed storage sizes. The fixed sizes allow much faster indexing through the keywords, but we don't want to waste too much space with unused data storage. As a compromise, I have used a storage blocked essentially defined as:

```
struct {
   short blockCount;
   short length;
   char keywordData[28];
} KeywordHolder;
```

If a keyword is more than 28 characters (defined as an enum in the real header file) then they keyword will take the next KeywordHolder in the list and use the entire structure (all 32 bytes) to contain the remainder of the keyword. Of course, of more than two blocks are required, they are used.

They keywords themselves can be up to 32763 characters, although keywords that long are considered bad form. (In the default implementation, there are 100 keywords, with a keyword block size of 32 bytes, per page, and the header on a keyword block is 4 bytes.)

Keywords can be referenced by ID, with the first keyword having an ID of 1. Keywords are not necessarily assigned in sequential order, since deletion of a keyword can leave a reusable hole inside the data page that may be reused (depending on the flag settings). While this may seem inefficient with memory, or more complicated to maintain, it is very important since it leave keywords static over a save and restore cycle. This means that an object can maintain a simple keyword reference without fear of the keyword ID becoming invalid.

Keywords can also be searched for entire or partial matches. The CKeywords class uses an instance of the BoyerMoore object to search an entire block quickly and efficiently for the named keyword.

Keyword References

Each keyword can also have multiple references associated with it. The associations are one or more longs which are stored internally and can be used as a link from the keyword to all objects which use that keyword. When the CKeywords object is instantiated, the number of longs used to make a single association is one of the optional parameters (the default value is 1 long). When a keyword reference is added a pointer to an array of longs is passed in.

The references are also stored in pages for searching and indexing. The pages allow minimal wasted storage while giving some flexibility to the memory requirements of the references.

Searching

Searching for the keywords is done through use of the BoyerMoore object described above. It allows for partial matches on keywords, or on entire keywords. References can also be searched to find all keywords attached to a given object.

Source Code

The full headers for these object is fairly extensive and will not fit in the space limitations for this paper. The headers and the source code for the objects described in this paper can be found on the MacHack ’96 CD-ROM.

Summary

While the objects shown in this paper are all fairly straightforward, they have the useful property of being reusable. They can be taken off the shelf and used in a program with little or no changes required. The advantages of this are obvious, but during development, most code written is still designed with a one-shot approach: make it work well enough for today and don’t worry about tomorrow.

Unfortunately, tomorrow we often have to use a slight variation of the code from yesterday, but we forget how the old code was written, or it is just different enough that the old code isn’t usable. If we take a little extra time to plan ahead today, we can save ourselves time in the future. As the saying goes “There’s never enough time to do a job right, but there is always time to do it over.”

It is past time for us to use the power of the tools at our disposal to do the job right the first time. Using simple techniques when we design objects, we can make tools that are more than just adequate for today, but actually useful next week as well. By asking what we want to accomplish, whether what we are writing actually accomplishes our goal, and whether it can be extended for other uses in the future we can save time and effort.

If we don’t, we will still be reinventing the wheel every time we start a new project.