

# Programming with Haiku

## Lesson 1

Written by DarkWorm



For those who are reading this after finishing *Learning to Program with Haiku*, we will start with a few concepts to round out our basic knowledge of C++. Today's main topics will be templates and the Standard Template Library. There is a lot of information needed to understand both, so go slowly and carefully.

## Templates

Templates are a way of generalizing a class or function to deal with any type of data. They are most often used in creating container classes such as stacks, lists, and others. Taking advantage of this flexibility requires a few small changes to the declaration of the class or function which uses them. A template-based array class could look like this:

```
template <class T>
class ObjectArray
{
public:
    ObjectArray(const uint32 &size);
    ObjectArray(const ObjectArray &from);
    ~ObjectArray(void);

    ObjectArray & operator=(const ObjectArray &from);
    T & operator[](int index);

    uint32 GetSize(void);

private:
    T *fData;
    uint32 fSize;
};
```

In this instance, the main difference between this and a regular class definition is the `template <class T>` section before the definition. This tells the compiler that any time we see `T`, we are talking about the type used by that class. Note that we can use pretty much any name we like for the type name, such as `T`, `Key`, `Type`, or whatever, but we have to use the name we declare in the template section to refer to the type name. The `ObjectArray` class has two properties: the number of elements, kept in `fSize`, and a heap-allocated buffer used to hold the individual elements pointed to by `fData`. Without worrying about the implementation, an `ObjectArray` instance could be used like this:

```
#include <stdio.h>
#include "ObjectArray.h"

int
main(void)
{
    ObjectArray<int32> intArray(10);
    for (uint32 i = 0; i < intArray.GetSize(); i++)
        intArray[i] = intArray.GetSize() - i;

    for (uint32 i = 0; i < intArray.GetSize(); i++)
        printf("intArray[%ld] is %lu\n", i, intArray[i]);
}
```

The key to using class templates is to specify the type in angle brackets. In the above example, we have created an `ObjectArray` instance which uses 32-bit integers as its type. It

does all of its work centered around `uint32` objects and does not concern itself with any other type. Once created, it cannot change its type – an `ObjectArray<uint32>` cannot be changed into an `ObjectArray<float>` or even an `ObjectArray<uint16>`, for example. Also, two `ObjectArray` objects with different types are treated as completely different types by the compiler, i.e. `ObjectArray<float>` and `ObjectArray<uint64>` are not the same type even though they both use the `ObjectArray` class as a template.

Defining functions in a template-based class isn't difficult. Many references to the class name need to include the type name. Here is the code used to define all of the functions in the above class definition for the `ObjectArray` class:

```
template <class T>
ObjectArray<T>::ObjectArray(const uint32 &size)
    : fData(NULL),
      fSize(size)
{
    if (fSize > 0)
        fData = new T[fSize];
}

template <class T>
ObjectArray<T>::ObjectArray(const ObjectArray &from)
    : fData(NULL),
      fSize(from.GetSize())
{
    if (fSize > 0)
    {
        fData = new T[fSize];
        for (uint32 i = 0; i < fSize; i++)
            fData[i] = from[i];
    }
}

template <class T>
ObjectArray<T>::~ObjectArray(void)
{
    delete [] fData;
}

template <class T>
ObjectArray<T> &
ObjectArray<T>::operator=(const ObjectArray &from)
{
    if (this == &from)
        return *this;

    delete [] fData;

    fSize = from.GetSize();
    fData = new T[fSize];
    for (uint32 i = 0; i < fSize; i++)
        fData[i] = from[i];
}
```

```

template <class T>
T &
ObjectArray<T>::operator[](int index)
{
    return fData[index];
}

```

```

template <class T>
uint32
ObjectArray<T>::GetSize(void)
{
    return fSize;
}

```

Aside from being careful about adding <T> in a few places and the extra typing needed at the top of each function definition, there is one other caveat when using class templates: templated-based function definitions need to be placed in the header instead of the main source file. This is because these are templates for function definitions, not the definitions themselves. The actual definitions are generated at compile time. If these functions were placed in the main source file, the linker would spit back a host of undefined reference errors which, needless to say, is more than a little confusing.

Sometimes you don't need the entire class to use a template. Perhaps all that is needed is a function or two which make use of templates. In this instance, all that is needed is to place the template declaration section in front of the return type for the function declaration like this:

```

class MyClass
{
    MyClass(void);
    ~MyClass(void);
    void SomeRegularFunction(int value);

    template<class T> void SomeTemplateFunction(T item);
};

```

In this case, SomeTemplateFunction() is the only part of MyClass which utilizes a template. Just like the functions for our ObjectArray class, it will need to be defined in the header and not in the main source file with the rest of the functions for MyClass. In these instances, only the function templates need to be in the header – the "regular" functions should go in a separate source file as they normally are.

## ***Using Templates: The Standard Template Library***

The best use of templates is for data containers. Luckily for us, a group of people took the time and effort to create an entire library of template-based containers called the Standard Template Library, or STL for short. Most of the Haiku API is so well-designed that we do not need to use the STL very often, but it does have a few containers that come in quite handy. Because the containers in the STL are so well-designed, most of the time our use of templates will be limited to using these containers instead of creating classes that use templates.

Any Haiku projects which use the STL need to add an extra library. GCC2 builds of Haiku use the library libstdc++.r4.so. Haiku GCC4 builds link to libstdc++.so for STL usage.

The containers provided by the STL fall into two categories: sequential and associative. The sequential containers are designed for working with items in a list which have a definitive order, such as a list. The BList class provided by the Haiku API is an example of a sequential container. Such containers are optimized for operating on the entire list one item at a time. Associative containers are intended for data which requires seemingly random lookup or by values that are not integers. Probably the most common case for needing an associative container is using a string to look up data. Choosing the specific container to use depends on the kind of work you intend to do with them.

## *Vector*

Header: <vector>

The vector class is very similar to an array except that memory allocation is handled automatically. Vectors are good at quickly accessing elements by index, iterating over elements in any order, and adding and removing elements from their end. Adding elements can be slow, but only when it runs out of internal storage and has to allocate more.

## *Deque*

Header: <deque>

A deque, usually pronounced "deck," is short for **double-ended queue**. Deques are very similar to vectors except that they are efficient at adding elements at both the beginning and end and their elements are not guaranteed to occupy a contiguous chunk of memory. This has both benefits and drawbacks. Using pointers to access elements of a deque is not safe, unlike elements of a vector, but they are a much better choice for storing large numbers of elements.

## *List*

Header: <list>

Lists are best described as a collection of dynamically-allocated items. They are best used for work which involves inserting, removing, and moving items around. One major drawback to using lists is that there is no way to quickly access an arbitrary element – accessing the tenth element in the list requires iterating from the beginning (or other reference point) to that element.

## **Namespaces**

Using STL containers also requires some extra typing because all of them are encapsulated into their own namespace. Namespaces are a way of creating groups of classes, functions, and data types. Often they are used to prevent conflicts between classes in different libraries that have the same name. A namespace is declared like this:

```
namespace myNamespace
{
    int32 foo, bar;
}
```

Accessing elements inside a namespace requires specifying the namespace. For example, all STL containers are inside the `std` namespace. Declaring a vector of `int32` objects would look something like this:

```
// Note that there is no '.h' for this header and others in the STL. It's
// just <vector>.
#include <vector>

std::vector<int32> intVector;
```

The double colon is the scope operator. Specifying items within the same namespace do not require it. Likewise, specifying an item in the global namespace from within another requires a double colon.

```
#include <stdio.h>

bool gSomeFlag = true;

namespace myNamespace
{
    int intValue = 1;

    void
    SomeFunction(void)
    {
        // This specifies the gSomeFlag which is in the default (global)
        // namespace
        if (::gSomeFlag)
            printf("myValue is %d\n", intValue);
    }
} // end myNamespace
```

Using a lot of items in another namespace can make for a lot of typing, so if you are using a particular namespace quite a lot, you can employ the `using` keyword to eliminate the extra typing.

```
#include <deque>
#include <stdio.h>

// This eliminates the need to add the std:: before each reference
// to deque containers.
using std::deque;

int
main(void)
{
    // Declare our deque to accept integers. Without the using statement
    // above, this would read std::deque<int> myDeque. Unless you're
    // using a lot of these, the using statement isn't really needed.
    deque<int> myDeque;

    // Add one element to the end of the list which has a value of 5
    myDeque.push_back(5);
}
```

```

    // Print the number of elements in the deque. In this case, we're
    // definitely not playing with a full deque.
    printf("This deque has %d elements\n", myDeque.size());

    return 0;
}

```

The `using` keyword offers fine-grained control over what requires us to type the namespace. In the above example, we removed the need to specify the namespace whenever we use the deque container. If we were also using the vector container, we would still have to type `std::vector<myType>` for vector type references. If there are no possibilities for name conflicts and we use many different containers from the STL, then we can make a blanket using declaration which covers everything in the `std` namespace:

```
using namespace std;
```

Be careful using the `using` keyword this way. If your code must deal with more than one namespace, it is highly advisable that you use it on a class-by-class basis or, better yet, not use it at all. However, if you are working with a Haiku program which just makes calls to the API and a few things from the STL, then using declarations which speed up your work a bit are just fine.

## **STL Iterators**

The designers of the containers in the STL were wise to attempt to keep the API for all of the containers as similar as possible. One way that they did this was to create iterators. Because not all of the containers lend themselves to using integers to access each element like an array does, iterators are used instead.

Each iterator is more or less a pointer to the element type used by the container. The `++` and `--` operators have been overloaded to go to the next or previous element in the container. Using a `vector<int>` in a for loop would look something like this:

```

#include <stdio.h>
#include <vector>

// This using statement only works for the vector class. It also makes
// the loop code a little more readable.
using std::vector;

int
main(void)
{
    vector<int> myVector;

    // Add a few values to our vector
    myVector.push_back(5);
    myVector.push_back(10);
    myVector.push_back(15);

    // The begin() method will always point to the first item in the
    // container. end() will always point to the last one. This format
    // works for *all* STL containers.
    for (vector<int>::iterator i = myVector.begin();
         i != myVector.end(); i++)

```

```

    {
        printf("myVector: %d\n", *i);
    }
}

```

We've seen a number of methods used without any explanation. Sadly, sometimes in the real world the only available documentation for code is the code itself, but it doesn't have to be that way here. These are the methods we've seen so far with vectors, deques, and lists and a few other useful ones. By no means is this an exhaustive list, however.

Method	Description
iterator begin();	Returns an iterator pointing to the first element in the container.
iterator end();	Returns an iterator pointing to the last element in the container.
size_type size();	Returns the number of elements in the container. size_type is an unsigned integer type.
void push_back(const T &val);	Adds an item to the end of the container's list which has the value val. Note that this effectively invalidates any existing iterators.
void pop_back();	Deletes the last item in the container.
void push_front(const T &val);	Adds an item to the beginning of the container's list which has the value val. Note that this effectively invalidates any existing iterators. This method is unavailable to the vector class.
void pop_front();	Deletes the first item in the container. This method is unavailable to the vector class.
void clear();	Deletes all items in the container.

## ***Tying It All Together: An Example Usage***

After seeing these containers, there aren't very many instances where using a BList seems all that advantageous. For example, one instance where using one of these containers would be more efficient than a standard issue BList is making a list of entry\_ref objects from a directory. Below is an example of how all of the information that we've seen about STL containers can be put to use in a way that makes sense.

```

#include <deque>
#include <Directory.h>
#include <Entry.h>
#include <FindDirectory.h>
#include <Path.h>
#include <stdio.h>

using std::deque;

int
main(void)
{

```

```

// Look up the home folder -- this is much preferable to a
// hard-coded way of accessing a system path.
BPath path;
find_directory(B_USER_DIRECTORY, &path);
BDirectory dir(path.Path());

deque<entry_ref> refDeque;

entry_ref ref;
while (dir.GetNextRef(&ref) == B_OK)
    refDeque.push_back(ref);

printf("Contents of the home folder: %s\n", path.Path());
for (deque<entry_ref>::iterator i = refDeque.begin();
     i != refDeque.end(); i++)
{
    // Note that because an iterator can be treated like
    // a pointer, we can access each entry_ref's name
    // using the iterator.
    printf("\t%s\n", i->name);
}
}

```

### ***Going Further***

- How could this example be changed to use lists instead of dequeues?

# Programming with Haiku

## Lesson 2

Written by DarkWorm



In our first lesson, we learned about how to generalize type handling using templates and some of the incredibly flexible data containers in Standard Template Library. The list, deque, and vector containers are good at storing data for sequential access, but there is more to the STL than merely these three. We will study the other major containers and the differences and similarities among them. There will also be some coverage of a portable string class which makes string manipulation much easier than with the standard C functions. There is a lot of information in this lesson. Go slowly and don't be afraid to re-read it once or twice, if necessary.

## C++ Strings

Before we tackle these other containers, it is necessary to know about the Standard C++ library, which uses concepts from the STL but provides other useful development tools. One of these is a general purpose string class.

Working with strings using the standard C functions such as `strcpy()` and `strstr()` is a lot of hassle. The wonderful `BString` class in the Support Kit eliminates this hassle, but so does the C++ `string` class. Most of the time you will find the `BString` class preferable because it is generally faster, can do far more, and is integrated into the Haiku API, but there are a few methods which the C++ strings have for which `BString` has no equivalent.

```
string substr(size_t pos = 0, size_t n = npos);
```

`substr()` returns a substring starting with the character at `pos` and ending at `n`. `npos` is a static value which equals the greatest size the string could be. Needless to say, it's silly to call this and use both of the default values because this just returns the entire string.

```
size_t find_first_not_of (const string &str, size_t pos = 0) const;
size_t find_first_not_of (const char *s, size_t pos, size_t n) const;
size_t find_first_not_of (const char *s, size_t pos = 0) const;
size_t find_first_not_of (char c, size_t pos = 0) const;
size_t find_last_not_of (const string& str, size_t pos = npos) const;
size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
size_t find_last_not_of (const char* s, size_t pos = npos) const;
size_t find_last_not_of (char c, size_t pos = npos) const;
```

These two functions search the string object for the first (or last) occurrence of a character that is not one of the ones specified in the first parameter of the call. Here is an example of how it can be used.

```
std::string myString = "/boot/home/config/settings";
size_t pos = myString.find_first_not_of('/');
```

In this example, the `char` version of `find_first_not_of()` is used to find the first character which is not a slash. `pos` in this example has a value of 1. Used in series, this could be a way to break up a file path into a series of folder names without resorting to using `strtok()`.

C++ strings are also in the `std` namespace and may be referred to as `std::string` to differentiate them from regular C-style strings. They are used with the header `<string>`. We won't be using C++ strings very much, but it's best to know about their existence because they are used fairly often on other platforms.

## **Associative Containers**

The associative containers provided to us by the Standard Template Library are necessary when it is slow or impossible to use integers to look up data in a container. While it is possible to manually search arrays, vectors, or other sequential containers using a loop, it is potentially slow to do so. For an occasional lookup, this is just fine, but if you have to repeatedly look up information this way, this can impose a serious performance hit in your code.

### *map*

Header: `<map>`

The `map` container centers around what is known as a key-value pair, i.e. a lookup value and the data that goes with it. Declaring a `map` object involves the types for both the key and value, like this:

```
map<BString, int32> myMap;
```

This declaration creates a `map` which uses strings to look up integers. It is common to see `maps` being used to look up data using strings as keys. By using `std::string` or `BString` as the key type, it is possible to use a regular C-style string to get the value.

```
printf("The value for %s is %d\n", "Some value", myMap["Some value"]);
```

The only caveat to using a `map` is that the key values are expected to be unique. The items in the `map` are actually another STL container: `pair`. The `pair` container merely associates two types together. Accessing the two types paired together is done via the properties `first` and `second`.

### *set*

The `set` container is similar to a `map` except that the values are also the keys and they are sorted. Like `map`, all items in a `set` are expected to be unique. It is not often used because there are other containers which are more flexible. The main purpose for using a `set` is guaranteed fast insertions and fast lookups. The implementation of the `set` container is normally pretty complicated so it can provide these guarantees.

### *multimap, multiset*

These are versions of `set` and `map` which do not require the keys to be unique. Calling the `find()` method still returns only one item with the key given to it, but these containers have an additional method, `equal_range()` which returns two iterators to a range that provides all of the elements with the specified key value.

## **Container Adapters**

In addition to the containers provided by the Standard Template Library, there are a few container adapters. These use a regular STL container to do the heavy lifting for a specialized interface.

## *queue*

A queue adapter normally sits on top of a deque container. Conceptually, items go in the back of the queue and come out the front, much like getting into the ticket line at the movies. This is sometimes called FIFO, or first-in, first-out processing. It provides the methods `front()`, `back()`, `push_back()`, and `pop_front()`.

## *priority\_queue*

The `priority_queue` adapter works a lot like the queue adapter with one significant difference: the first item out is not the first item in. Instead, the first item out is the one with the highest priority. The two containers that can be used as the backend – to do all of the heavy lifting – are the `vector`, which is the default, and the `deque`.

## *stack*

The `stack` can sit on top of a deque, `vector`, or `list`. It is used for LIFO processing, as in last-in, first-out. It can be likened to a stack of trays in a cafeteria: the last tray placed on the stack is also the first one to be taken off.

## **Common STL Container Methods**

There are a lot of different containers provided by the STL, and sometimes it's hard to remember which is which. Thankfully, there is a common set of methods for all of them.

```
iterator begin();  
const_iterator begin() const;
```

Returns an iterator which refers to the first element in the container. Because the associative containers sort their elements in ascending order, `begin()` will return the item with the lowest value.

```
iterator end();  
const_iterator end() const;
```

Returns an iterator which refers to an element which is past the end of the elements in the container – this is not the same as the last element. This method is most often used in loops, particularly for loops.

```
iterator rbegin();  
const_iterator rbegin() const;  
iterator rend();  
const_iterator rend() const;
```

These methods do the same job as `begin()` and `end()` but starting at the end of the container and working toward the beginning. `rbegin()` returns the last element of the container and `rend()` returns an iterator which is before the first element. These methods correspond with using a `reverse_iterator` in a loop instead of a regular one.

```
size_type size() const;
```

This is the number of objects the container holds.

```
size_type max_size() const;
```

`max_size()` returns the maximum number of objects that the container can hold based on limitations imposed by the system.

```
bool empty() const;
```

Returns true if the container has zero elements.

```
void resize(size_type newSize, T from = T());
```

The container is resized to hold `newSize` elements. If this is fewer than the current number, then the excess elements are dropped. If the new size is greater, then new elements are created from the object passed as the parameter `from`. If none is specified, the default constructor for the container object's type is used. This method is available only to sequential containers like `vector`.

```
reference front();  
const reference front() const;  
reference back();  
const reference back() const;
```

These methods return the first or last element of the container, respectively. This is not the same as the iterator returned by `begin()` or `rbegin()`. These methods are not available to associative containers like `map`.

*vector, deque*

```
reference operator[size_type index];  
const_reference operator[size_type index] const;
```

*map*

```
T & operator[const key_type &key];
```

Using the array operator on a `deque`, `vector`, or `map` returns the item with the specified index. In the case of `map`, this is the value at the specified key. If a `map` does not have a value at the specified key, it is created with an empty value. This operator is available for only these three containers.

```
reference at(size_type index);  
const_reference at(size_type index) const;
```

`at()` works just like the array operator with two key differences: it's only available to the `deque` and `vector` containers and if an index is used that is out of the container's bounds, `at()` throws an `out_of_bounds` exception.

```
template class<InputIterator>  
void assign(InputIterator first, InputIterator last);  
void assign(size_type newSize, const T &from);
```

`assign` is a wonderful way to set items in a sequential container to a value all at once or to copy one container into another. The first version takes iterators from

another container and copies those elements into the owning container starting with `first` through, but not including, `last`. The second version sets all elements to the value specified by `from`. In both cases, the container is resized to `newSize` or the number of elements specified by the range in the iterator version.

```
iterator insert(iterator pos, const T &item);
template <class InputIterator>
void insert(iterator pos, InputIterator first, InputIterator last);
```

*vector, deque, and list only*

```
void insert(iterator pos, size_type count, const T &item);
```

*map and set only*

```
pair<iterator, bool> insert(const value_type &item);
```

*multimap and multiset only*

```
iterator insert (const value_type &item);
```

`insert()` adds another item to a container. This method is common to all STL containers, although each container has an additional form of the call unique to itself. The speed of an item insertion depends on the implementation of the container. For example, adding an item in the middle of a vector is potentially slow, but adding an item to its end is quite fast.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

*set, multiset, map, and multimap only*

```
size_type erase(const key_type &lookupValue);
```

`erase()` deletes an item from a container. Like `insert()`, performance of this call depends on the implementation of the container.

```
swap(<container to swap with>);
```

This call takes another container of the same type and swaps the items between the two containers. All pointers, references, and other outside data related to the items kept in the two containers remain valid.

```
void clear();
```

In one fell swoop, this call deletes all items in the container, making it completely empty.

```
void push_front(const T &item);
void pop_front();
```

These two calls enable you to add or remove an item from the front of a deque or list. `pop_front()` not only removes the item from the container, it also deletes the item, too.

```
void push_back(const T &item);  
void pop_back();
```

These two calls perform the same operations as the front versions, but they operate on the items at the back. However, these calls are available for the vector container, as well as deque and list.

```
key_compare key_comp() const;
```

This returns the object used to compare items in the container. It can either be a pointer to a function or an instance of a class which implements the function call operator. The comparison function tests two objects of the container's type and returns true if the first item is to be less than or to be placed before the second item in the container or false in any other case. This function is only available to associative containers.

```
value_compare val_comp() const;
```

This works just like key\_comp() except that it returns the function used to compare two values. For the set container, this is the same as key\_comp(). This function is only available to associative containers.

```
iterator find(const key_type &lookupValue) const;
```

Searches the container for the element which matches lookupValue and returns an iterator which points to it or end() if not found. This function is only available to associative containers.

```
size_type count(const key_type &lookupValue) const;
```

Returns the number of elements in the container which match the lookup value. Although this method is available to all associative containers, it only makes sense to use it on multiset and multimap, because map and set require their lookup values to be unique.

```
iterator lower_bound(const key_type &lookupValue);  
const_iterator lower_bound(const key_type &lookupValue) const;  
iterator upper_bound(const key_type &lookupValue) const;  
pair<iterator, iterator> equal_range(const key_type &lookupValue) const;
```

lower\_bound() returns an iterator which points to the first item which is greater than or equal to the lookup value. upper\_bound() returns an iterator to the first item which is greater than the lookup value. equal\_range() returns two iterators, the first of which is the same as lower\_bound(lookupValue) and the second is the same as upper\_bound(lookupValue). Like count(), these methods are available to all associative containers, but they only make sense to use with multiset and multimap.

## ***STL and the Standard Library: So What?***

Having taken a whirlwind tour through a lot of different template classes after only a short introduction to namespaces and templates, this might all be a little overwhelming. Not to worry. These don't have to be used as often as you might think. Haiku has a class internal to Tracker called `BObjectList` which provides all of the ease-of-use of the `BList` class with the optional ability to have it handle memory management. This covers the need for indexed storage. The `map` and `multimap` containers do really well for random access containers. The remaining containers are more for reference in specialized instances, cross-platform programming, and recognizing them in others' code. They also are necessary when nesting a container within another, such as a `map` of `vectors`.

The same can be said for the Standard C++ library. Some of it will come in handy for Haiku development, and some of it won't be immediately useful. Their usefulness partly depends on how much development you do on other platforms, such as Linux or Windows. If you plan on programming just for Haiku, you probably won't use much of the standard library or the STL, but if you also develop for other platforms, using these will ease the transitions made between platforms.

# Programming with Haiku

## Lesson 3

Written by DarkWorm



In this final lesson on the C++ language, we will be learning about another way to handle errors using language constructs called exceptions and object-oriented file manipulation using streams.

## **C++ Input / Output Streams**

C developers are quite familiar with using `open()`, `close()`, `fprintf()`, and a host of other functions to read and write files and print text to the screen. C++ developers have a similar set of language constructs which make for clearer file operations. You might remember that the `BString` class overloads the `<<` operator to append information as a string. That usage comes from how C++ deals with file operations.

The way that C++ interacts with I/O streams is more flexible and looks less cluttered. It uses double arrow operators for interaction with streams. `<<` is used for writing to a stream and `>>` is used for reading. The standard pipes `stdout`, `stdin`, and `stderr` have been replaced with `cout`, `cin`, and `cerr`, respectively, plus an additional output stream for logging, `clog`. Their capabilities and uses have not changed, however. The collection of functions have been turned into methods which can be overloaded when necessary. In short, the C++ way makes it possible to learn a small set of methods which are much more flexible than their C counterparts.

Let's compare the two ways of working with files. Here is an example of a C++ program which uses C functions to read a file and print it to console.

```
#include <stdio.h>

int
FileExists(const char *path)
{
    // This function tests for the existence of a file by trying
    // to open it. There are better ways of doing this, but this
    // will work well enough for our purposes for the moment.

    // If we were given a NULL pointer, we will return a -1 to
    // indicate an error condition.
    if (!path)
        return -1;

    // Attempt to open the file for reading.
    FILE *file = fopen(path, "r");

    // Our return value will be 1 if the file exists and 0 if
    // it doesn't. ferror() will return a nonzero result if
    // there was a problem opening the file and a 0 if the file
    // opened OK.
    int returnValue;

    // ferror will crash if given a NULL pointer.
    if (!file || ferror(file) != 0)
        returnValue = 0;
    else
    {
        fclose(file);
        returnValue = 1;
    }
}
```

```

    return returnValue;
}

int
MakeTestFile(const char *path)
{
    // Always check for NULL pointers when dealing with strings.
    if (!path)
        return -1;

    // Open the file and erase the contents if it already exists.
    FILE *file = fopen(path, "w");

    if (!file || ferror(file))
    {
        // We have a different error code if we couldn't create the
        // file. This makes it possible for us to know if we messed
        // up by passing a NULL pointer or if there was a
        // file-related error.
        fprintf(stderr, "Couldn't create the file %s\n", path);
        return 0;
    }

    // The stream handles for stdout, stdin, and stderr are already
    // defined for us, so we can use them without any extra work, like
    // in the if() condition above.
    fprintf(file, "This is a file.\nThis is only a file.\n"
        "Had this been a real emergency, do you think I'd "
        "be around to tell you?\n");

    fclose(file);
    return 1;
}

int
main(void)
{
    int returnValue = 0;

    // Let's use a test file in /boot/home called MyTestFile.txt.
    const char *filePath = "/boot/home/MyTestFile.txt";

    // Make the test file if it doesn't already exist and bail out of
    // our program entirely if there is a problem creating it.
    if (!FileExists(filePath))
    {
        returnValue = MakeTestFile(filePath);
        if (returnValue != 1)
            return returnValue;
    }

    printf("Printing file %s:\n", filePath);

    // We got this far, so it's safe to print the file
    FILE *file = fopen(filePath, "r");

    if (!file || ferror(file))

```

```

    {
        fprintf(stderr, "Couldn't print the file %s\n", filePath);
        return 0;
    }

    char inString[1024];

    // fgets will return a NULL pointer when it reaches the end of the
    // file, so this little loop will print the entire file and quit
    // at its end.
    while (fgets(inString, 1024, file))
        fprintf(stdout, "%s", inString);

    fclose(file);

    return 0;
}

```

If this program were to be rewritten in C++, it might look something like this:

```

#include <fstream>
#include <iostream>

using namespace std;

int
FileExists(string path)
{
    // This function tests for the existence of a file by trying
    // to open it. There are better ways of doing this, but this
    // will work well enough for our purposes for the moment.

    if (path.empty())
        return -1;

    // Attempt to open the file for reading
    ifstream file;
    file.open(path.c_str());

    // good() returns true if everything's hunky-dory with our file.
    return file.good();
}

int
MakeTestFile(string path)
{
    // Always check for NULL pointers when dealing with strings.
    if (path.empty())
        return -1;

    // Open the file and erase the contents if it already exists.
    ofstream outFile;
    outFile.open(path.c_str());

    // Check to see if we hit any problems.
    if (!outFile)
    {
        // The endl constant is a special constant which represents

```

```

        // an end-of-line character. By using the endl constant
        // instead of just a '\n' sequence, we are programming for
        // portability without thinking about it.
        cerr << "Couldn't create the file " << path << endl;
        return 0;
    }

    outFile << "This is a file." << endl
            << "This is only a file." << endl
            << "Had this been a real emergency, do you think I'd "
            << "be around to tell you?" << endl;

    outFile.close();
    return 1;
}

int
main(void)
{
    int returnValue = 0;

    // Let's use a test file in /boot/home called MyTestFile.txt.
    string filePath("/boot/home/MyTestFile.txt");

    // Make the test file if it doesn't already exist and bail out of
    // our program entirely if there is a problem creating it.
    if (!FileExists(filePath))
    {
        returnValue = MakeTestFile(filePath);
        if (returnValue != 1)
            return returnValue;
    }

    cout << "Printing file " << filePath << ": " << endl;

    // We got this far, so it's safe to print the file.
    ifstream inFile;
    inFile.open(filePath.c_str());

    if (!inFile)
    {
        cerr << "Couldn't print the file %s" << filePath << endl;
        return 0;
    }

    string inString;

    getline(inFile, inString);
    while (!inString.empty())
    {
        // getline() strips end-of-line characters, so we need to
        // add one.
        cout << inString << endl;
        getline(inFile, inString);
    }

    inFile.close();
}

```

```
    return 0;
}
```

It looks a little strange, but both versions accomplish the same task. The difference is the potential to do more. The same interface can be used to operate on strings using the `istringstream` and `ostringstream` classes. New `istream` subclasses can be created to operate in new ways, and the `<<` and `>>` operators can be overloaded to better integrate our own classes with C++ streams. There are many cases where we can utilize methods which take the burden of memory management off our backs, such as `getline()`. Let's take a look at some of the methods available to us from the `istream` and `ostream` classes.

### *istream Methods*

operator `>>`

Extract information from the stream. This is similar to `fscanf()` or `sscanf()`.

```
streamsize gcount() const;
```

Returns the number of bytes read during the last `get()` or `read()` operation.

```
int get();
int peek();
```

Both methods get one character from the stream and return it. `peek()` does so without actually moving the stream read pointer.

```
istream & get(char &c);
```

Gets one character from the stream and places it in `c`.

```
istream & get(char *string, streamsize count);
istream & get(char *string, streamsize count, char delimiter);
```

Reads characters from the stream until one of the following conditions is met: it has read `count - 1` characters, it encounters the end of the file, or, in the case of the `delimiter` version, it encounters the `delimiter` character.

```
istream & getline(char *string, streamsize count);
istream & getline(char *string, streamsize count, char delimiter);
```

Reads one line from the stream, up to `count` characters or until `delimiter` is encountered.

```
istream & read(char *buffer, streamsize count);
```

Reads `count` bytes from the stream unless the end of the file is reached.

```
streampos tellg();
istream & seekg(streampos position);
istream & seekg(streamoff offset, ios_base::seekdir direction);
```

These methods get and set the position for the next `get()` call. The position to be set can be either absolute (position) or relative to the current one (offset, direction).

### *ostream Methods*

operator `<<`

Write to the stream with formatting, like `fprintf()` or `sprintf()`.

```
ostream & put(char c);
```

Write a character to the stream.

```
ostream & write(char *string, streamsize count);
```

Write a string of count length to the stream.

```
streampos tellp();  
ostream & seekp(streampos pos);  
ostream & seekp(streamoff offset, ios_base::seekdir direction);
```

These methods get and set the position for the next `put()` or `write()` call. The position to be set can be either absolute (position) or relative to the current one (offset, direction).

### *Methods Common to istream and ostream*

```
bool good() const;  
bool bad() const;  
bool fail() const;  
bool operator ! () const;  
bool eof() const;
```

These functions deal with the error state of the stream. When a file stream comes to its end, the `eof` flag is set, whose state is returned by the `eof()` method. The `bad` flag for a stream is set when an error has occurred which affects the integrity of the stream. The `fail` flag is set when individual operations fail for one reason or another, while the `bad` flag tends to be set for problems that affect operations in general. `bad()` returns true when the `bad` flag is set, but `fail()` returns true when either `bad` or `fail` is set. The `!` operator does the same thing. `good()` isn't really the opposite of `bad()`. Instead, it returns false whenever any of the error flags are set, `bad`, `fail`, or `eof`. In short, if you're reading a file, looping over `while(myStream.good())` will be sufficient to ensure that you can read the file.

```
streamsize width() const;  
streamsize width(streamsize wide);
```

Get or set the width of a field. If you are looking to print with a set alignment or a fixed width, you'll need these methods.

```
char fill() const;
char fill(char c);
```

Set or get the fill character used to pad for alignment or fixed-width formatting. The default is a space.

Wow. That's a lot of methods. Luckily, only a handful of these are necessary for just basic file operations like reading or writing a file. The methods in the list above aren't even the full list of available methods. They are just the you'll likely ever need in day-to-day coding.

## **Formatting with C++ Streams**

One example of the increased flexibility that C++ streams provide is formatting. `printf()` and its brethren provide a wealth of formatting options, but `cin` and `cout` have more. They are used the same way that `endl` is used to add an end-of-line character.

Most stream manipulators, unlike `endl`, actually modify the state of the stream to which they are applied. For example, the `boolalpha` manipulator causes boolean values to be converted to their string equivalents. Considering that they are normally converted to the string equivalents of their numeric values (1 for true, 0 for false), this is a nice convenience. By sending a `boolalpha` manipulator to a stream, all future boolean values are converted to "true" or "false." This mode can be turned off by sending a `noboolalpha` manipulator to the stream.

```
#include <iostream>

using namespace std;

int
main(void)
{
    // The hex manipulator causes integers to be displayed in hexadecimal
    cout << boolalpha << hex;
    cout << true << endl;
    cout << 123 << endl;

    // Manipulators can be sent inline like this or on their own,
    // like the above.
    cout << noboolalpha << true << endl;

    return 0;
}
```

The output from this program looks like this:

```
true
7b
1
```

Here is a list of some of the other available manipulators.

Manipulator	Description
boolalpha, noboolalpha	Turns on/off conversion of boolean values to their string equivalents, i.e. true → "true".
dec, hex, oct	Set the numeric base mode to decimal, hexadecimal, or octal.
flush	Flushes the file buffer. Any data which is waiting to be written to the stream is written.
skipws, noskipws	Toggles whitespace skipping. When enabled, this causes reads to skip over tabs, spaces, and newlines. This mode is a huge time-saver when reading configuration files.
showbase, noshowbase	Prepend numbers based on their numeric base. Hexadecimal numbers begin with 0x and octal numbers begin with 0. Decimal numbers have no prefix.

This is by no means a complete list. A more-complete list may be found in any C++ reference, such as that provided at <http://www.cplusplus.org/>

## C++ Exceptions

Exceptions are a way of building error handling into our programs. Haiku doesn't generally use them because they incur a significant performance hit, but knowing about them is nonetheless important.

Use of exceptions centers around three different C++ language elements: try blocks, throw statements, and catch blocks. When a section of code may run into an exceptional error condition, it is placed into a try block. Should it have a problem, it will throw an exception. This causes code execution to move up the series of nested function calls, known as the **call stack**, until it encounters a catch block which handles the kind of exception raised. If it gets to the top of the call stack and still remains unhandled, your program will abort. Here is an example of how exceptions are used:

```
#include <iostream>

using namespace std;

void
SomeFunction(void)
{
    // Let's say something unexpected happens. We'll throw an exception.
    throw 10;
}

int
main(void)
{
    try
    {
        // We placed a function which might raise an exception inside
        // this try block. Should one occur, it will be handled in
        // the catch block.
    }
}
```

```

        SomeFunction();
    }
    catch (int error)
    {
        // Should an exception get to the top of the call stack, it
        // causes the program to completely abort, which is a bad
        // thing. If you have a try block, you should definitely
        // have a catch block after it.
        cout << "An unusual error occurred. Exception number "
             << error << endl;
    }

    return 0;
}

```

Exceptions are rarely used in Haiku programming because the API provides sufficient error-handling capabilities and, as mentioned before, because exception-handling causes a significant performance hit.

### ***Going Further***

- Look over the other manipulators in another reference. What kinds of cool things could be done with them?
- If you were going to design a simple memory database with records having a fixed size, how could you represent different kinds of data? What kind of STL containers could you use? How would you read a record? How would you go about writing one? Deleting one? How about saving and loading the thing?

# Programming with Haiku

## Lesson 4

Written by DarkWorm



## **Source Control: What is It?**

In my early days as a developer on the Haiku project I had troubles on occasion because I had to use a source control manager (SCM). I didn't understand it and I didn't want to take the time to learn about it from some tutorial online. I wanted to be able to write code with as few hurdles as possible. How I wish that I'd understood source control then.

**Source control**, also known as **revision control** or **version control**, is a tool or set of tools which facilitates development on a single codebase by many people at once. This is done by recording the changes that are made by each person and ensuring that one person's changes cannot be applied at the same time as another's. Most also provide for working on a separate copy of the main sources, called a **branch**.

Using an SCM forces your workflow to have some structure, which is actually a good thing for those who have a hard time getting organized, provided that they are willing to work with it. Day-to-day coding involves checking out others' updates and checking in your own. On occasion, a change must be undone, called **reverting** a change. Sometimes a feature is large enough that it necessitates working over the course of several check-ins, called **commits**. In these cases, a branch is created so that the development of the feature benefits from source control without disturbing others' work. When the feature is completed, it is merged back into the main part of the tree.

## **Source Control: Why Use It?**

The benefits of using an SCM as part of your workflow far outweigh any drawbacks almost 100% of the time. If you are part of a project with more than one person, the automation of change tracking frees up time to spend on other tasks. It also removes the human factor of applying changes. While it might be possible for two or three people to work together by e-mailing files to each other, the potential for mistakes is great and it cannot be easily sustained for an extended period of time. The ability to undo changes with a command or two is a major time-saver.

A few drawbacks accompany using source control. Setting up a project in source control can take some time. Changing from one SCM to another is not always easy. Choosing a source control manager isn't easy – there are many available and the advantages of each are not always clear. Setting up your own SCM server, such as for a business or for private hosting of a closed source project, has challenges of its own as well. These headaches are considered minor by most veteran coders because often they have had at least one instance where source control got them out of a jam.

## **Source Control: Which One?**

On other platforms, such as Linux or OS X, there are a plethora of different SCMs from which to choose. As of this writing, there are four available for Haiku, described below.

### ***Concurrent Versioning System (CVS)***

CVS is one of the oldest SCMs still available and in use. It was originally developed as an improvement over the set of tools known as Revision Control System (RCS). It uses a client-server architecture and is still widely used. CVS is considered by some to be not worth using. When compared to other choices, it tends to come up short on features or has limitations that

others do not have, such as not being able to rename or move files. It is not recommended for inexperienced developers.

### *Subversion (SVN)*

Subversion was started with the intention of fixing the problems present in CVS, and is also a widely-popular, well-liked source control tool. Like CVS, Subversion uses a client-server architecture. It supports moving and renaming files, branches are relatively fast, and commits are truly atomic operations. It does have problems, as well, though. One common criticism is that while branching is simple, merging a branch back into the main one does not always work well and can be tedious and/or time-consuming. Overall, it is a good tool and, as of this writing, is the tool used by the Haiku project.

### *Git*

Git was originally developed by Linus Torvalds for work on the Linux kernel. Unlike Subversion and CVS, it uses a distributed model where each developer has a full copy of the repository. It was written to be whatever CVS isn't – to protect against corruption, to be very fast, and to work much like the proprietary SCM BitKeeper. Git has quickly developed a passionate following. One of its main drawbacks is that while there are two implementations for Windows, neither is an ideal solution. While very, very fast, it can also be somewhat confusing while getting accustomed to using it. In many ways, it has a design perspective similar to that of Linux as an operating system.

### *Mercurial (hg)*

Mercurial is a brother-in-arms to Git, having been started at about the same time and for the same basic reason: the company behind BitKeeper, BitMover, withdrew their version of BitKeeper which was free to open source projects not developing a competing SCM. It is written mostly in Python and is available on most major platforms and Haiku. Like Git, it has a loyal following and there is a never-ending controversy between Mercurial users and Git users about which is better, similar to that of two well-known colas. The design perspective for Mercurial could be likened to that of OS X or Haiku – simple, but not oversimplified. It is a recommended starting point for a developer to learn how to use source control and will be the one referenced in the future. Just like a preferred IDE, if you have a different preference, that's just fine.

## ***First Steps Using Source Control***

Although Paladin provides a convenient interface to using source control, it does not go in-depth into some of the finer points of using any of those which it supports. For the purposes of this lesson, we will just cover the basics of using Mercurial from the command line.

The first task on the order of business is to tell Mercurial who you are. Create a file in your home folder called `.hgrc`, open it in a text editor, and place the following contents in it:

```
[ui]
username = My Name <myemail@foo.com>
```

Of course, you will want to substitute your own name and e-mail address. Also, unless you prefer the nano editor included in Haiku, you'll want to set the text editor used by Mercurial when you describe your changes when you check them in. My preference is to use Pe, an

excellent text editor written specifically for Haiku. To use Pe as your text editor, create a file in your home folder called `.profile` if it doesn't already exist and add this line to it:

```
export EDITOR=lpe
```

If you have a different editor you'd prefer to use, substitute the name of your preferred editor for `lpe`. If the executable is not in `/boot/system/bin`, `/boot/common/bin`, or `/boot/home/config/bin`, you'll need to use the full path to the executable. Your `.profile` file is a Bash script executed at the beginning of each Terminal session, so you can place other customizations here, too, if you like.

Now that the initial setup is done, let's make a repository. Mercurial can be used to add source control to a new project or an existing one. Open a Terminal window, go to an empty folder, and enter this command:

```
$ hg init
```

This uses the current directory as the top folder in the repository. All subfolders are considered part of it, although empty directories will be ignored. You won't see Mercurial print anything after typing the command, but if you issue an `ls -a` command, you'll see that a `.hg` directory was created – this folder is where Mercurial keeps all of its information for the entire repository. While you're at it, create or copy over a couple of test text files in the directory that we'll use in a moment.

Now we will add your test files to the repository. Mercurial and other SCMs will track only those files which you have told it to track, although they will tell you if they see files they don't recognize. Adding files to Mercurial's list of files to manage is simply another command:

```
$ hg add
adding ObjectArray.h
adding foo.cpp
```

All files in the current folder and all subfolders will be added to the repository. Of course, you can specify a file or list of files after the word `add`, instead. Adding files to the repository doesn't change anything unless you check in the changes.

Before we make our first commit, it's often good to make sure you know ahead of time what changes have been made. This can be done in one of two ways. The first way is to use `hg status`. This will print out a list of file which are not up-to-date, which can be added, removed, modified, or unrecognized files. The other way is `hg diff`, which shows the actual changes made to each file. Here is what `hg status` looks like:

```
$ hg status
A ObjectArray.h
A foo.cpp
```

Two files have been added since the last commit, or in our case, since the creation of the repository. Other codes for files can be found in the following table.

Status Code	Meaning
M	Modified
A	Added
R	Removed
C	Clean
!	Missing – file does not exist, but is still being tracked
?	Not tracked
I	Ignored

If you are not already familiar with the `diff` command used in the Terminal, give the `hg diff` command a try to see how it looks, but be prepared for a lot of text to be printed. It is often best to view a diff using a pager in the Terminal such as `hg diff | less` or doing it from Paladin where you get a scrollable text window.

Before we do our commit, let's examine what we would need to do if something happened that we didn't want to commit to the repository. Let's say that you added a bunch of `printf()` calls to a file for some debugging, but you don't want them going into the tree. Although you could go back and remove them all manually, if these `printf()` calls were the only changes you made to the file, you could revert it.

Reverting a file undoes all changes made to it since the last commit. If you've accidentally added a file that you don't want tracked, it will un-add it. If you accidentally delete a tracked file, Mercurial will replace it with a new copy. Modified files will go back to an unmodified state. In short, it fixes any mistakes you've made, and when you revert a modified file, Mercurial creates a `.orig` file which contains the changes you made in case you want them after all.

A revert can be done in several ways. You can revert just one file or the entire tree. The revert can go back to a specific revision. It can also be done without backing up the changes made. Here are some of the options for `hg revert`:

Command	Action
<code>hg revert --all</code>	Reverts all files in the tree
<code>hg revert myFile.cpp</code>	Reverts myFile.cpp. Changes will be backed up to myFile.cpp.orig
<code>hg revert --no-backup myFile.cpp</code>	Reverts myFile.cpp, but makes no backup
<code>hg revert -r d8787f07dd69 --all-files --no-backup</code>	Reverts the entire source tree back to the specified revision (changeset), making no backups to changes

Considering the myriad ways that a developer can make mistakes, knowing the different ways that `revert` works can save you time, effort, and stress.

Let's move on and check in our changes. Enter this command to initiate a commit:

```
$ hg commit
```

After entering this command, Mercurial will open an editor to allow you to add a message to describe the commit. If you didn't specify an editor in your `.profile`, this will be the console text editor `nano`. Use the message "Initial check-in" or something similar, save the file in the editor, and close it. Mercurial may or may not print anything. Rest assured that even if it doesn't, your first commit was successful. You can confirm this, though, with the `hg log` command, which can be done for the entire repository or just certain files.

```
$ hg log
changeset:  0:0dbb51f0e1fa
tag:        tip
user:       DarkWyrm <darkwyrm@gmail.com>
date:       Sun Aug 15 21:30:56 2010 -0400
summary:    Initial commit
```

## ***Working with Others Using Mercurial***

If you're just working on your own project and have no intention of working with others on it, these commands are all you'll need. However, having a project on an open source hosting site such as BitBucket or Sourceforge involves a few others, as well.

Let's pretend for a moment that you have applied for hosting a project called MyProject at a site called MyMercurial. After submitting the application, you are approved and the site has created your repository. Now what?

First, you will need to make a local copy of the repository hosted by MyMercurial. You will need to get the URL of the repository from the hosting site which is specific to the hosting site and your project. For our example, the repository URL is <http://mymercurial.foo/hg/myproject>. To make our local repository, we will use the `hg clone` command. Mercurial will probably print something similar to this:

```
$ hg clone http://mymercurial.foo/hg/myproject
destination directory: myproject
requesting all changes
adding changesets
adding manifests
adding file changes
added 0 changesets with 0 changes to 0 files
updating to branch default
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

You will now have a subfolder of the current one called `myproject`. It doesn't have any files in it, but it will be easy for us to send changes to the remote one. Most of the rest of the work is exactly the same as what we did a moment ago: copying files for the project into the folder, using the `hg add` command to add files to it, and `hg commit` to check them in.

When working with a project hosted online, there is one extra step in the workflow: pushing changes. Commits only apply to the repository sitting on your hard drive, unlike centralized source control tools like CVS and Subversion. Getting your changes to the online repository is done with `hg push`. The results look similar to this:

```
$ hg push
```

```
pushing to /boot/home/testrepo
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Getting changes from the online repository is similar to publishing them to it. This is done with the command `hg pull`. This grabs the changes from the online repository and downloads them. It does not, however, automatically merge them into your sources unless you add the `-u` switch. Omitting the switch means that once the `pull` is complete, executing an `hg merge` and `hg commit` is needed afterward.

### ***Wrapping it All Up: Source Control in a Nutshell***

Working with source control can seem complicated at first, but concepts carry over from one SCM to another, and the basics don't involve much in most situations. Most of the time, you'll follow a workflow something like this:

1. Write and/or modify code.
2. Commit your changes locally.
3. Repeat steps 1 and 2 as many times as desired. When you're ready to update the online repository, continue to step 4.
4. Pull and merge remote changes.
5. Push your modifications to the remote repository.

When you look at the workflow this way, it doesn't seem very complicated and for a very good reason: it isn't complicated. More advanced source control use, such as using branches, is beyond the scope of this lesson but not much more complicated than the above series of steps.

If source control is so simple, why doesn't everyone use it? In most cases, it is because of ignorance, laziness, or both. Integrating source control into your development workflow will make your work easier and potentially save you from major problems.

# Programming with Haiku

Unit 1 Review

Written by DarkWorm



## **Lesson 1**

1. What are templates?
2. What are templates most often used to create?
3. How is a template-based class declaration different from a regular one? A template-based function declaration?
4. Why must functions which use templates be defined in a header file?
5. What is the difference between a vector and a deque?
6. What is the main drawback of the list container?
7. What is the namespace used by the Standard Template Library?
8. What does the using keyword do?

## **Lesson 2**

1. What is the difference between the map and set containers?
2. What is the purpose of the equal\_range() method used by multimap and multiset?
3. What is FIFO processing and which container adapter is well-suited to it? LIFO processing?
4. What is the difference between the queue and priority\_queue containers?

## **Lesson 3**

1. What are the four standard C++ streams?
2. What is the difference between istream::bad() and istream()::fail()?
3. Why should the endl constant be used instead of '\n' when writing an end-of-line character to a C++ stream?
4. What is the call stack?
5. What does the boolalpha formatter do?
6. Why are exceptions not normally used in Haiku?

## **Lesson 4**

1. Why was the Subversion project started?
2. What setup work should be done before Mercurial is used?
3. What is the purpose of the shell variable EDITOR in relation to Mercurial?
4. What command creates a Mercurial repository?
5. What is the difference between hg commit and hg push?
6. What does hg revert do?

# Programming with Haiku

## Lesson 5

Written by DarkWorm



Let's take some time to put together all of the different bits of code that we've been learning about. Since the first lesson, we have examined the following topics:

- Templates
- Namespaces
- Iterators
- The C++ string class
- The STL Associative containers: map, set, multimap, and multiset
- The STL Sequential containers: vector, deque, and list
- The STL Container Adapters: queue and priority\_queue
- C++ input and output streams, a.k.a cout and friends
- Exceptions

There is enough material here that a book could be written to get a good, strong understanding of effective use of the Standard Template Library and the Standard C++ Library. Expert use is not our goal for this context, but having a good working knowledge of these topics will help make our code better when writing applications for Haiku.

### ***Project: Reading Paladin Projects***

For those unfamiliar, Paladin is one of the Integrated Development Environments available for Haiku. It was designed to have an interface similar to BeIDE, which was the main IDE for BeOS back in the day. One feature that sets it apart from other IDEs for Haiku is that its unique project file format is a text file with a specific format. This makes it possible to easily migrate to and away from it.

The format itself is relatively simple. Each file is a list of key/value entry pairs with one entry per line. This makes it possible to read and interpret the project file progressively. It also increases forward compatibility because new keys can be ignored by older versions of the program.

This project will be a cross-platform project reader. We will be using the Standard C++ Library along with the STL to ensure that it can be used on just about any operating system. The main use of this project would be to serve as a start for a program to convert Paladin projects to makefiles, Jamfiles, shell scripts, or other build systems.

```
#include <fstream>
#include <iostream>
#include <list>
#include <map>
#include <string>
#include <strings.h>
#include <vector>

// Because we're doing so much in the std namespace, this will save
// us a *lot* of extra typing.
using namespace std;

// While these could -- and probably should -- be classes, we will just
// use structures for the sake of space and simplicity.
typedef struct
{
    // File paths can be stored as either absolute paths, obviously
```

```

    // beginning the entry with a /, or as a path relative to the
    // project file's location.
    string path;

    // Dependencies are stored as a list of file paths separated by
    // pipe symbols (|).
    string dependencies;
} ProjectFile;

typedef struct
{
    string name;
    bool expanded;
    vector<ProjectFile> files;
} ProjectGroup;

// The actual Project class used by Paladin is much more complicated
// because it also has some properties for maintaining state while the
// program is running, but this structure has all of the data that is used
// for building and maintaining a project.
typedef struct
{
    map<string,string>                properties;

    vector<string>                    localIncludes;
    vector<string>                    systemIncludes;
    vector<ProjectGroup>              groups;
    vector<string>                    libraries;
} PaladinProject;

int
ReadPaladinProject(const char *path, PaladinProject &proj)
{
    // Create an Input File Stream for reading the file
    ifstream file;

    file.open(path, ifstream::in);
    if (!file.is_open())
    {
        // endl is a cross-platform stand-in for an end-of-line
        // character. The EOL character is different on Windows,
        // Haiku/Linux/UNIX, and OS X and this saves us from
        // having to figure it out without sacrificing portability.
        cout << "Couldn't open the file " << path << endl;
        return -1;
    }

    // Empty the project's data to make sure we're not building
    // upon existing baggage.
    proj.properties.clear();
    proj.localIncludes.clear();
    proj.systemIncludes.clear();
    proj.groups.clear();
}

```

```

while (!file.eof())
{
    string strData;

    // While the fstream class has a getline() method, it only
    // works on regular strings. There is a global getline()
    // function in <string> which reads data from a stream into
    // a C++ string. This is the version that we will use.
    getline(file, strData);

    // An empty line shouldn't exist in a Paladin project, but
    // let's handle the case just to prevent headaches.
    if (strData.empty())
        continue;

    size_t pos = strData.find('=');

    // npos is the maximum size for a C++ string. find() will
    // return npos if the string searched for is not found.
    if (pos == string::npos)
        continue;

    string key = strData.substr(0, pos);
    string value = strData.substr(pos + 1, string::npos);

    if (key.compare("GROUP") == 0)
    {
        // Using vectors saves us from having to worry about
        // memory management and pointers. Instead, all that we
        // have to do is create a new group on the stack which
        // will be used to initialize the new element in the
        // groups vector.
        ProjectGroup newGroup;
        newGroup.name = value;
        proj.groups.push_back(newGroup);
    }
    else if (key.compare("EXPANDGROUP") == 0)
    {
        if (!proj.groups.empty())
            proj.groups.back().expanded =
                strcasecmp(value.c_str(), "yes");
    }
    else if (key.compare("SOURCEFILE") == 0)
    {
        // Quite a few dots are used to create the new file, but
        // that's OK. It sure beats messing around with pointers.
        ProjectFile newFile;
        newFile.path = value;
        proj.groups.back().files.push_back(newFile);
    }
    else if (key.compare("DEPENDENCY") == 0)
        proj.groups.back().files.back().dependencies = value;
    else if (key.compare("LIBRARY") == 0)
        proj.libraries.push_back(value);
    else
        proj.properties[key] = value;
}

```

```

        return 0;
    }

    int
    main(int argc, char **argv)
    {
        PaladinProject project;

        if (argc == 2)
            ReadPaladinProject(argv[1], project);
        else
            cout << "Usage: " << argv[0] << " <path>\n";

        map<string,string>::iterator i;
        for (i = project.properties.begin(); i != project.properties.end();
             i++)
            cout << i->first << ": " << i->second << endl;

        return 0;
    }

```

## Going Further

- Use this project as a starting point for printing information about a Paladin project.
- Create a program which reads a Paladin project and spits out a makefile or Jamfile.

## Unit 1 Review Answers

### Lesson 1

1. Templates are a way of making classes and functions work with types in a generalized way.
2. Templates are most often used to create container classes.
3. A template-based class declaration is preceded by the `template <class T>` declaration placed before the `class MyClass` line. A template-based function places this text before the rest of the function's declaration.
4. Function definitions which use templates must be placed in a header file because the templates are generated at compile time. If this is not done, linker errors ensue.
5. There are actually two main differences between deques and vectors. First, it is not safe to use pointers to refer to the elements in a deque, unlike a vector. Second, it is possible to add elements only to the back of a vector, whereas elements can be added to either the back or front of a deque.
6. The main drawback of the list container is that it is not possible to quickly access an arbitrary element – getting to a specific item involves iterating from its beginning.
7. The Standard Template Library uses the `std` namespace.
8. The `using` keyword allows you to refer to members of a namespace without a prefix.

### Lesson 2

1. The `map` container stores elements as key-value pairs. The `set` container is different in that the values are the same as the keys.

2. `equal_range()` returns two iterators which refer to all items matching a value in a `multiset` or `multimap`.
3. FIFO refers to **F**irst-**I**n, **F**irst-**O**ut processing, which is where the items that are the first to go into a container are also the first to leave it, much like a line at a movie theater. LIFO stands for **L**ast-**I**n, **F**irst-**O**ut processing, where the last items into a container are the first to leave. The Towers of Hanoi puzzle exemplifies this.
4. The difference between a queue and a `priority_queue` is that the item with the highest priority is the first item out of a `priority_queue`, which does not necessarily translate to the the first item in. A regular queue is strictly FIFO.

### *Lesson 3*

1. The four standard C++ streams are `cin`, `cout`, `cerr`, and `clog`.
2. `fail()` usually applies to the failure of an individual call, such as a write `fail`. `bad()` is normally true when there is an error condition in the state of things that applies to more than just an individual method failure.
3. `endl` should be used instead of just `'\n'` because `endl` is portable across operating systems – not every operating system uses `'\n'` as its end-of-line character.
4. The call stack is the series of nested function calls in a program, starting with `main()`.
5. The `boolalpha` formatter translates boolean values to "true" and "false" strings.
6. Exceptions are not normally used in Haiku for performance reasons – unrolling the call stack is slow and the error handling provided by the API is normally sufficient.

### *Lesson 4*

7. The Subversion project was started to correct the problems inherent in the CVS version control system.
8. Setting your name and e-mail address in the `~/.hgrc` file must be done before doing version control with Mercurial.
9. If the `EDITOR` shell variable has been set, Mercurial will use it to edit commit messages.
10. The `hg init` command creates a Mercurial repository.
11. `hg commit` checks in changes to the local repository. `hg push` sends them from the local repository to a remote one.
12. `hg revert` undoes the changes made to one or more files since the last check-in. It can also undo changes back to a specific revision.

# Programming with Haiku

## Lesson 12

Written by DarkWyrn



## Attributes

The Be filesystem that Haiku uses allows files to have extended attributes. These attributes are **metadata** – information about a file that is not part of the file's contents. Although a number of other operating systems feature this in their filesystems, none of them have pervasive use of them like Haiku does.

The flexibility that attributes provide is not necessarily obvious at the start. For example, MP3 files can have different attributes, such as artist, album, and so forth. No special tag-reading code is needed for MP3s processed this way. E-mails in Haiku are stored as individual files, so it is trivial to search, for example, for all e-mail from a person which was received after a certain date. Person files, which are used to store contact information, are actually zero-byte files which have all of their information stored in attributes. Some contact managers, such as Mr. Peeps!, have expanded the number of attributes dedicated to a person file and even made them store regular data.

The main C++ functions used to manipulate attributes are methods provided by the BNode class.

```
status_t GetAttrInfo(const char *name, attr_info *info) const;
```

Gets the type and size of the attribute named name and places it into the corresponding properties of info, which must be already allocated when the call is made. B\_ENTRY\_NOT\_FOUND is returned if the node does not have the specified attribute and B\_FILE\_ERROR is returned if the BNode is uninitialized. B\_OK is returned on success. Below is the declaration for the attr\_info structure:

```
typedef struct attr_info
{
    uint32    type;
    off_t     size;
} attr_info;
```

```
ssize_t ReadAttr(const char *name, type_code type, off_t offset,
                void *buffer, size_t length);
```

ReadAttr() reads the data from attribute name and copies up to length bytes into buffer. As of this writing, offset is not used. The type parameter can, like BMessage identifier constants, be any 4-byte integer value, but it is normally one of the predefined constants found in TypeConstants.h, such as B\_STRING\_TYPE or B\_INT32\_TYPE. The number of bytes actually read is returned.

```
ssize_t WriteAttr(const char *name, type_code type, off_t offset,
                 void *buffer, size_t length);
```

Dealing with WriteAttr() is almost exactly the same as with ReadAttr() except that the attribute is erased and the data in buffer replaces it. **Note: Indexed attributes**, which are those that can be accessed via a query, **may be no bigger than 255 bytes**. They also must be one of the following types: string, int32, uint32, int64, uint64, float, or double.

```
status_t RemoveAttr(const char *name);
```

This method deletes the specified attribute and returns B\_OK if successful.

```
status_t RenameAttr(const char *oldname, const char *newname);
```

This performs more of a move than a rename. Beware that if there is an existing attribute named newname, it will be clobbered.

```
status_t ReadAttrString(const char *name, BString *out) const;
status_t WriteAttrString(const char *name, const BString *data);
```

These undocumented functions are wildly convenient if you work with string attributes. The data in the named string is placed in out in the read version and data is written to disk in the write version.

Here is an example of some code which reads an attribute. In this example, we read the E-mail Name attribute.

```
#include <fs_attr.h>
#include <Node.h>
#include <String.h>

BString
GetEmailName(const char *path)
{
    BString out;
    BNode node(path);
    if (node.InitCheck() != B_OK)
        return out;

    // This is to make sure that the attribute exists and what
    // its size is.
    attr_info attrInfo;
    if (node.GetAttrInfo("META:name", &attrInfo) != B_OK)
        return out;

    // BString::LockBuffer() and UnlockBuffer() allow us to have
    // direct access to the internal character buffer that the BString
    // uses. LockBuffer() takes one parameter which is the maximum
    // size that the character buffer needs to be.
    char *nameBuffer = out.LockBuffer(attrInfo.size + 1);
    node.ReadAttr("META:name", attrInfo.type, 0, nameBuffer,
                 attrInfo.size);
    nameBuffer[attrInfo.size] = '\0';
    out.UnlockBuffer();

    return out;
}
```

For arbitrary attributes, this works quite well. However, there are certain attributes which are system standards for which a class was created to simplify our lives and save us poor, helpless developers from having to remember these most common attribute names. This includes icons and file types. This help comes in the form of the BNodeInfo class.

## ***BNodeInfo***

```
status_t GetAppHint(entry_ref *app_ref);
status_t SetAppHint(const entry_ref app_ref);
```

The app hint for a file suggests to the system which application should be used to open this particular file. `app_ref` is considered a hint because it may end up pointing to something that isn't an application or some other problem that would prevent it from opening the file. For those curious, this information is stored in the attribute "BEOS:PPATH". These two methods aren't used very often.

```
status_t GetIcon(BBitmap *icon, icon_size size = B_LARGE_ICON);
status_t SetIcon(const BBitmap *icon, icon_size size = B_LARGE_ICON);

status_t GetIcon(uint8 **data, size_t *size, type_code *type) const;
status_t SetIcon(const uint8 *data, size_t size);

static status_t GetTrackerIcon(entry_ref *ref, BBitmap *icon,
                               icon_size which = B_LARGE_ICON)
```

The first two of these methods act upon the attribute of the file. "BEOS:M:STD\_ICON" holds a 16x16 pixel 256-color icon and "BEOS:L:STD\_ICON" is used for a 32x32 pixel, 256-color icon. These are standard attributes for all BeOS operating systems. Haiku introduces another one, "BEOS:ICON", which is used to store the vector icon. Unlike the other two, the data for this icon is stored in the Haiku Vector Icon Format (HVIF) format. The versions of these two calls which do not use the BBitmap class were created to work specifically with the vector icon for a file. `GetTrackerIcon()` gets the icon for the file which would be shown by Tracker which may not necessarily be the same icon as what would be returned by `GetIcon()`, which will be better explained in a moment. In most cases, if you want to get the icon for a file, use `GetTrackerIcon()`.

```
status_t GetPreferredApp(char *signature, app_verb = B_OPEN);
status_t SetPreferredApp(char *signature, app_verb = B_OPEN);
```

These two methods deal with the preferred application for a file. This only deals with this specific file and not the file's type in general. The attribute used here is "BEOS:PREF\_APP".

```
status_t GetType(char *type);
status_t SetType(const char *type);
```

Set and get the file's type. This is always a MIME string. Note that if this attribute doesn't exist, calling the global function `update_mime_info()` may help. The attribute on which these methods operate is "BEOS:TYPE".

Note that sometimes it is not as convenient to use the BNodeInfo class' methods as it is with those of BNode, working directly with the attributes. Why? Because BFile is a child class of BNode, it is often easier to reuse an existing BFile object – especially using the `ReadAttrString()` and `WriteAttrString()` methods.

## ***Think Locally, Act Globally***

One of the ways in which Haiku is customizable for more advanced users is the way in which it handles the file attributes we examined a moment ago. The global settings are stored in the system's MIME database and manipulated with the `BMimeType` class. However, individual files can be customized to override these settings.

Let's say for a moment that `.xyz` files are opened with `XYZEdit`. This information can be seen and changed in the FileTypes preferences application or programmatically, but there is one file called `SpecialFile.xyz` which we want to always open with `XYZOtherEdit`. This can be set using the FileType Tracker add-on, which modifies the "BEOS:PREF\_APP" attribute on the individual file. When `SpecialFile.xyz` is double-clicked, Tracker will open it in `XYZOtherEdit`. All other `.xyz` files are opened in the regular editor. If you want to make changes for all `.xyz` files, then using the `BMimeType` class is necessary. We'll study that class in detail later on.

## ***Closing Thoughts***

The existence of attributes in the BFS filesystem is not unique. Other filesystems, such as XFS and ReiserFS have them, as well. What makes Haiku different is that the major operating systems (Linux, Windows, OS X) do not leverage them extensively, which is sad if you consider the many ways that they can be used. If you poke around in Haiku, you'll find them used for many different tasks and in different ways. Next time we'll look at one powerful operating system feature which centers around them: queries.

## ***Going Further***

- Using the Terminal commands `listattr` and `catattr`, see what attributes are used by these kinds of files: People file, an application, an e-mail, and an MP3.
- If you were going to create a Task file for a to-do list application, the Haiku way of storing all of a task's details would be using attributes. Using the general naming scheme from an e-mail's attributes ("MAIL:subject", etc.) as a guide, what would be the names and types of some of the attributes you would use for a task file's information?

# Programming with Haiku

## Lesson 13

Written by DarkWyrn



## Queries

Haiku's parent operating system, BeOS, was far ahead of its time and possessed advanced features that were missing from Windows 95 and MacOS, the competing operating systems of its day. In addition to extended file attributes, it also provided query support. Queries are lightning-fast file searches which leverage Haiku's unique filesystem: BFS. While other operating systems were recursively searching every single file for matches within filenames and the like, BeOS users were using queries to organize MP3s, contacts, e-mail, and photos. Haiku users still do all of this, but it's often left underused except by advanced users and developers. Even now more popular operating systems cannot search for files as quickly and easily with the same power without taxing the rest of the operating system.

## Query Syntax

Queries are usually run via Tracker's Find window. Most of them use the *by name* or *by attribute* modes. Hardcore Haiku users love the *by formula* setting, which uses the actual syntax of a filesystem query without all the niceties. Fortunately, the syntax of a query is relatively simple and can be learned quickly by entering a test query in one of the other modes and then changing to the formula mode.

Let's start with a simple query: find all new e-mail.

1. Open Tracker's Find window by pressing Command-F or choosing *Find...* from the Deskbar or the File menu in any Tracker window.
2. Instead of leaving it set to *All files and folders*, change the file type to *E-mail* in the text submenu.
3. Change *by name* to *by attribute*.
4. Now click on the menu field marked *Name*, choose *is* from the *Status* submenu, and type the word *new* into the text box. This is what the query will search for, but take a look at the query's real syntax by changing the mode to *by formula*.

Having done all of this, you will see the following text in the Find window:

```
((MAIL:status=="[nN][eE][wW]")&&(BEOS:TYPE=="text/x-email"))
```

For those unfamiliar with regular expressions, a set of characters within a pair of square brackets matches any single character in that set. `[nN]` matches the letter N, regardless of case, for example. This query will match any e-mail which has a status attribute of *new* without paying any attention to capitalization. Queries are case-sensitive, but any query generated with the name or attribute modes are converted so that they are not case-sensitive.

Queries can be quite complicated, but they don't have to be that way. A simpler query which would get the job done as well as the one above could look like this:

```
MAIL:status=="[nN]ew"
```

Since e-mail files are the only files which have the `MAIL:status` attribute, just searching for the attribute makes more sense.

If a query is to be run on a particular attribute, it must be indexed. The Terminal commands `lsindex`, `rindex`, and `mkdirindex` manipulate and list which attributes are indexed. Note that when an attribute is added to the filesystem's index with the `mkdirindex` command, all files having that attribute are not automatically found by a query. They must be reindexed. This can be accomplished by copying the file somewhere and then copying it back or using the `reindex` Terminal command. *A word of warning to the wise:* don't be afraid to add an attribute to the index, but don't go crazy over them, either. Adding lots of them will slow down overall performance of all query-based searches.

## Using BQuery

There are two ways that BQuery can be used to run a query. The simpler way is really easy if you understand the actual query syntax. The more complicated way is more flexible, but not necessary in most cases. We'll examine the simpler method first. Here is some very basic code for running a query on the boot volume:

```
#include <Query.h>
#include <stdio.h>
#include <String.h>
#include <VolumeRoster.h>
#include <Volume.h>

int
main(void)
{
    // Get a BVolume object which represents the boot volume. A BQuery
    // object will require one.
    BVolumeRoster volRoster;
    BVolume bootVolume;
    volRoster.GetBootVolume(&bootVolume);

    // A quick summary for using BQuery:
    // 1) Make the BQuery object.
    // 2) Set the target volume with SetVolume().
    // 3) Specify the search terms with SetPredicate().
    // 4) Call Fetch() to start the search
    // 5) Iterate over the results list using GetNextEntry(),
    //    GetNextRef(), or GetNextDirents().

    BString predicate("MAIL:subject == *");
    BQuery query;
    query.SetVolume(&bootVolume);
    query.SetPredicate(predicate.String());
    if (query.Fetch() == B_OK)
    {
        printf("Results of query \"%s\":\n", predicate.String());
        entry_ref ref;
        while (query.GetNextRef(&ref) == B_OK)
            printf("\t%s\n", ref.name);
    }
}
```

With code this easy, you would think that queries would be used more in third party applications!

The more flexible method is more complicated because of the way that the query is assembled. It uses a token stack – chaining together individual components of the search terms in a specific order called Reverse Polish Notation. Each element is added to the predicate using methods like `PushOp()`, and `PushAttr()`.

The **Reverse Polish Notation (RPN)** entry system dates back to the 1950s, but it can still be found in many different areas, particularly financial and scientific calculators. Also known as **Postfix notation**, it groups mathematical operands together and places the operator at the end. For example, what we would normally write as  $(5 + 6) * 3$  becomes `5 6 + 3 *` in RPN. Parentheses are not necessary if the precedence of mathematical operators is upheld. RPN is very easy for computers, but it can be a real headache for people, having had algebraic entry ingrained into us since elementary school. Luckily for us, queries normally have a simple syntax, so it isn't that much more difficult to construct a query this way.

Here are the methods which are used to construct queries using RPN:

```
void PushAttr(const char *attrName);
void PushOp(query_op operator);

void PushUInt32(uint32 value);
void PushInt32(int32 value);
void PushUInt64(uint64 value);
void PushInt64(int64 value);
void PushFloat(float value);
void PushDouble(double value);
void PushString(const char *string, bool ignoreCase = false);
```

The top two methods are for adding attribute names and comparison operators; the rest are for adding values of different types.

The operators which can be used with `PushOp()` are as follows:

Operator	Operation
B_EQ	==
B_NE	!=
B_GT	>
B_LT	<
B_GE	>=
B_LE	<=
B_CONTAINS	The same as the regular expression <code>*value*</code>
B_BEGINS_WITH	The same as the regular expression <code>*value</code>
B_ENDS_WITH	The same as the regular expression <code>value*</code>
B_AND	&&
B_OR	
B_NOT	!

Modifying the previous query example to use RPN results in this code:

```

#include <Entry.h>
#include <Query.h>
#include <stdio.h>
#include <String.h>
#include <Volume.h>
#include <VolumeRoster.h>

int
main(void)
{
    // Get a BVolume object which represents the boot volume. A BQuery
    // object will require one.
    BVolumeRoster volRoster;
    BVolume bootVolume;
    volRoster.GetBootVolume(&bootVolume);

    // A quick summary for using BQuery:
    // 1) Make the BQuery object.
    // 2) Set the target volume with SetVolume().
    // 3) Specify the search terms with Push*().
    // 4) Call Fetch() to start the search
    // 5) Iterate over the results list using GetNextEntry(),
    //     GetNextRef(), or GetNextDirents().

    BString predicate("MAIL:subject == *");
    BQuery query;
    query.SetVolume(&bootVolume);
    query.PushAttr("MAIL:subject");
    query.PushString("*");
    query.PushOp(B_EQ);
    if (query.Fetch() == B_OK)
    {
        printf("Results of query \"%s\":\n", predicate.String());
        entry_ref ref;
        while (query.GetNextRef(&ref) == B_OK)
            printf("\t%s\n", ref.name);
    }
}

```

Be aware that the two techniques cannot be mixed. Any search terms added to a BQuery by way of a Push method takes precedence over anything passed to SetPredicate().

## ***Searching in Real Time Using Live Queries***

The above code examples use queries statically – the results are read and do not change, but a query can also do live updates. Live queries send update messages to your program when a file suddenly matches the query or disappears off the face of the earth.

Making a query live is easy: pass a valid BHandler or BLooper to the SetTarget() method before calling Fetch(). Handling updates requires some finesse, though. When an update message is received, it may be while you're still busy reading the results using one of the GetNext methods, so message handling needs to be synchronized with the reading of the results. Also, do be careful that the BMessenger object passed to SetTarget() is not deleted or allowed to go out of scope until you are done with your query. Deleting it any sooner will cause update messages to not be sent any more.

The query update message has the identifier B\_QUERY\_UPDATE. When one is received, you need to then read the 32-bit integer field opcode to find out what other data fields the message contains.

Opcode B\_ENTRY\_CREATED:

Field Name	Type	Description
opcode	int32	Identifier for the message, equaling B_ENTRY_CREATED in this case.
name	string	The name of the new entry
directory	int64	The ino_t number for the directory in which the entry resides.
device	int32	The dev_t number of the device on which the entry exists.
node	int64	The ino_t of the entry itself.

Opcode B\_ENTRY\_DELETED:

Field Name	Type	Description
opcode	int32	Identifier for the message, equaling B_ENTRY_DELETED in this case.
directory	int64	The ino_t number for the directory in which the entry formerly resided.
device	int32	The dev_t number of the device on which the entry used to exist.
node	int64	The ino_t of the removed entry.

At first glance, you might think that live queries are almost always the best choice. There is a drawback to using them: there is some overhead which must be maintained for these messages to actually be useful. All of the information which is obtained from the creation messages – the name, node, and so forth – needs to be stored away because there is no name field sent for B\_ENTRY\_DELETED. This critical piece of information makes it impossible to create an entry\_ref, which happens to be the most common way to store the location of a file or directory outside of strings.

When using a live query, store away the information gleaned from whichever GetNext method you use and do the same for any B\_ENTRY\_CREATED messages. This way, if and when you receive B\_ENTRY\_DELETED messages you will be able to look up the proper item from the information that you are given.

## ***Concluding Thoughts***

Queries are easy to use, fast, and powerful. If you have to look up a collection of files on the system, such as all people files in the case of a contact manager, they provide an easy way of finding and retrieving files. Keep them in mind as you work on projects – you may find a new use for them which blows people's minds.

## ***Going Further***

- Run the `lsindex` command in the Terminal. Most, if not all of them, are pretty self-explanatory. What uses can you think of for some of them?
- If you were going to write a music organizer, how could you use queries and attributes, both indexed and not, to get the most power out of the app as possible?
- How could you go about leveraging queries in these parts of a personal organizer app: contacts, appointments, tasks, and e-mail?

# Programming with Haiku

Lesson 14

Written by DarkWorm



## Monitoring Nodes

Nodes are dumb. Check the *Be Book* – you'll find the exact same thought there, too. They're always lost because they don't know where they are in the filesystem. Truth be told, though, it's because BNode objects are concerned with an entry's data, not metadata like its name and location. They are also incredibly handy because we can ask the operating system to notify us of changes in the filesystem, such as the mounting and unmounting of volumes, the creation of new files, the modification of a file's attributes, or news of other interesting file-related events.

Setting up node monitoring is relatively easy, but first you will need to decide on the kinds of changes for which you wish to receive notifications. There are five types of changes that can be watched.

Mode	Description
B_WATCH_NAME	Watch for changes in file names, including moving or deleting nodes.
B_WATCH_STAT	Watch a file's features manipulated by the <code>stat()</code> function, including creation and modification times, size, permissions, and ownership.
B_WATCH_ATTR	Watch the file's attributes, including adding or removing them.
B_WATCH_DIRECTORY	This only works for directories. It will watch for the creation, deletion, or renaming of entries in the directory. If used on a file, it won't do anything.
B_WATCH_ALL	Watches for everything described above.
B_WATCH_MOUNT	Instead of watching a file or directory, using this flag will cause you to be notified of volumes being mounted or unmounted. The <code>nref</code> argument isn't needed if this is the only flag used in a <code>watch_node()</code> call. <code>B_WATCH_ALL</code> doesn't include this flag.
B_STOP_WATCHING	Turns off watching on the node pointed to by <code>nref</code> .

```
watch_node(const node_ref *nref, uint32 flags, BMessenger messenger);
watch_node(const node_ref *nref, uint32 flags, const BHandler *handler,
           const BLooper *looper = NULL);
```

These calls start the node monitor to watch the node pointed to by `nref` for changes specified in `flags`. Messages are sent by the `BMessenger` messenger or to the `BHandler` or `BLooper` specified. Note that the target of the `BMessenger` must be within the application calling `watch_node()`. The system has only 4096 monitoring slots, so don't go overboard. Also, each call to `watch_node()` consumes a slot even if the node is already watched by the monitor.

```
status_t stop_watching(BMessenger messenger);
status_t stop_watching(const BHandler *handler,
                       const BLooper *looper = NULL);
```

These functions end monitoring for any nodes for which messages are sent to the specified target, i.e. if all node monitoring messages are sent to the same target, this frees all slots in one fell swoop.

## ***The Node Monitor's Update Messages***

As nice as live updates are to have in your application, handling the Node Monitor's update messages can get a bit complicated. The reason for this is a combination of handling the various update messages and mapping them to appropriate actions for your application.

The update messages themselves are strikingly similar to update messages sent by live queries. The `what` field of a node monitor message is `B_NODE_MONITOR`. Checking the opcode field of this message gives you more information about what kind of update you're receiving. The opcode is an `int32`.

### ***Opcode `B_ENTRY_CREATED`***

Conditions for receiving:

- `B_WATCH_DIRECTORY` on directory in which the node was created

<b>Field</b>	<b>Type</b>	<b>Description</b>
<code>name</code>	<code>string</code>	Name of the new entry.
<code>directory</code>	<code>int64</code>	The node number ( <code>ino_t</code> ) for the new entry's parent directory.
<code>device</code>	<code>int32</code>	The id ( <code>dev_t</code> ) of the volume on which the entry was created.
<code>node</code>	<code>int64</code>	The node number of the new entry.

If you watch a directory, this is one possible message you'll receive. Of all of the update messages, this one is the most helpful. The `name`, `device`, and `directory` fields can be used to construct an `entry_ref` which points to the new entry. `device` and `node` can be used to create a `node_ref`, useful if you want to monitor the new entry's node. `device` and `directory` can be used to also create a `node_ref`, but one which points to the entry's parent directory and is suitable for initializing a `BDirectory` object. If you plan on handling `B_ENTRY_REMOVED` opcodes, you'd better make both an `entry_ref` and a `node_ref` for the new entry and stash it away somewhere for later use.

### ***Opcode `B_ENTRY_REMOVED`***

Conditions for receiving:

- `B_WATCH_DIRECTORY` on directory in which the node resided
- `B_WATCH_NAME` on the former node

<b>Field</b>	<b>Type</b>	<b>Description</b>
<code>directory</code>	<code>int64</code>	The node number ( <code>ino_t</code> ) for the former entry's parent directory.

Field	Type	Description
device	int32	The id (dev_t) of the volume from which the entry was removed.
node	int64	The node number of the former entry. Of course, because the node no longer exists, it is invalid and useless except for comparing against cached values.

Notifications for removed entries require extra work to be useful, sadly. The problem lies in the fact that there is no name field sent, preventing construction of an entry\_ref, which just happens to be the most common way of storing an entry's location without consuming a file descriptor. As a result, if you plan on handling removal notifications, you will need to save away both a node\_ref and an entry\_ref for each entry on the filesystem which you wish to track. There will be more on this later.

### *Opcode B\_ENTRY\_MOVED*

Conditions for receiving:

- B\_WATCH\_DIRECTORY on the source or destination directory
- B\_WATCH\_NAME on the node itself

Field	Type	Description
name	string	Name of the moved entry.
from directory	int64	The node number (ino_t) for the entry's source directory.
to directory	int64	The node number (ino_t) for the entry's destination directory.
device	int32	The id (dev_t) of the volume on which the entry resides.
node	int64	The node number of the moved entry. This doesn't change even though the entry has moved.

Handling B\_ENTRY\_MOVED is almost the same as B\_ENTRY\_CREATED except for the changes in directory field names.

### *Opcode B\_STAT\_CHANGED*

Conditions for receiving:

- B\_WATCH\_STAT on the node itself

Field	Type	Description
device	int32	The id (dev_t) of the volume on which the entry resides.
node	int64	The node number of the entry.

### *Opcode B\_ATTR\_CHANGED*

Conditions for receiving:

- B\_WATCH\_ATTR on the node itself

Field	Type	Description
device	int32	The id (dev_t) of the volume on which the entry resides.
node	int64	The node number of the entry.

Changes in stat data are as big of a pain to handle as entry removals and for the same reason: having to stash away node\_refs. The Be Book even recommends storing away a copy of each entry's stat data, too, which wouldn't be a bad idea as long as you're careful about memory usage. Handling attribute changes are pretty much the same thing.

### *Opcode B\_DEVICE\_MOUNTED*

Conditions for receiving:

- B\_WATCH\_MOUNT flag used with watch\_node().

Field	Type	Description
device	int32	The id (dev_t) of the volume on which the new volume's mount point resides
new device	int32	The id (dev_t) of the new volume.
node	int64	The node number of the entry.

### *Opcode B\_DEVICE\_UNMOUNTED*

Conditions for receiving:

- B\_WATCH\_MOUNT flag used with watch\_node().

device	int32	The id (dev_t) of the former volume.
--------	-------	--------------------------------------

Volume-related notifications don't require very much effort, thankfully. Do be aware that dev\_t numbers are apparently recycled fairly often, but the only time you'll ever need it is if you've been tracking volumes and keeping a copy of each volume's dev\_t identifier stored away somewhere.

## **Handling Update Messages**

At this point you may be wondering how all of these messages fit together or how to use them in your own programs. The work itself isn't very difficult, but it can be tedious. The way node monitoring fits into your program largely depends on what kind of program you're writing and the purpose for which you want notifications. If you are working on a file manager of some kind, you'll probably be monitoring everything, including volume-related events. This is definitely the more complicated scenario. More likely, though, you'll just be monitoring the node for a document your program is editing, in which case your job is

relatively easy. We'll be looking at the simpler of the two cases since the file manager scenario is very implementation-specific and is just an extension of the other.

Let's say for this instance that we are writing a simple text editor. Architecturally, we'll say that each actual document is encapsulated by a Document class and BView subclass has been created for the editor, predictably called DocumentEditor. What we wish to accomplish with node monitoring is to be informed of outside changes to the document currently being edited.

The changes we will need to handle in this case are removal, moving, and stat changes. It's possible to merely ignore removals, but we want to ensure that the user's data is kept safe – if the user didn't have any changes made and closed the window, the contents of the document stored on the disk and in memory would be lost, potentially leading to frustration and gnashing of teeth. It would be better to ask if the user wishes to re-save the document.

### *Setup Before Watching Files*

The removal and stat notifications received from the node monitor only have enough information for creating node\_ref structures, so our document class will need to store a stat structure and a node\_ref. The Document class could look something like this:

```
#include <Node.h>
#include <sys/stat.h>

class Document
{
public:
    Document(const char *path);
    ~Document(void);

    const char *GetName(void) const;

    status_t    Load(const char *path);
    void        NodeMoved(const entry_ref &ref);

    node_ref    GetNodeRef(void) const;
    entry_ref   GetRef(void) const;
    struct stat GetStatData(void) const;

    // various methods here

private:
    entry_ref   fRef;
    node_ref    fNodeRef;
    struct stat fStatData;

    // More document-related properties here
};
```

Keeping an entry\_ref around is a cheap way of storing the location of the document without using a file descriptor. fNodeRef and fStatData are kept around for node monitoring purposes. The Load() method will need to set these properties if everything else about the document has been successful. NodeMoved() is necessary because the owning DocumentEditor instance will receive the notifications from the Node Monitor and will need to pass on information so that in the event that the user – or something else, for that matter – moves the file while it is open, the editor doesn't try to save the document in the old location.

The Load() method could look something like this:

```
Document::Load(const char *path)
{
    BFile file(path, B_READ_ONLY);
    if (file.InitCheck() != B_OK)
        return file.InitCheck();

    // A bunch of data loading muck goes here. Nothing to see here.
    // Move along, now. ;-)

    // Assuming that everything went well in loading the document,
    // let's save the info about the file on the disk.
    BEntry entry(path);
    entry.GetRef(&fRef);
    entry.GetNodeRef(&fNodeRef);
    entry.GetStat(&fStatData);
}
```

There! Now that the initial setup has been taken care of, we can move on to the DocumentEditor class. We'll just say that it has a Load() method, too.

```
#include <NodeMonitor.h>

status_t
DocumentEditor::Load(const char *path)
{
    status_t status = fDocument.Load(path);
    if (status != B_OK)
        return status;

    // Document loaded OK, so set up node monitoring. BView inherits
    // from BHandler, so this is an easy call.
    node_ref nref = fDocument.GetNodeRef();
    watch_node(&nref, B_WATCH_NAME | B_WATCH_STAT, this);
}
```

### *Fielding Update Messages*

With node monitoring now set up, all that is left is to tweak MessageReceived() to handle the update messages.

```
DocumentEditor::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        // A bunch of other message-handling cases here

        case B_NODE_MONITOR:
        {
            // We'll use a separate function to prevent making
            // MessageReceived() any messier than it already tends
            // to be.
            HandleNodeMonitoring(msg);
            break;
        }
        default:

```

```

        {
            BView::MessageReceived(msg);
            break;
        }
    }
}

void
DocumentEditor::HandleNodeMonitoring(BMessage *msg)
{
    int32 opcode;
    if (msg->FindInt32("opcode", &opcode) != B_OK)
        return;

    switch (opcode)
    {
    case B_ENTRY_REMOVED:
    {
        // Document has been lost on disk. Alert user.
        BString errmsg;
        errmsg += fDocument.GetName();
        errmsg << " has been deleted on the disk. Do you wish "
            << "to re-save it to prevent data loss?";
        BAlert *alert = new BAlert("MyCoolEditor",
            errmsg.String(),
            "No", "Yes");

        if (alert->Go() == 1)
            fDocument.Save();
        break;
    }
    case B_ENTRY_MOVED:
    {
        dev_t device;
        ino_t destDir;
        BString name;
        if (msg->FindInt64("to directory", &destDir) == B_OK &&
            msg->FindInt32("device", &device) == B_OK) &&
            msg->FindString("name", &name) == B_OK)
        {
            entry_ref newRef;
            newRef.setname(name.String());
            newRef.device = device;
            newRef.dir = destDir;
            fDocument.NodeMoved(ref);
        }
        break;
    }
    case B_STAT_CHANGED:
    {
        // Check to see what has changed. We'll ignore (for now)
        // changes in permissions, but a change in
        // modification time means the file has changed, in which
        // case we'll ask about reloading from disk.
        struct stat oldStat = fDocument.GetStatData();

        struct stat newStat;
        entry_ref ref = fDocument.GetRef();
        BFile file(&ref);
    }
    }
}

```

```

file.GetStat(&newStat);
if (newStat.st_mtime != oldStat.st_mtime)
{
    BString errmsg;
    errmsg += fDocument.GetName();
    errmsg << " has been changed on the disk. Do you "
        << "wish to reload it?";
    BAlert *alert = new BAlert("MyCoolEditor",
        errmsg.String(),
        "No", "Yes");

    if (alert->Go() == 1)
    {
        BMessage refMsg(B_REFS_RECEIVED);
        refMsg.AddRef("refs", fDocument.GetRef());
        Window()->PostMessage(refMsg);
    }
}

break;
}
default:
break;
}
}
}

```

In case you're wondering, `Document::NodeMoved()` just uses the same `BEntry` code as in `Load()` to update the document's `node_ref` and `stat` data.

## Going Further

- Give some thought to how node monitoring might be used in a file manager application.
- Look over the code from the open source file manager `Seeker` to see how node monitoring was used to track entries in the currently-open folder.

# Programming with Haiku

Lesson 15

Written by DarkWorm



## ***Our Own File Types***

While there are a great many different kinds of files already available which cover everything from pictures to video to e-mail, sometimes it is necessary for our programs to use their own specialized types of files. While it's possible to just create a file with some non-standard extension, actually creating a useful, full-blown file type in Haiku involves a little more work. None of it is difficult, but it does require a piecing together some odds and ends from different parts of the Storage Kit to make it all work well.

The process for making our own file type has a few basic steps: determine the MIME type and extension(s), design an icon for the type, and installing it into the MIME database. Once it's installed, there isn't much needed to use your new file type, either.

## ***Creating the New Type***

First, we need to think up a little information before we can create our new file type. Ask yourself three questions:

1. What is the extension(s) for my new type?
2. What is the MIME type for my new type?
3. What will the icon look like?

For our purposes here, we'll say we're creating a new kind of text document for a word processor called MyWrite. It would make sense to use an extension that is obviously tied to our program, so we'll use a 4-letter extension: mywr.

MIME types aren't complicated, but they do require a little thought. The supertype for the document must be determined. MyWrite files are text documents, so they have the `text` supertype. Choices for other files include `audio`, `video`, `image`, and `application`. The application type is generally used for formats which don't fit into another supertype. Compressed archives, such as 7-Zip and BZip2 fit into this category. Once the new type's supertype is set, the rest of the type is constructed. Non-standard MIME types start with "x-". Vendor-specific types start with "vnd.". For MyWrite, the full MIME type will be `text/x-mywrite`.

For some developers, the icon is the hardest part because they write code, not draw pretty pictures. A type's icon doesn't have to be fancy, although it does reflect better on your program if it looks reasonably professional. Haiku's icon tool of choice is Icon-O-Matic, which happens to be bundled in with the rest of the operating system. For the sake of focus, we won't go into the details of creating an icon with Icon-O-Matic – that will be another day's lesson. We'll just continue on and assume that you have an icon that will work.

To install our new MIME type we will make use of the `BMimeType` class. It is the official interface with Haiku's MIME database. While we could use the FileTypes preferences application, we need to be able to create the type programmatically. Here's some code that will explain how to install a basic type.

```
#include <Mime.h>           // The header for BMimeType

// A header which holds the icon data converted into C++ code -- more on
// this afterward.
```

```

#include "IconDefs.h"

void
InstallMyWriteType(void)
{
    BMimeType mime;

    mime.SetType("text/x-MyWrite");
    mime.SetShortDescription("MyWrite text document");
    mime.SetLongDescription("MyWrite text document");

    // Set the icon for the type using the data stored in a constant
    // data structure defined in our IconDefs.h. This function call
    // is not documented in the Be Book because it was introduced into
    // the API in Haiku.
    mime.SetIcon(kVectorIconBits, sizeof(kVectorIconBits));

    // Set the extension for our type
    BMessage extMsg;
    extMsg.AddString("extensions", "mywr");
    mime.SetFileExtensions(&extMsg);

    // This sets the signature of the preferred application for
    // opening our file type.
    mime.SetPreferredApp("application/x-vnd.test.MyWrite");

    // If we were doing this on BeOS R5 or Zeta, a call to
    // BMimeType::Install() would be necessary to make all of the
    // changes at once. Haiku does not have this requirement and updates
    // the type as each method is called.
}

```

Nothing in the above code should look terribly surprising, especially if you have ever used the FileTypes preferences application. Each of the methods sets basic information about the new type. There is another optional method call which is quite handy: `SetSnifferRule()`. Sniffer rules have nothing to do with airplane glue; instead, they are all about automatic identification of files of your new type. The only problem with MIME sniffer rules is their syntax – it's incredibly flexible, making simple rules merely a little weird-looking and the more complicated ones enough to give you a headache. No matter. The convenience they offer is almost always worth the trouble, which isn't much in the vast majority of cases.

## ***Automagical File Identification: MIME Sniffing***

A MIME sniffer rule always follows the following format:

```
rating [begin:end] ( [begin:end] 'pattern1' | ... )
```

At face value, this strange-looking text amounts to a priority rating, a byte range, and a set of patterns placed inside a pair of parentheses. This is only partially true, however. At the lowest level in a rule are what we'll call a **pattern pair**. This consists of a text-matching pattern and an optional byte range for the pattern. The pattern itself must be placed inside a pair of single quotes. One or more pattern pairs can be placed inside a pair of parentheses, separated by a single pipe character (`|`). The example below shows three pattern pairs.

```
([0:15] 'Sample' | 'Another Sample' | [10:] 'Third example')
```

The first pair searches for the word *Sample* in the first 15 characters of the file. The second pair matches only if *Another Sample* shows up at the beginning of the file. The third pair starts searching for *Third example* starting at the tenth byte of the file and continuing to the file's end.

The next level up from a pattern pair in a MIME sniffer rule is the **pattern set**. A pattern set is an optional byte range paired with one or more pattern pairs inside a set of parentheses. The byte range used in a pattern set should be considered a fallback range – it is only used for a pattern pair if the pair does not specify a byte range.

At the top level, a MIME sniffer rule is just a priority rating and one or more pattern sets. The priority is simply a floating point number from 0.0 to 1.0 which rates it in comparison to other rules for the MIME type. Each pattern set is considered to be logically chained together with a boolean AND operator – all pattern sets must match for the rule to match. Below is an example of how this works.

```
.5 ('foo') ([20:] 'BAR') ([0:10] 'baz')
```

This rule has a priority of 0.5 and requires *foo* to be at the beginning of the file, *BAR* to be somewhere in the file after the 19<sup>th</sup> byte, and *baz* to be somewhere in the first ten bytes.

Beyond all of the confusion that this may cause, there are a few other options in constructing patterns which come in handy. First, *-i* can be placed in front of the first pattern pair in a pattern set to make all of the patterns in the set case-insensitive. Second, a mask can be applied to any pattern to specify certain bytes which do or do not matter by using an ampersand (&) in a pattern pair. Third, octal and hexadecimal values can be specified using *\* or *\x* as an escape prefix. Fourth, floating point values can be specified using scientific notation. These options can be observed in the examples below, taken from the private Haiku header file `Parser.h`.

```
200e-3 (-i 'ab')  
0.70 ("8BPS \000\000\000\000" & 0xffffffff0000ffffffff )
```

The first example has a priority of 0.2 and looks for *ab* at the beginning of the file, ignoring case. The second is a little more complicated: in the first ten bytes of the file, the first four bytes are expected to be '8BPS', bytes five and six are ignored, and the next four are expected to be zero. The ignored bytes are specified by the zero bytes in the mask. This second rule is used by Haiku to identify Adobe Photoshop files, which always begin with a series of bytes which look like this:

```
38 42 50 53 00 01 00 00 00 00 00 00
```

The first four of these bytes is the string 8BPS, the next two are a two-byte integer with a value of one, and the last six are reserved bytes which are always supposed to be zero.

## ***Type-Specific Attributes***

Some file types have specific attributes associated with them. For example, Ogg Vorbis and MP3 files use the custom attributes `Audio:album`, `Audio:artist`, and `Audio:title` to store information traditionally kept in tags. Should your new type also have a use for extra attributes, it's a pretty simple operation to add them. You will store information about them in

a BMessage and pass the message to BMimeType::SetAttrInfo(). Let's say that we wanted to tweak the Person file type to have separate attributes for a person's first and last names. Here is the code that you would write to set these two custom attributes.

```
#include <Message.h>
#include <Mime.h>

int
main(void)
{
    BMimeType mime("application/x-person");
    BMessage attrMsg;

    // We get the existing information for Person files because otherwise
    // we will replace it with the two measly attributes that we have
    // below. We want to add to the existing attributes for Person files,
    // not replace them.
    mime.GetAttrInfo(&attrMsg);

    // Each of these fields needs to be added to the message for the
    // custom attribute to be useful.
    attrMsg.AddString("attr:public_name", "First Name");
    attrMsg.AddString("attr:name", "META:firstname");
    attrMsg.AddInt32("attr:type", B_STRING_TYPE);
    attrMsg.AddBool("attr:viewable", true);
    attrMsg.AddBool("attr:editable", true);

    // These three fields are not documented in the Be Book, but are used
    // by Tracker to determine how the information is displayed in a
    // Tracker window.
    attrMsg.AddInt32("attr:width", 120);
    attrMsg.AddInt32("attr:alignment", B_ALIGN_LEFT);
    attrMsg.AddBool("attr:extra", false);

    attrMsg.AddString("attr:public_name", "Last Name");
    attrMsg.AddString("attr:name", "META:lastname");
    attrMsg.AddInt32("attr:type", B_STRING_TYPE);
    attrMsg.AddBool("attr:viewable", true);
    attrMsg.AddBool("attr:editable", true);
    attrMsg.AddInt32("attr:width", 120);
    attrMsg.AddInt32("attr:alignment", B_ALIGN_LEFT);
    attrMsg.AddBool("attr:extra", false);

    mime.SetAttrInfo(&attrMsg);
}
```

## ***Final Thoughts***

Our new type is ready to use now! To set a file to our new type, we have a couple of options:

1. BNode::WriteAttr()
2. BNode::WriteAttrString()
3. BNodeInfo::SetType()

All three methods work properly on BeOS R5 and Zeta, but as of this writing only the third way works properly under Haiku due to an unresolved bug in the operating system. Your new

file type will show up in the FileTypes preferences application and will look and act just like any other standard system file type.

# Programming with Haiku

Unit 2 Review

Written by DarkWorm



## **Lesson 6**

1. What class is responsible for creating the connection with the app\_server?

## **Lesson 7**

1. What is the purpose of the Support Kit?
2. What is the name of the method responsible for handling messages sent to a BWindow?
3. What does the BButton method ResizeToPreferred() do? Why should we use it?

## **Lesson 8**

1. What class do all GUI controls ultimately inherit from?
2. Why is the B\_WILL\_DRAW flag important for BView objects?
3. What is the difference between the methods Bounds() and Frame()?

## **Lesson 9**

1. What is the difference between BListView's SetSelectionMode() and SetInvocationMessage()?
2. What is the purpose of SetTarget()?

## **Lesson 10**

1. What is the difference between a BFile and a BEntry?
2. What is an entry\_ref?
3. How is an entry\_ref different from BEntry?

## **Lesson 11**

1. Can a BFile object perform operations on attributes? Why or why not?
2. What must be done in order to read a BFile's data into a BString object?

## **Lesson 12**

1. What is metadata?
2. What is the size limit of an indexed attribute?
3. What is the airspeed velocity of an unladen swallow?
4. Which of these types may NOT be used in an indexed attribute?
  - a) float
  - b) bool
  - c) string
  - d) int64
  - e) char
5. Which class should be used to set the preferred application for a file type? How about for an individual file?

## **Lesson 13**

1. How can you find out if a particular attribute can be queried?
2. What Terminal command can make it possible for an attribute to be queried?
3. What is the side effect of having a large number of attributes which can be queried?

4. What is Reverse Polish Notation?

**Bonus:** Which functions can be used to find out the names of indexed attributes on a volume?

### **Lesson 14**

1. Which of these is NOT a node-monitoring flag?
  1. B\_WATCH\_NAME
  2. B\_WATCH\_STAT
  3. B\_WATCH\_ATTR
  4. B\_WATCH\_FILE
2. Why do both `entry_ref` and `node_ref` objects need to be stored away to handle `B_ENTRY_REMOVED` operations?

### **Lesson 15**

1. Write a MIME sniffer rule which looks for the pattern 'FOOBAR' followed by two zero bytes in the first thirty-two characters of a file.

# Programming with Haiku

Lesson 16

Written by DarkWorm



Fonts are one of those things which some people go ga-ga over – maintaining a collection of thousands for no apparent reason except for liking them – and others couldn't possibly care less about. Regardless of the camp to which you belong, knowing even a little bit about how to manipulate text display in Haiku is quite handy.

## ***Using Fonts in Haiku***

Like many other kinds of add-ons in Haiku, there is a folder dedicated to holding fonts the user wishes to add to the system. The `find_directory()` constant for it is `B_COMMON_FONTS_DIRECTORY`, which maps to `/boot/common/data/fonts` in Haiku and `/boot/home/config/fonts` for Zeta and BeOS R5. R5 and Zeta required fonts to be sorted into subfolders, *ttfonts* for TrueType fonts and *psfonts* for PostScript. Haiku removes this restriction. Haiku also removes the requirement to reboot the system to use newly-installed fonts – copy them in and away you go.

Since the days of BeOS R5, font handling has changed considerably. Text rendering quality was a sore spot for some BeOS users. R5 did not provide very good antialiasing for text. The situation improved considerably in Zeta with the change to a different font engine – one developed by the renowned font company Bitstream. However, this improvement bit the dust along with the rest of Zeta. Haiku, fortunately, uses the open source FreeType library to provide excellent font rendering and make more kinds of fonts usable than ever before.

## ***A Bit About Typography***

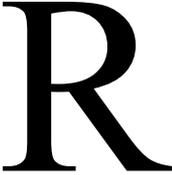
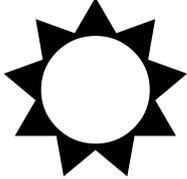
No lesson about fonts would be complete without a little background about typography, which is the process of arranging type for printing. The basics of typography were started with Gutenberg's printing press, and it has steadily grown more complicated ever since. You won't need to have an exhaustive knowledge of the subject to use text in your applications, but knowing some of terms is quite beneficial.

Fonts are grouped together into families. A font family is little more than a name. For example, Times New Roman, Century Schoolbook L, and Helvetica are all examples of well-known font families. Each family contains one or more individual styles or **faces**. While you will easily recognize Italic and Bold as being familiar, there are others, as well. Sometimes you will hear the term **font weight** when styles are discussed. A font's weight is the heaviness of the type. Here is a non-exhaustive list of font weights, from the lightest to the heaviest.

- Extra light
- Light
- Book
- Normal / Regular / Plain
- Medium
- Demi-bold
- Bold
- Black
- Extra Black

Font families are often grouped into generalized categories. The specific categories used depend on whom you ask, but for our purposes we will use three basic categories: serif, sans serif, and decorative. Serif fonts are the most common in reading literature. A **serif** is a

decorative stroke appearing in certain locations of letters. Characters in sans serif fonts do not have them. Decorative fonts are most often used for logos or specific effects. Here are examples of the capital letter R as rendered in three different fonts.

Times New Roman	Arial	Wingdings
		

Times New Roman is a serif font, Arial is sans serif, and Wingdings is decorative. Notice the little lines extending out to the sides at the ends of the legs of the R shown in Times New Roman. These are serifs.

Along with these terms, here are some others that you will see:

**Leading** – Pronounced to rhyme with the word 'bedding', this is the amount of space placed between lines of text.

**Glyph** – A character used in a font. Depending on the font used this can be a letter, a number, or something else entirely.

**Kerning** – The amount of horizontal space placed between individual letters. Proportional fonts – those without a fixed character width – use different amounts of space between letters for better readability.

**Baseline** – An imaginary line upon which letters are placed.

**Ascender** – Part of a lowercase character which sticks up above the rest, such as the top part of a lowercase letter d.

**Descender** – Part of a lowercase character which hangs down below the baseline, such as the bottom part of a lowercase letter p.

**Point** – Points are a unit of measurement which is not to be confused with pixels. There are 72 points in an inch.

## ***Working with Fonts***

Fonts are a potentially complex subject, but what do you do if all you want to do is print some text on the screen? Not a problem. Let's start with the simplest of examples: a BView that shows some text.

1. Create a new project in Paladin with the *Main Window with GUI and Menu* template.
2. Open App.cpp and set the application signature to "application/x-vnd.test-FontDemo1" and save it.
3. Create a new file called MainView.cpp and check the box to also create a header.  
Now let's get to the real work.

### *MainWindow.cpp*

```
#include "MainWindow.h"

#include <Application.h>
#include <Menu.h>
#include <MenuItem.h>
#include <View.h>

#include "MainView.h"

MainWindow::MainWindow(void)
    : BWindow(BRect(100,100,500,400), "Font Demo", B_TITLED_WINDOW,
              B_ASYNCHRONOUS_CONTROLS)
{
    BRect r(Bounds());
    r.bottom = 20;
    fMenuBar = new BMenuBar(r, "menubar");
    AddChild(fMenuBar);

    r = Bounds();
    r.top = 20;
    MainView *view = new MainView(r);
    AddChild(view);
}

void
MainWindow::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        default:
        {
            BWindow::MessageReceived(msg);
            break;
        }
    }
}

bool
MainWindow::QuitRequested(void)
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return true;
}
```

### *MainView.h*

```
#ifndef MAINVIEW_H
#define MAINVIEW_H

#include <View.h>

class MainView : public BView
{
public:
    MainView(const BRect &frame);
    void      Draw(BRect update);
};

#endif
```

### *MainView.cpp*

```
#include "MainView.h"

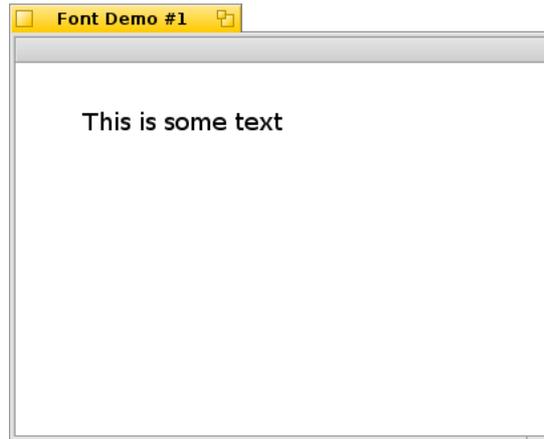
MainView::MainView(const BRect &frame)
    : BView(frame, "textview", B_FOLLOW_ALL, B_WILL_DRAW)
{
}

void
MainView::Draw(BRect update)
{
    // The Draw hook function is called whenever a BView is asked to
    // redraw itself on the screen. We will just write some code to draw
    // some text.
    BFont font;
    font.SetSize(18.0);

    SetFont(&font);

    // DrawString uses the BView's current font settings, draw mode, and
    // color to draw the text. Note that the point specified is the left
    // end of the baseline, so calculating where to draw text can be a
    // little backwards when compared to drawing anything else.
    DrawString("This is some text", BPoint(50,50));
}
```

This example is about as simple as it gets. Running the demo results in this window:



BView's `Draw()` method is not limited to merely drawing text, but we'll save that for another lesson. It is to be noted, though, that the graphics state of the `MainView` class – i.e. the pen location, font size, current high color, etc. – only changes when we change it. With that said, we could just as easily move everything except the `DrawString()` call into the constructor to make drawing faster, but this kind of optimization will only work for simpler controls.

## ***What Do We Use?***

If you take a quick peek at `Font.h`, you'll see that there are a ton of different methods for the `BFont` class. The vast majority are for specialized use, such as selecting different encodings or rotating the text. Here are the methods that you will use most:

```
void GetHeight(font_height *height) const;
```

Gets the pixel value for the font's leading, maximum ascender height, and maximum descender size, taking the current font size into account. `font_height` is just a struct containing three floats: `ascent`, `descent`, and `leading`. If you want to figure out the safe pixel height for a line of text printed with the font, call this and add the three values together.

```
void TruncateString(BString *string, uint32 mode, float maxWidth);
```

This method yanks characters from the string until it fits in `maxWidth` pixels. There are four modes: `B_TRUNCATE_BEGINNING`, `B_TRUNCATE_MIDDLE`, `B_TRUNCATE_END`, and `B_TRUNCATE_SMART`. The first three modes are obvious. The last one, according to the *Be Book*, is apparently meant to be used with the related method `GetTruncatedStrings()` to cut the strings in such a way that they are all different, paying attention to word boundaries, separators, punctuation, etc. As of R5, it was unimplemented. The Haiku implementation functions exactly the same as `B_TRUNCATE_MIDDLE`.

```
bool IsFixed(void) const;
```

Returns true if the font is a fixed-width font.

```
void SetFace(uint16 face);
uint16 Face(void) const;
```

Sets or gets the face of the font. Instead of string values, these calls rely on integer constants, which is handy if you want to specify a certain style without having to figure out its name. Here are the face constants:

- B\_ITALIC\_FACE
- B\_UNDERSCORE\_FACE
- B\_NEGATIVE\_FACE
- B\_OUTLINED\_FACE
- B\_STRIKEOUT\_FACE
- B\_BOLD\_FACE
- B\_REGULAR\_FACE
- B\_CONDENSED\_FACE\*
- B\_LIGHT\_FACE\*
- B\_HEAVY\_FACE\*

\*The CONDENSED, LIGHT, and HEAVY face constants are new in Haiku and not available to Zeta or BeOS R5 programs.

```
void SetFamilyAndFace(const font_family family, uint16 face);
void SetFamilyAndStyle(const font_family family, const font_style style);
void GetFamilyAndStyle(font_family *family, font_style *style);
```

These methods make for an easy way to get or set a specific style.

```
void SetSize(float size);
float Size(void) const;
```

Set the point size for the font. As of this writing, the Be Book states a limit of 10,000 points, but Haiku does not have this limitation, unlike BeOS.

```
float StringWidth(const char *string) const;
float StringWidth(const char *string, int32 length) const;
```

Returns the width of string in the current size.

```
int32 count_font_families(void);
int32 count_font_styles(font_family family);
status_t get_font_family(int32 index, font_family *family,
                        uint32 *flags = NULL);
status_t get_font_style(font_family family, int32 index, font_style *style,
                       uint32 *flags = NULL);
```

These global functions are used to iteratively read all fonts installed on the system.

## ***Other Tips and Tricks***

- There are three global font objects: `be_plain_font`, `be_bold_font`, and `be_fixed_font`. These built-ins are set by the user with the Fonts preferences application. Use these fonts in your applications whenever possible to help keep a consistent look to the interface.
- If you draw text on a colored background, make sure you set the view's low color to that of the background or else your text will look kinda weird.
- To draw text on top of a picture, set the view's drawing mode to `B_OP_ALPHA` before drawing the text.
- Tweaks can be made to the kerning with the methods related to escapements.

## ***Going Further***

- Play around with the `FontDemo` demonstration app bundled with Haiku. See how each of the different effects changes the text.
- Try using a `BMessageRunner` to make an animated text demo that changes sizes, rotates the text, or something else cool.

## ***Answers to Unit 2 Review***

### ***Lesson 6***

1. The `BApplication` class sets up communications with the `app_server`.

### ***Lesson 7***

1. The Support Kit is a collection of classes used to support the other kits with general-purpose classes for string handling, memory management, and more.
2. `MessageReceived()` handles messages sent to a `BWindow`, as well as any `BHandler` subclass, such as `BApplication`, `BLooper`, and `BView`.
3. `ResizeToPreferred()` causes a `BButton` to resize itself to the smallest size which properly displays the label. We should use it because it makes our code work regardless of the size of the font used in the GUI.

### ***Lesson 8***

1. All GUI controls ultimately inherit from the `BView` class.
2. `B_WILL_DRAW` tells the `app_server` that the `BView` in question needs to be sent redraw messages.
3. The difference between `Bounds()` and `Frame()` is that `Bounds()` returns the size of the `BWindow` or `BView` and `Frame()` returns its size and location in its parent's coordinates.

### ***Lesson 9***

1. `SetSelectionMode()` sets the message sent whenever a `BListView`'s selection changes. The invocation message, set by `SetInvocationMessage()`, sets the message sent when the user double-clicks on an item in a `BListView`.
2. `SetTarget()` chooses a `BLooper` or `BHandler` which is to become the target of messages sent by a control.

## Lesson 10

1. The main difference between a BFile and a BEntry is that a BEntry handles a file or directory's presence in the filesystem, such as its location and other related data, whereas a BFile manipulates a file's data, such the information stored in the file and its attributes.
2. An entry\_ref is a lightweight structure which can describe the location of a file or directory on disk.
3. entry\_ref instances may not point to a location which does not exist, and they do not consume a file handle, unlike BEntry objects.

## Lesson 11

1. Yes. It is a subclass of BNode, which has methods to manipulate attributes.
2. A BString object's Lock() method must be called to get a pointer which can be passed to BFile::Read(). Care also must be taken to ensure that data read from the BFile is not greater than the size specified in BString::Lock().

## Lesson 12

1. Metadata is information about a file which is not part of the file's data, such as its modification time.
2. An indexed attribute may be no larger than 255 bytes.
3. African or European?
4. A bool may not be used in an indexed attribute. Data of type char can be because the char type is simply reinterpreted integer data.
5. BMimeType should be used to set the preferred application for an entire file type, whereas BNodeInfo should be used for individual files.

## Lesson 13

1. lsindex will tell you the names of all indexed attributes on a volume.
2. mkindex can make an attribute able to be queried.
3. Having a large number of indexed attributes on a volume will slow down all queries on that volume.
4. Reverse Polish Notation is an entry system where the operator is entered after each operand instead of between them.

**Bonus:** The functions fs\_open\_index\_dir(), fs\_read\_index\_dir(), and fs\_close\_index\_dir() are used to programmatically get the names of indexed attributes on a volume.

## Lesson 14

1. B\_WATCH\_FILE is not a node-monitoring flag.
2. entry\_ref and node\_ref objects need to be stored away to handle B\_ENTRY\_REMOVED operations because the node-monitoring message sent does not provide the removed entry's name, making it impossible to create an entry\_ref strictly based on the information provided in the message. It does, however, provide enough to construct a node\_ref.

## Lesson 15

1. 0.5 [0:32] ( "FOOBAR\000\000" )

# Programming with Haiku

## Lesson 17

Written by DarkWorm



The Interface Kit is all about creating and using controls for the graphical interface. In some of the earlier lessons, we touched on using some of the existing controls in the kit, but to fully understand how to use them, we will create our own custom control. In doing so, we will learn about the different parts of the Interface Kit and how to use them effectively.

## ***The Haiku Way of Controls***

If you have been involved in programming for a while, you will probably know something about good code design. If not, you can learn a little from the Haiku API. The developers at Be Inc. were careful to put together the API in a way that it's pretty easy to use. It also uses a few paradigms which would be wise to follow in the appropriate places.

One concept of control design used in the Interface Kit is using lightweight items for lists. The `BListView` and `BOutlineListView` classes exemplify this. Each can have potentially hundreds of items, so each must take up as little space as possible. `BStringItem` objects, instead of each one carrying the sizable overhead of being a `BView` individually, depend on their owning list for drawing themselves. They only concern themselves with a little data, such as in the case of `BStringItem`.

Another rule of thumb used in the Interface Kit is utilizing existing classes to implement your control whenever possible. `BTextControl` is just a single-line special case of `BTextView`, for example. By using standard controls in your own instead of implementing new ones, you can help keep the interface consistent for the user.

The API used to interact with your control should also follow conventions used by other classes in the Interface Kit. Provide hook functions for events. Better yet, if your control has a value and a label, make your control a subclass of `BControl`. Note that not all controls in the Interface Kit do so, such as `BListView`. If you don't subclass `BControl`, use the `BInvoker` class as a mix-in to easily change message targets and otherwise provide a familiar interface to messaging for your control. Make most data returning functions `const` functions – ones which tell the compiler that they don't change the state of the object. Lastly, don't write a new control unless one available in the kit isn't appropriate to the task at hand. Often one or two controls used together can perform the same task a new control coded specifically for the task. Work smarter, not harder.

## ***Our New Control: ColorWell***

One kind of control which the Haiku API lacks is a color display control. In addition to images and text, Haiku also allows colors to be dragged and dropped from one location to another. We will be creating a well which holds and displays a color value and, later on, supports dragging colors to other locations and receiving such dragged colors. Note that we will not be creating a color picker to modify the color – that task is easily handled by the `BColorControl` class.

There are a few things that we should keep in mind for our control. First, colors can be represented by 32-bit integers and as `rgb_color` objects, so we will want to support both ways of assigning and returning color values. Second, we should have both a round style and a square style to fit with the style of the program in which it might be used. Third, it should be possible to disable our control, so we should also think of a look for the control when it is disabled which is easy to discern as such.

Here's what our header should look like:

## ColorWell1.h

```
#ifndef COLOR_WELL_H
#define COLOR_WELL_H

#include <Control.h>

enum
{
    COLORWELL_SQUARE_WELL,
    COLORWELL_ROUND_WELL,
};

class ColorWell : public BControl
{
public:
    ColorWell(BRect frame, const char *name,
             BMessage *msg,
             int32 resize = B_FOLLOW_LEFT |
                          B_FOLLOW_TOP,
             int32 flags = B_WILL_DRAW,
             int32 style = COLORWELL_SQUARE_WELL);
    ~ColorWell(void);

    virtual void SetValue(int32 value);
    virtual void SetValue(const rgb_color &color);
    virtual void SetValue(const uint8 &r, const uint8 &g,
                          const uint8 &b);
    rgb_color ValueAsColor(void) const;

    virtual int32 SetStyle(const int32 &style);
    Style(void) const;

    virtual void Draw(BRect update);

private:
    void DrawRound(void);
    void DrawSquare(void);

    rgb_color fDisabledColor,
              fColor;

    int32 fStyle;
};

#endif
```

Having given this a look, think for a moment what might go into each of these methods.

Before we go diving into code, let's take a quick side trip to learn a bit about how BViews handle drawing. There are four steps to how a BView draws itself: invalidation, drawing, child drawing, and post-child drawing. Invalidation just means that the BView tells the app\_server that a certain section of its area needs to be redrawn. Invalid regions are caused by all sorts of things, such as if a window hid part of it and now it doesn't or perhaps an internal value has changed and the BView needs to update a label. Later on, the BView will actually draw itself, handled by the Draw() hook function. Once a view has drawn itself, it typically will make sure that any child views also are redrawn as needed. It is also possible for a BView to do some drawing after all of its child BViews have finished updating themselves.

Any drawing of this kind is done with the `DrawAfterChildren()` hook function. It is not often used, however.

## ***Coding the ColorWell***

### *ColorWell1.cpp*

```
#include "ColorWell1.h"
```

```
ColorWell::ColorWell(BRect frame, const char *name, BMessage *message,
    int32 resize, int32 flags, int32 style)
    : BControl(frame,name,NULL,message, resize, flags)
{
    SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
    SetLowColor(0,0,0);

    fColor.red = 0;
    fColor.green = 0;
    fColor.blue = 0;
    fColor.alpha = 255;

    fDisabledColor.red = 190;
    fDisabledColor.green = 190;
    fDisabledColor.blue = 190;
    fDisabledColor.alpha = 255;
}
```

```
ColorWell::~ColorWell(void)
{
}
```

```
void
```

```
ColorWell::SetValue(int32 value)
```

```
{
    // It might seem strange to support setting the color from an
    // integer, but this behavior is expected for BControl-derived
    // controls, so we will do our best to support it.
    BControl::SetValue(value);

    // int32's can be used to pass colors around, being that they contain
    // 4 8-bit integers. Converting them to RGB format requires a little
    // bit shift division.
    fColor.red = (value & 0xFF000000) >> 24;
    fColor.green = (value & 0x00FF0000) >> 16;
    fColor.blue = (value & 0x0000FF00) >> 8;
    fColor.alpha = 255;

    SetHighColor(fColor);
    Draw(Bounds());
}
```

```
void
```

```
ColorWell::SetValue(const rgb_color &col)
```

```
{
    fColor = col;
}
```

```

    fColor.alpha = 255;

    // Calling the BControl version of this method is necessary because
    // BControl::Value() needs to return the proper value regardless of
    // which way the value was set.
    BControl::SetValue((fColor.red << 24) + (fColor.green << 16) +
                       (fColor.blue << 8) + 255);
    SetHighColor(col);
    Draw(Bounds());
}

```

```

void
ColorWell::SetValue(const uint8 &r, const uint8 &g, const uint8 &b)
{
    fColor.red = r;
    fColor.green = g;
    fColor.blue = b;
    fColor.alpha = 255;

    BControl::SetValue((fColor.red << 24) + (fColor.green << 16) +
                       (fColor.blue << 8) + 255);
    SetHighColor(r,g,b);
    Draw(Bounds());
}

```

```

void
ColorWell::SetStyle(const int32 &style)
{
    if (style != fStyle)
    {
        fStyle = style;
        Invalidate();
    }
}

```

```

int32
ColorWell::Style(void) const
{
    return fStyle;
}

```

```

void
ColorWell::Draw(BRect update)
{
    if (fStyle == COLORWELL_SQUARE_WELL)
        DrawSquare();
    else
        DrawRound();
}

```

```

rgb_color
ColorWell::ValueAsColor(void) const
{
    return fColor;
}

```

```

void
ColorWell::DrawRound(void)
{
    // Although real controls require more work to look nice, just a
    // simple black border will do for our purposes.
    if (IsEnabled())
        SetHighColor(fColor);
    else
        SetHighColor(fDisabledColor);

    FillEllipse(Bounds());

    SetHighColor(0,0,0);
    StrokeEllipse(Bounds());
}

```

```

void
ColorWell::DrawSquare(void)
{
    // The square version of our ColorWell doesn't get any more
    // complicated than the round one.
    if (IsEnabled())
        SetHighColor(fColor);
    else
        SetHighColor(fDisabledColor);

    FillRect(Bounds());

    SetHighColor(0,0,0);
    StrokeRect(Bounds());
}

```

None of these methods require very much thought. This shouldn't be surprising, though. The framework that Haiku provides for creating controls has just enough features to do much of the heavy lifting without being terribly complicated. Now let's round out the rest of the project with the necessary GUI to go with our new control.

### *App.h*

```

#ifndef APP_H
#define APP_H

#include <Application.h>

class App : public BApplication
{
public:
    App(void);
};

#endif

```

### *App.cpp*

```

#include "App.h"
#include "MainWindow.h"

```

```

App::App(void)
    : BApplication("application/x-vnd.jy-ColorWellDemo1")
{
    MainWindow *mainwin = new MainWindow();
    mainwin->Show();
}

```

```

int
main(void)
{
    App *app = new App();
    app->Run();
    delete app;
    return 0;
}

```

### *MainWindow.h*

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>
#include <MenuBar.h>

class ColorWell;

class MainWindow : public BWindow
{
public:
    MainWindow(void);
    void MessageReceived(BMessage *msg);
    bool QuitRequested(void);

private:
    BMenuBar *fMenuBar;
    ColorWell *fColorWell;
};

#endif

```

### *MainWindow.cpp*

```

#include "MainWindow.h"

#include <Application.h>
#include <Menu.h>
#include <MenuItem.h>
#include <View.h>

#include "ColorWell1.h"

enum
{
    M_SET_COLOR = 'stcl',
    M_COLOR_UPDATED = 'mcup',
    M_SET_SHAPE_CIRCLE = 'sscr',
    M_SET_SHAPE_SQUARE = 'sssq'
};

```

```

MainWindow::MainWindow(void)
:   BWindow(BRect(100,100,500,400),"ColorWell Demo",
           B_TITLED_WINDOW, B_ASYNCHRONOUS_CONTROLS)
{
    BRect r(Bounds());
    r.bottom = 20;
    fMenuBar = new BMenuBar(r,"menubar");
    AddChild(fMenuBar);

    // Create a background view to make the window look semi-normal. If
    // you're going to have a group of controls in a BWindow, it's best
    // to create a background view.
    r = Bounds();
    r.top = 20;
    BView *background = new BView(r, "background", B_FOLLOW_ALL,
                                  B_WILL_DRAW);

    // SetViewColor() sets the background color of a view. ui_color() is
    // a global C++ function which returns an rgb_color given one of the
    // system color constants. B_PANEL_BACKGROUND_COLOR happens to be the
    // default background color for BViews. By default, this is
    // (216, 216, 216).
    background->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
    AddChild(background);

    // Create our color well control. It's bigger than really necessary,
    // but it'll be fine for a demo.
    fColorWell = new ColorWell(BRect(15, 15, 165, 165), "color well",
                               new BMessage(M_COLOR_UPDATED));

    // Note that we call the background view's AddChild method here. If
    // two views which have the same parent overlap each other, the
    // results, according to the Be Book, are unpredictable. In most
    // cases, though, this sometimes results in drawing errors and many
    // times the view which was added later doesn't receive mouse events.
    background->AddChild(fColorWell);

    BMenu *menu = new BMenu("Color");
    fMenuBar->AddItem(menu);

    // BMessages can have data attached to them. As such, they are an
    // incredibly flexible data container. Here we are just going to
    // attach color values to the message for each color menu item. This
    // is not the standard way of attaching a color, but it will work
    // well enough for this example.
    BMessage *msg = new BMessage(M_SET_COLOR);
    msg->AddInt8("red", 160);
    msg->AddInt8("green", 0);
    msg->AddInt8("blue", 0);
    menu->AddItem(new BMenuItem("Red", msg, 'R', B_COMMAND_KEY));
    msg = new BMessage(M_SET_COLOR);
    msg->AddInt8("red", 0);
    msg->AddInt8("green", 160);
    msg->AddInt8("blue", 0);
    menu->AddItem(new BMenuItem("Green", msg, 'G', B_COMMAND_KEY));

    msg = new BMessage(M_SET_COLOR);
    msg->AddInt8("red", 0);

```

```

msg->AddInt8("green", 0);
msg->AddInt8("blue", 160);
menu->AddItem(new BMenuItem("Blue", msg, 'B', B_COMMAND_KEY));

menu = new BMenu("Shape");
fMenuBar->AddItem(menu);

menu->AddItem(new BMenuItem("Square",
    new BMessage(M_SET_SHAPE_SQUARE), 'S', B_COMMAND_KEY));
menu->AddItem(new BMenuItem("Circle",
    new BMessage(M_SET_SHAPE_CIRCLE), 'C', B_COMMAND_KEY));
}

```

```

void
MainWindow::MessageReceived(BMessage *msg)
{
    switch (msg->what)
    {
        case M_SET_COLOR:
        {
            // Yank the stored color values and plonk them into the
            // ColorWell.
            int8 red, green, blue;
            msg->FindInt8("red", &red);
            msg->FindInt8("green", &green);
            msg->FindInt8("blue", &blue);

            fColorWell->SetValue(red, green, blue);
            break;
        }
        case M_SET_SHAPE_CIRCLE:
        {
            fColorWell->SetStyle(COLORWELL_ROUND_WELL);
            break;
        }
        case M_SET_SHAPE_SQUARE:
        {
            fColorWell->SetStyle(COLORWELL_SQUARE_WELL);
            break;
        }
        default:
        {
            BWindow::MessageReceived(msg);
            break;
        }
    }
}

```

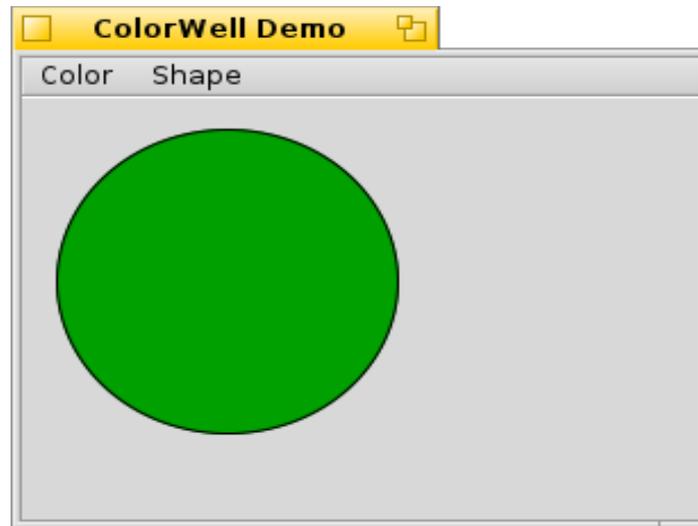
```

bool
MainWindow::QuitRequested(void)
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return true;
}

```

With all of this code in place, build your program and you should have a working demo app which displays a big swatch of color which can be changed to suit our whims. It doesn't do

much except display the current color – right now it makes a good companion to a BColorControl because one edits the color and the other displays it. Not bad.



### ***Final Thoughts***

At this point we have a very basic control. If this were going to just be a quick and dirty class to get a job done in a much larger project, we could probably get away with leaving it as it is, but since the goal is a full-fledged Haiku control, there is more that will need to be done. It isn't very often you will need to implement all of the different possible bells and whistles that we will be putting into our ColorWell class, but knowing how to use each set of features will open up new ideas that you may not have had otherwise.

### ***Going Further***

This isn't just some exercise. It's your code, too, so try thinking up some uses you might have for this control in your other projects and perhaps some extra features you might want to implement that could come in handy.

# Programming with Haiku

## Lesson 18

Written by DarkWorm



## Application Scripting

One of the most complicated – and least understood – parts of the Haiku operating system is its system for scripting applications. This is mostly because it looks like a complicated mess to the uninitiated, and compared to the C++ API, it is. Unfortunately, this is the price paid for the incredible flexibility it provides. The official documentation on the subject in the Be Book doesn't exactly make it any easier, either. By the end of this lesson, you should have a good grasp of the way that application scripting is done with Haiku.

Application scripting abstracts away all of the C++ details of the API and provides a way to interact with an application using nothing but messages to manipulate objects and properties. While any message can be technically sent to an application to script it, here is a list of the official scripting message constants which can be used to describe most of the interface.

Command	Description
B_COUNT_PROPERTIES	Get the number of instances of a property.
B_CREATE_PROPERTY	Create a new instance of a property. Note that this can't add new properties to a suite – it can only create new instances of an existing property.
B_DELETE_PROPERTY	Delete an instance of a property.
B_EXECUTE_PROPERTY	Run a property like a method.
B_GET_PROPERTY	Get the value of a property.
B_SET_PROPERTY	Set the value of a property.

## Scripting Suites

These commands don't help us very much without one other key piece of the puzzle: getting the list of properties for a particular object, i.e. the list of scripting interfaces an object supports. The BHandler class is the foundation of Haiku's scripting support. Each scriptable object inherits from the BHandler class, so every one of them has the properties Suites, Messenger, and InternalName. This interface, known in the lingo as a **suite**, has the name "vnd.Be-handler", which is not a MIME type even though it bears a striking resemblance to one.

The Suites property is of particular interest to us. If we ask an object to GET it, it will reply with a message which contains a list of the suites and properties that it supports. We can test this out using the Terminal command `hey`, which comes bundled with Haiku. Run this command from the Terminal:

```
hey Tracker getsuites
```

This certainly looks simple enough. `hey` hides some of the complexity for us. The output from the above command reminds us of the beast with which we are dealing.

```
Reply BMessage(B_REPLY):
"suites" (B_STRING_TYPE) : "suite/x-vnd.Be-TRAK"
"suites" (B_STRING_TYPE) : "suite/vnd.Be-application"
"suites" (B_STRING_TYPE) : "suite/vnd.Be-looper"
```

```

"suites" (B_STRING_TYPE) : "suite/vnd.Be-handler"
"messages" (B_PROPERTY_INFO_TYPE) :
  property  commands  specifiers  types
-----
  Trash    B_DELETE_PROPERTY  DIRECT
           Usage: delete Trash # Empties the Trash
  Folder   B_CREATE_PROPERTY  DIRECT      RREF
           Usage: create Folder to path # creates a new folder
  Preferences B_EXECUTE_PROPERTY  DIRECT
           Usage: shows Tracker preferences

"messages" (B_PROPERTY_INFO_TYPE) :
  property  commands  specifiers  types
-----
  Window    Usage:      INDEX REV.INDEX
  Window    Usage:      NAME
  Looper    Usage:      INDEX REV.INDEX
  Looper    Usage:      ID
  Looper    Usage:      NAME
  Name      B_GET_PROPERTY  DIRECT      CSTR
           Usage:
  Window    B_COUNT_PROPERTIES  DIRECT      LONG
           Usage:
  Loopers   B_GET_PROPERTY  DIRECT      MSNG
           Usage:
  Windows   B_GET_PROPERTY  DIRECT      MSNG
           Usage:
  Looper    B_COUNT_PROPERTIES  DIRECT      LONG
           Usage:

"messages" (B_PROPERTY_INFO_TYPE) :
  property  commands  specifiers  types
-----
  Handler    Usage:      INDEX REV.INDEX
  Handlers   B_GET_PROPERTY  DIRECT      MSNG
           Usage:
  Handler    B_COUNT_PROPERTIES  DIRECT      LONG
           Usage:

"messages" (B_PROPERTY_INFO_TYPE) :
  property  commands  specifiers  types
-----
  Suites     B_GET_PROPERTY  DIRECT      (suites CSTR)
(messages SCTD)
  Messenger  B_GET_PROPERTY  DIRECT      MSNG
           Usage:
  InternalName B_GET_PROPERTY  DIRECT      CSTR
           Usage:

"error" (B_INT32_TYPE) : 0 (0x00000000)

```

What a mess! There is a way to make sense of it, though. We are looking at a dump of the different suites that the main part of Tracker provides. There are four of them:

```

"suite/x-vnd.Be-TRAK"
"suite/vnd.Be-application"
"suite/vnd.Be-looper"
"suite/vnd.Be-handler"

```

The rest of the information printed is a dump of each suite in the same order as the list at the top. Instead of trying to figure all four of them out at the same time, let's just look at the last one and pick it apart.

A better way to describe this information would be like this:

Property Name	Command	Specifier	Return Type
Suites	Get	Direct	String array, BPropertyInfo array
Messenger	Get	Direct	BMessenger
InternalName	Get	Direct	String

## Specifiers

The word `Direct` used in the third column of the table is the specifier for a scripting command. Specifiers are the way that properties are referenced. Here is a list of the available specifiers:

Specifier	Description
<code>B_DIRECT_SPECIFIER</code>	Used by properties where no additional information is needed to reference the property. This is the specifier used by non-array properties, like <code>Name</code> or <code>Frame</code> .
<code>B_NAME_SPECIFIER</code>	A string is used to determine which property instance is to be used.
<code>B_ID_SPECIFIER</code>	A unique 32-bit integer is used to reference a particular property instance.
<code>B_INDEX_SPECIFIER</code>	A 32-bit integer specifies a property instance the same way that you would an index for an array.
<code>B_REVERSE_INDEX_SPECIFIER</code>	This works the same way, but counts from the end of the array. A reverse index of -1 is the last element in the list.
<code>B_RANGE_SPECIFIER</code>	One or more consecutive property instances can be referenced at one time.
<code>B_REVERSE_RANGE_SPECIFIER</code>	This works starting from the end of the array instead of the beginning. More on both range specifiers can be found below in the section on using the scripting API from C++.

Most – but not all – properties can be referenced using only one kind of specifier. When the `hey` command prints out all of the properties in a suite, it lists each property by specifier type. The `"vnd.Be-application"` suite used by `BApplication` objects could be described using our simplified form like this:

Property Name	Command	Specifier	Return Type
Window	Any*	Index, Reverse Index	Object
Window	Any*	Name	Object

Property Name	Command	Specifier	Return Type
Window	Count	Direct	int32
Looper		Index, Reverse Index	Object
Looper		Name	Object
Looper		ID	Object
Looper	Count	Direct	int32
Name	Get	Direct	String
Windows	Get	Direct	BMessage
Loopers	Get	Direct	BMessage

A window, for example, can be referenced by index, reverse-order index, or by name. Getting the window itself isn't very useful, but accessing a window's properties is. Here are a few of the more useful properties in the "vnd.Be-window" window suite:

Property Name	Command	Specifier	Return Type
Title	Get, Set	Direct	String
Frame	Get, Set	Direct	BRect
MenuBar	Any	Direct	Object
View	Count	Direct	Int32
View	Any*	Name, Index, Reverse Index	Object

Let's play around a bit with StyledEdit using these Terminal commands:

```
open /system/apps/StyledEdit
hey StyledEdit set Title of Window "Untitled 1" to "Haiku Rocks"
hey StyledEdit set Frame of Window 1 to "BRect(100,100,500,400)"
hey StyledEdit get Title of Window -1
```

First, we open StyledEdit, then we change the title of the first blank document window to "Haiku Rocks" and both move and resize the window. StyledEdit's Window 0 happens to be the Open File dialog window, which is not really what we want. Lastly, we obtain the title of the last window.

Any time a property returns an object, it means that an additional specifier must be used to get at the subobject's properties. We can home in on a Window's views or menus this way.

```
hey StyledEdit get InternalName of View -1 of View 1 of Window -1
```

The name returned is `textview`. Very interesting! We just homed in on the `BTextView` used to hold the text in a document window! Perhaps the most amazing part about it all is that we didn't use one bit of C or C++ to do it. `hey` provides an interface to the scripting API which can be used easily from any language.

## ***Concluding Thoughts***

The scripting API that Haiku provides is very deep and powerful, but far underutilized or understood. With the ground we have covered here, you should be starting to get an idea of its potential. Any Haiku application can be "remotely controlled" in a limited way without any extra effort from the developer. An obscure program from the days of BeOS called DogWhistle used scripting with Tracker to create a unique file management application, for example. With a little work, your programs can be leveraged by others to do something which was previously not possible.

## ***Going Further***

- Fire up the accompanying demo application *Scripting Explorer* and use it to tinker around with some of the programs bundled with Haiku. See what interfaces some of the programs provide. How could they be used by others?
- What programs bundled with Haiku are currently of limited scripting use and could be expanded to be much more powerful? For what could some of them be used?
- For a real thinking challenge, what might go into a library designed to program the API purely from the scripting API? How could it be implemented?

# Programming with Haiku

## Lesson 19

Written by DarkWorm



In our previous lesson we explored the Haiku scripting API and how it could be manipulated with the `hey` Terminal command. Now we will go deeper and get into the mechanics of using C++ for scripting.

## ***More About Haiku Messaging***

At first, the thought of using C++ to manipulate another program via the scripting API might seem pointless, but there are reasons to do so. One of them is for the sheer power it affords. `hey` has its limits, the most significant of which is its inability to use the `Execute` command and the need to escape certain kinds of data in order to get around the `bash` shell. Using C++ for scripting requires an understanding of the Haiku API's messaging classes that goes a little further than passing acquaintance, but it confers the full power made available by the scripting interface.

Most GUI development doesn't involve much mucking around with the messaging classes because the API has been designed well. The vast majority of the time we're responding to messages and explicitly sending them only once in a while. Let's take a quick peek at the messaging classes which we'll use for scripting.

`BHandler` and `BLooper` do the grunt work for message handling. `BHandlers` respond to messages. `BLoopers` receive a message and pass it through the list of attached `BHandlers` until one of them decides to do something with it, but because the `BLooper` class is a subclass of `BHandler`, `BLoopers` can also respond to a message. This happens most often when sending a message to a `BWindow` and its `BView`-based controls – `BWindow` is a subclass of `BLooper` and `BView` is a direct subclass of `BHandler`.

`BMessenger` is the means of transportation for all of these scripting messages getting sent around. It targets either a `BLooper` or `BHandler`, regardless of if its target is in your program or somewhere out there in the system, and provides a way to send messages to this target.

## ***C++ Scripting***

Scripting with C++ is not that different from using `hey`. More typing is needed, but the concepts are almost exactly the same: each `hey` command is a single message, the command is merely the `BMessage`'s `what` property, and everything else is a specifier manipulated by `BMessage`'s specifier-related methods.

Let's start by translating a `hey` command – which has a more human-friendly interface – to C++. Here it is:

```
hey StyledEdit get Title of Window 1
```

This asks `StyledEdit` for the title of its first document window. There are three pieces of data to use: the target, the scripting action, and the list of specifiers. Code to send this message looks like this:

```
#include <Message.h>
#include <Messenger.h>
#include <String.h>

#include <stdio.h>
```

```

int
main(void)
{
    status_t status;
    // Point the messenger at the StyledEdit application. The status
    // parameter is set to an error condition if StyledEdit is not
    // running.
    BMessenger messenger("application/x-vnd.Haiku-StyledEdit", -1,
                          &status);
    if (status == B_OK)
    {
        // Set the command
        BMessage msg(B_GET_PROPERTY), reply;

        // Set the specifiers. Note that just like the hey command,
        // they are added in order of most specific to least.
        msg.AddSpecifier("Title");
        msg.AddSpecifier("Window", 1L);
        if (messenger.SendMessage(&msg, &reply) == B_OK)
        {
            // Just about any time a scripting message returns an
            // error, it is the constant B_MESSAGE_NOT_UNDERSTOOD.

            BString title;
            if (reply.FindString("result", &title) == B_OK)
                printf("Title of StyledEdit Window 1: %s\n",
                      title.String());
            else
                printf("Couldn't get title of StyledEdit Window"
                      " 1\n");
        }
    }
    else
        printf("StyledEdit does not appear to be running.\n");

    return 0;
}

```

As you can see, it's not any more difficult to send messages from C++ even if it involves more typing. Understanding the why and how of scripting for Haiku is the hardest part. Getting the suites from a target is involved, but not difficult. The code below obtains the handled suites for the first document window in StyledEdit, parses them, and prints them out in a shorter, more English-like format. Observe:

```

#include <Message.h>
#include <Messenger.h>
#include <PropertyInfo.h>
#include <String.h>

#include <stdio.h>

int
main(void)
{
    // C++ version of "hey StyledEdit get Suites of Window 1"
    status_t status;
    BMessenger messenger("application/x-vnd.Haiku-StyledEdit", -1,
                          &status);

```



```

        break;
    }
    case B_GET_PROPERTY:
    {
        cmdLabel = "Get";
        break;
    }
    case B_SET_PROPERTY:
    {
        cmdLabel = "Set";
        break;
    }
    default:
        break;
}

if (cmdLabel.CountChars())
{
    if (cmdIndex > 0 &&
        commands.CountChars() > 0)
        commands << ", ";
    commands << cmdLabel;
}

    cmdIndex++;
} // end while (commands)

if (commands.CountChars() == 0)
    commands = "None";

int32 specIndex = 0;
if (info[j].specifiers[0] == 0)
    specifiers = "All";
else
    while (info[j].specifiers[specIndex])
    {
        BString label;
        switch (info[j].specifiers[specIndex])
        {
            case B_DIRECT_SPECIFIER:
            {
                label = "Direct";
                break;
            }
            case B_NAME_SPECIFIER:
            {
                label = "Name";
                break;
            }
            case B_ID_SPECIFIER:
            {
                label = "ID";
                break;
            }
            case B_INDEX_SPECIFIER:
            {
                label = "Index";
                break;
            }
        }
    }

```

```

        case B_REVERSE_INDEX_SPECIFIER:
        {
            label = "ReverseIndex";
            break;
        }
        case B_RANGE_SPECIFIER:
        {
            label = "Range";
            break;
        }
        case B_REVERSE_RANGE_SPECIFIER:
        {
            label = "ReverseRange";
            break;
        }
        default:
            break;
    }

    if (label.CountChars())
    {
        if (specIndex > 0 &&
            specifiers.CountChars() > 0)
            specifiers << ", ";
        specifiers << label;
    }

    specIndex++;
} // end while(specifiers)

if (specifiers.CountChars() == 0)
    specifiers = "None";

printf("%s: %s (%s)\n", info[j].name,
        commands.String(), specifiers.String());

if (info[j].usage && strlen(info[j].usage) > 0)
    printf("\t%s\n", info[j].usage);
} // end for each property

printf("\n");

    i++;
} // end for each suite
} // end while each suite name
} // end if status == B_OK
else
    printf("StyledEdit does not appear to be running.\n");

return 0;
}

```

Talk about a lot of code to do something that doesn't amount to much! This is the most complicated part of Haiku scripting. Of course, if the suite you are looking for is one of the standard ones defined by the Haiku API, then all of this work is not even necessary.

## Implementing Scripting Support

Making your own classes respond to scripting messages beyond the system defaults is a great way to make your programs even more powerful and enable possibilities that you may or may not have even imagined. Depending on what you wish to do, implementing them may require more than a little work. We'll use our ColorWell control for examples of what can be done.

Two requirements must be met to implement additional scripting support: an object must be a subclass of BHandler and it must implement three key BHandler hook functions: GetSupportedSuites(), MessageReceived(), and ResolveSpecifier(). The exact steps taken look like this:

1. Create a static property\_info structure to hold the description of your control's suite.
2. Implement GetSupportedSuites().
3. Tweak your control's MessageReceived() to test for messages with specifiers and handle them separately from regular messages.
4. Write ResolveSpecifier().

An array of property\_info structures are used to define your control's scripting interface. The definition of the structure looks like this:

```
struct property_info
{
    // The name of the property this structure describes.
    char *name;

    // A zero-terminated list of supported commands. Use a zero for the
    // first command to act as a wildcard which matches any command.
    uint32 commands[10];

    // A zero-terminated list of supported specifiers. Use a zero for the
    // first specifier to act as a wildcard which matches any specifier.
    uint32 specifiers[10];

    // A string which describes the property.
    char *usage;

    // Extra space for your own use, if you like. The system won't
    // touch it.
    uint32 extra_data;
};
```

For our ColorWell class, we will define four properties: IsRound, a boolean value which sets the style to round if true and rectangular if false, and three integer properties: Red, Green, and Blue. These three will support getting and setting of the individual color values of the ColorWell's color. Here is the resulting suite definition:

```
static property_info sColorWellProperties[] =
{
    { "IsRound", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
      { B_DIRECT_SPECIFIER, 0 },
      "True if the color well is round, false if rectangular.", 0,
      { B_BOOL_TYPE }
    },
    { "Red", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
```

```

        { B_DIRECT_SPECIFIER, 0 },
        "The red value for the color well.", 0,
        { B_INT32_TYPE }
    },
    {
        "Green", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
        { B_DIRECT_SPECIFIER, 0 },
        "The green value for the color well.", 0,
        { B_INT32_TYPE }
    },
    {
        "Blue", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
        { B_DIRECT_SPECIFIER, 0 },
        "The blue value for the color well.", 0,
        { B_INT32_TYPE }
    },
};

```

Now that the suite definition is done, writing `GetSupportedSuites()` is a piece of cake. It will almost always do nothing more than add the name of the suite to the message it is given, add the static property list as a flattened `BPropertyInfo` instance, and return the parent class' version of the function.

```

BHandler *
ColorWell::ResolveSpecifier(BMessage *msg, int32 index,
                           BMessage *specifier, int32 what,
                           const char *property)
{
    BPropertyInfo propertyInfo(sColorWellProperties);
    if (propertyInfo.FindMatch(msg, index, specifier, what, property)
        >= 0)
        return this;

    return BControl::ResolveSpecifier(msg, index, specifier, what,
                                      property);
}

```

Don't worry about the `BPropertyInfo` object in the sample. This class does about as little as it can get away with – it wraps around the `property_info` structure to provide some convenience functions, the most useful of which are `Flatten()`, `Unflatten()`, and `FindMatch()`.

The go-between work which bridges the scripting interface's properties to your control is done in `MessageReceived()`. Luckily, the glue code is nice and simple.

```

void
ColorWell::MessageReceived(BMessage *msg)
{
    // Our ColorWell class doesn't handle any special messages, so if
    // a message doesn't have any specifiers, we just pass it to the
    // parent class' version.
    if (!msg->HasSpecifiers())
        BControl::MessageReceived(msg);

    // Scripting depends on a reply message.
    BMessage reply(B_REPLY);

    // These variables will hold information about the current specifier.
    status_t status = B_ERROR;
}

```

```

int32 index;
BMessage specifier;
int32 what;
const char *property;

if (msg->GetCurrentSpecifier(&index, &specifier, &what, &property)
    != B_OK)
    return BHandler::MessageReceived(msg);

// FindMatch() searches its property_info array for a property
// which matches the specifiers in the message. It returns an index
// to the element which matches or -1 if a match was not found.
BPropertyInfo propInfo(sColorWellProperties);
switch (propInfo.FindMatch(msg, index, &specifier, what, property))
{
    // These cases are the glue code which make each property
    // do something. Just like a regular MessageReceived() case,
    // it passes unrecognized properties to the parent class.
    case 0: // IsRound
    {
        if (msg->what == B_SET_PROPERTY)
        {
            bool isRound;
            if (msg->FindBool("data", &isRound) == B_OK)
            {
                SetStyle(isRound ? COLORWELL_ROUND_WELL :
                    COLORWELL_SQUARE_WELL);
                status = B_OK;
            }
        }
        else
        if (msg->what == B_GET_PROPERTY)
        {
            reply.AddBool("result",
                Style() == COLORWELL_ROUND_WELL);
            status = B_OK;
        }
        break;
    }
    case 1: // Red
    {
        if (msg->what == B_SET_PROPERTY)
        {
            int32 newValue;
            if (msg->FindInt32("data", &newValue) == B_OK)
            {
                rgb_color color = ValueAsColor();
                color.red = newValue;
                SetValue(color);
                status = B_OK;
            }
        }
        else
        if (msg->what == B_GET_PROPERTY)
        {
            rgb_color color = ValueAsColor();
            reply.AddInt32("result", color.red);
            status = B_OK;
        }
    }
}

```

```

        break;
    }
    case 2: // Green
    {
        if (msg->what == B_SET_PROPERTY)
        {
            int32 newValue;
            if (msg->FindInt32("data", &newValue) == B_OK)
            {
                rgb_color color = ValueAsColor();
                color.green = newValue;
                SetValue(color);
                status = B_OK;
            }
        }
        else
        if (msg->what == B_GET_PROPERTY)
        {
            rgb_color color = ValueAsColor();
            reply.AddInt32("result", color.green);
            status = B_OK;
        }
        break;
    }
    case 3: // Blue
    {
        if (msg->what == B_SET_PROPERTY)
        {
            int32 newValue;
            if (msg->FindInt32("data", &newValue) == B_OK)
            {
                rgb_color color = ValueAsColor();
                color.blue = newValue;
                SetValue(color);
                status = B_OK;
            }
        }
        else
        if (msg->what == B_GET_PROPERTY)
        {
            rgb_color color = ValueAsColor();
            reply.AddInt32("result", color.blue);
            status = B_OK;
        }
        break;
    }
    default:
        return BControl::MessageReceived(msg);
}

// If we were not able to handle one of our messages, we have
// an error condition. We need to describe the error using strerror
// and make it clear that something went wrong.
if (status != B_OK)
{
    reply.what = B_MESSAGE_NOT_UNDERSTOOD;
    reply.AddString("message", strerror(status));
}

```

```

    // We return an error code even if we were successful.
    reply.AddInt32("error", status);

    msg->SendReply(&reply);
}

```

ResolveSpecifier() can be short and simple or long and more complicated depending on the kinds of properties that your control uses. The purpose of this method is to determine which handler is supposed to receive and respond to the message.

```

BHandler * ResolveSpecifier(BMessage *msg, int32 index,
                           BMessage *specifier, int32 what,
                           const char *property);

```

msg points to the scripting message being passed around. specifier contains the current specifier which is found at the index index. what holds the what value of the specifier message and property contains the name of the targeted property.

There are a few different ways that your properties can respond to a scripting message, and the amount of code involved in implementing ResolveSpecifier() is a direct result of these actions. The first way is for a property to return a BHandler which is attached to some other BLooper. The second is when a property returns a BHandler which is attached to the same BLooper as your control. The third is to return a value which is handled by your control, such as a calculated value or the result of calling your control's methods.

### *ResolveSpecifier() Method #1: Handler in a Remote Looper*

BApplication uses this method to resolve its Window property. This is an excerpt from Haiku's source code for BApplication. Here the BApplication is handling a specifier for a window by index or reverse index.

```

if (propInfo.FindMatch(message, 0, specifier, what, property, &data) >= 0)
{
    switch (data) {
        case kWindowByIndex:
        {
            int32 index;
            err = specifier->FindInt32("index", &index);
            if (err != B_OK)
                break;

            if (what == B_REVERSE_INDEX_SPECIFIER)
                index = CountWindows() - index;

            BWindow *window = WindowAt(index);
            if (window != NULL) {
                message->PopSpecifier();
                BMessenger(window).SendMessage(message);
            } else
                err = B_BAD_INDEX;
            break;
        }
    }
}

```

At the end of BApplication's version of this function is a call to return NULL. The key here is the call to PopSpecifier() and the NULL return value. The scripting message is passed on to the proper messenger, but the specifier is popped off so that the target doesn't try to resolve the same one, and NULL is returned by the function because the BLooper is no longer responsible to resolve the specifier – the targeted BLooper will take it from here.

### *ResolveSpecifier() Method #2: Handler in the Current Looper*

BView takes this route to resolve its View property. If the view has children, they will obviously belong to the same looper.

```
case 4: // View property
{
    if (!fFirstChild) {
        err = B_NAME_NOT_FOUND;
        replyMsg.AddString("message", "This window doesn't have "
            "children.");
        break;
    }

    // Get the child view based on what method was used
    BView* child = NULL;
    switch (what) {
        case B_INDEX_SPECIFIER: {
            int32 index;
            err = specifier->FindInt32("index", &index);
            if (err == B_OK)
                child = ChildAt(index);
            break;
        }
        case B_REVERSE_INDEX_SPECIFIER: {
            int32 rindex;
            err = specifier->FindInt32("index", &rindex);
            if (err == B_OK)
                child = ChildAt(CountChildren() - rindex);
            break;
        }
        case B_NAME_SPECIFIER: {
            const char* name;
            err = specifier->FindString("name", &name);
            if (err == B_OK)
                child = FindView(name);
            break;
        }
    }

    // Pass the message to the proper child...
    if (child != NULL) {
        msg->PopSpecifier();
        return child;
    }

    // ...or not, if it wasn't found
    if (err == B_OK)
        err = B_BAD_INDEX;

    replyMsg.AddString("message",
        "Cannot find view at/with specified index/name.");
}
```

```
        break;
    }
```

While this way also calls `PopSpecifier()` to avoid repeatedly processing the same specifier, it returns a non-NULL value because the BHandler targeted belongs to the same BLooper that initiated the specifier resolution.

### *ResolveSpecifier() Method #3: Resolution*

Most specifiers will be resolved by the target that receives them, and this is, by far, the most common way of resolving specifiers. In these instances, your control will return itself. If it isn't able to resolve everything, the control should return the parent class' version of `ResolveSpecifier()`.

```
BHandler *
ColorWell::ResolveSpecifier(BMessage *msg, int32 index,
                           BMessage *specifier, int32 what,
                           const char *property)
{
    BPropertyInfo propertyInfo(sColorWellProperties);
    int32 index = propertyInfo.FindMatch(msg, index, specifier, what,
                                        property);
    // If the property happens to be one in ColorWell's list, return
    // the ColorWell object.
    if (index >= 0)
        return this;

    // If we make it this far, it means that it's not one we recognize,
    // so we will return the inherited version.
    return BControl::ResolveSpecifier(msg, index, specifier, what,
                                     property);
}
```

### **Concluding Thoughts**

Having finished with all three scripting-related functions, we almost have a completely finished control. Sending messages to it via `hey` works the way it should, so now it's possible to change its color remotely. What could possibly be left? Only a little, as we'll see in the next lesson.

### **Going Further**

- Setting each color channel individually is a lot of work. Come up with a way to set all three at once.
- Create properties to set the color using the HSL (Hue, Saturation, Luminance) color model.

# Programming with Haiku

Lesson 20

Written by DarkWorm



## ***Drag and Drop Support***

Drag and drop is one of the easiest ways to integrate your program with others in the system. Many times developers will create their new 133t project in a vacuum, forgetting the fact that the user does not use this new project in a vacuum, but often in conjunction with other programs. For example, working on a document in a word processor may also require occasionally jumping to an image editor for adding pictures to the user's document. By adding drag and drop support, you enable the user to work much faster. Depending on how you implement it in your programs, adding drop support can be dead simple, with drag support only a little harder.

According to the Be Book, within Haiku there are two ways to do drag and drop: the simple way and the negotiated way. Both rely on messaging to get the job done. The simple way packages some data into a message which is sent to the target when the button is released. The receiving program recognizes the data and takes appropriate actions. Negotiated drag and drop, while conjuring up images of a shady character in some back alley asking a mild-mannered businessman about a watch, works like this:

1. The sender places into the drag message format-related data fields, which we'll get to in a moment.
2. The receiver reads through the format fields and replies with a request for the desired data format and possibly the desired actions to be performed on the data.
3. The sender receives the reply, performs any requested actions on the data, and sends the data itself in the requested format.
4. The receiver reads the data from this second message and does whatever it chooses with it.

## ***Simple Drag and Drop***

If you do nothing else in your programs, handle simple drops. Very little effort on your part is required to do so because it's just a little extra code in a BView's `MessageReceived()` method which does this:

```
void
MyControl::MessageReceived(BMessage *msg)
{
    if (msg->WasDropped())
    {
        // Here's where your drop support goes.
        entry_ref ref;
        int32 i = 0;
        while (msg->FindRef("refs", i++, &ref) == B_OK)
            printf ("File dropped: %s\n", ref.name);
    }

    switch (msg->what)
    {
        default:
        {
            BView::MessageReceived(msg);
            break;
        }
    }
}
```

Our example here doesn't do very much. The most common form that drop support takes is dragging one or more files from a Tracker window and dropping them onto a control in your program. When this happens, the message has an entry\_ref in the "refs" field for each file dropped. Once your program has read each ref, it can do whatever it likes. The above example just prints the name of each file dropped to the Terminal. Many times your program will check each files' type and open those which it supports.

Simple drag support is almost as easy. It will require the implementation of a few of BView's methods which we have not discussed before: `MouseDown()` and `MouseMoved()`. The main purpose behind `MouseDown()` is to start tracking the mouse. `MouseMoved()` is implemented in order to actually start the drag operation using the `DragMessage()` method. Once `DragMessage()` is invoked, the system handles the rest.

```
void DragMessage(BMessage *msg, BRect rect, BHandler *replyTarget = NULL);
void DragMessage(BMessage *msg, BBitmap *bitmap, BPoint pt,
                 BHandler *replyTarget = NULL);
void DragMessage(BMessage *msg, BBitmap *bitmap, BPoint pt,
                 drawing_mode mode, BHandler *replyTarget = NULL);
```

`DragMessage()`, which initiates the drag session itself, is the key function in implementing drag support. The message passed to it contains all of the data to be passed to the drop target. The `BRect` version displays an outline as the user is dragging. The other two versions of `DragMessage()` display a picture while the user is dragging. The third version also enables specifying a different drawing mode for displaying the bitmap, which makes possible nifty effects like drag previews which use transparency. `replyTarget` is only used for negotiated drag and drop, which we will learn about later – for simple drag and drop, we just leave it `NULL`.

Here is a simple example for a `BView` which performs drag and drop.

### *DragView.h*

```
#ifndef DRAGVIEW_H
#define DRAGVIEW_H

#include <View.h>

class DragView : public BView
{
public:
    DragView(BRect frame);
    void MouseDown(BPoint pt);
    void Draw(BRect update);

private:
    BRect fDragRect;
};

#endif
```

### *DragView.cpp*

```
#include "DragView.h"

enum
{
```

```

        M_DRAG = 'drag'
};

DragView::DragView(BRect frame)
    : BView(frame, "dragview", B_FOLLOW_LEFT | B_FOLLOW_TOP,
            B_WILL_DRAW),
      fDragRect(10, 10, 50, 50)
{
}

void
DragView::MouseDown(BPoint pt)
{
    // For our example, we only support dragging with the left button
    BPoint temp;
    uint32 buttons;
    GetMouse(&temp, &buttons);

    if (fDragRect.Contains(temp))
    {
        // SetMouseEventMask can only be called from within
        // MouseDown(). It forces all mouse events to be sent to this
        // view until the button is released. This saves us from having
        // to manually code the mouse tracking.
        SetMouseEventMask(B_POINTER_EVENTS, 0);

        BMessage dragMsg(M_DRAG);
        dragMsg.AddInt32("buttons", buttons);
        DragMessage(&dragMsg, fDragRect);

        // If you allocate the drag message on the heap, make sure
        // you delete it after DragMessage() returns. If you use a
        // BBitmap version of DrawMessage(), DON'T delete the bitmap
        // after calling DragMessage() because the system will do
        // that for you.
    }
}

void
DragView::Draw(BRect update)
{
    SetHighColor(0, 0, 160);
    FillRect(fDragRect);

    DrawString("Try dragging the square", BPoint(fDragRect.left,
                                                fDragRect.bottom + 50));
}

```

As you can see the only significant part of this code is the section in `MouseDown()`. Everything else is just fluff to make the demo a little nicer.

## ***Negotiated Drag and Drop Support***

This kind of drag and drop is only rarely used because the simple version requires very little effort and typically gets the job done. Still, this version provides a much greater opportunity

for other programs to interact with yours because the negotiation process bridges the gap between two programs that otherwise know nothing about each other.

### *Drag Initiation*

While the actual process of detecting and initiating the drag operation is the same as for simple drag and drop, the message itself needs to follow a particular protocol. First of all, the what field of the message needs to be `B_SIMPLE_DATA`. Second, the message needs to add the formats your program is willing to provide in a `BMessage`. Last, the supported actions for the data need to be added. There are also some optional fields which may be added that are detailed below.

The data formats supported by your program are placed in the `be:types` field as a list of MIME type strings. If your program is willing to pass data by way of a file – which is a good thing for large chunks of data – your program will need to add the `B_FILE_MIME_TYPE` constant, defined in `<MimeType.h>`, to the `be:types` field and populate the `be:filetypes` and `be:type_descriptions` fields. Note that if your program can pass data via a `BMessage` or a file, you should add the other types first and add `B_FILE_MIME_TYPE` last. If your program will only send data via a file, then `B_FILE_MIME_TYPE` should be the first (and only) value in the field. The `be:filetypes` field is just like `be:types` – a list of MIME type strings. `be:type_descriptions` holds a string description of the format which may or may not be displayed to the user.

Not only is the data format negotiable, the delivery method is also. The `be:actions` field is populated by your program with constants for supported actions on the data. There are four of them plus two legacy actions which are specific to Tracker, but are no longer supported.

<b>Constant</b>	<b>Description</b>
<code>B_COPY_TARGET</code>	Your program can send a copy of the data.
<code>B_MOVE_TARGET</code>	Your program can send a copy of the data and delete the original, "moving" the data to the receiving program.
<code>B_LINK_TARGET</code>	Your program can send a symlink to the data.
<code>B_TRASH_TARGET</code>	Your program can delete the data without sending it. One possible use for this is to drag something from your program to the Trash.
<code>B_COPY_SELECTION_TO</code>	This one is Tracker-specific. It is sent when there are one or more items in a Tracker window and it is willing to copy them somewhere. The destination is specified in the <code>refs</code> field of the negotiation message if this action is requested. While supported at one time and documented in the Be Book, as of this writing Tracker no longer supports this action, but it may be in the future.
<code>B_MOVE_SELECTION_TO</code>	This is also Tracker-specific. It works just like <code>B_COPY_SELECTION_TO</code> , but moving the files instead. This, too, is no longer supported by Tracker but may be in a future revision.

## Optional Drag Message Fields

Although `be:types` and `be:actions` are required fields, there are a few optional fields besides them. `be:clip_name` is a suggested name for the dropped data which the receiver can ignore or use if your program provides it. `be:originator` and `be:originator_data` are used for asynchronous messaging.

If your program needs to do negotiated drag and drop asynchronously, then using the `be:originator` and `be:originator_data` fields may not be a bad idea. The intent behind the fields is to provide a way for your program to identify its own drag and drop negotiation messages and track state information during the process. `be:originator` should be filled with something that identifies your program. What exactly this entails is up to you – it could be your program's signature, handler token for the sending BView, or something else. The only code that will be interacting with it will be your own. `be:originator_data` can be filled with data that your program will need to complete the drag and drop negotiations. Of course, if your program doesn't need this field, it can be easily left out without any ill effects.

`be:data` was used in the original drag and drop protocol, but it has been since deprecated. It is mentioned here in case your program interacts with very old or very long-lived BeOS applications.

The `_drop_point_` and `_drop_offset_` fields are BPoint fields which give some coordinate information for the drag. Both are automatically added by the system. `_drop_point_` contains the screen coordinates of the mouse when the data was dropped. `_drop_offset_` is the distance from the top left corner of the drag rectangle.

## Drag Negotiation

Once the drag recipient has received the initial drag message, it is up to the recipient to decide what it wants done and how. The actions listed in `be:actions` are to become the what value for the reply. The message itself is expected to have at least one of these fields:

Field	Description
<code>be:types</code>	One or more strings specifying the formats the recipient is willing to accept data in a BMessage.
<code>be:filetypes</code>	One or more strings specifying the formats the recipient is willing to accept data in a file.
<code>directory</code>	An <code>entry_ref</code> pointing to the directory in which the data file is to be created. The recipient is responsible for ensuring that the directory is valid.
<code>name</code>	A string containing the name of the file for the data. The recipient is responsible for ensuring that the name is available.

None of the fields are required if they don't apply. The sender doesn't even have to respond to a `B_TRASH_TARGET` message – it just needs to toss the requested data in the can.

## Receiving the Data

A data message is sent if and only if a message-based format was specified in the negotiation message – no message is sent if data is to be sent only by file. The data message has the value

B\_MIME\_DATA and contains one field. The field's name is the MIME type for the data, e.g. a "text/plain" field containing a string which is the requested data. Once the data has been received, the recipient is free to do with it whatever it likes.

The Be Book does not mention the procedure for obtaining data from a file-based negotiation and there are no known applications as of this writing which use negotiated drag and drop, but there is still a way to handle this. Considering that file-based transfers are best used for large chunks of data, asynchronously monitoring for the data file is best. Using the Node Monitor would be the easiest way to watch for the creation of the file.

### *A Negotiated Summary*

No wonder negotiated drag and drop isn't used much! Here is a quick summary to remember everything:

1. The sender makes a B\_SIMPLE\_DATA message with format strings in be:types and be:actions and possibly be:filetypes, be:type\_descriptions, be:clip\_name, be:originator, or be:originator\_data.
2. The recipient replies with a message of the action desired, such as B\_COPY\_TARGET, and attaches the desired be:types and possibly be:filetypes, directory, and name, depending on the action requested and the methods desired.
3. If the data is to be sent via BMessage, the sender replies to the negotiation message with a B\_MIME\_DATA message with one field named after the data's MIME type which contains the data.

### *Implementing Simple Drag and Drop*

Our ColorWell class should definitely allow drag and drop, both for receiving colors dropped onto it and dragging its color elsewhere. There are no official standards, the protocol used by an ancient color picker called roColour is supported by many applications, including Tracker, so we will follow suit. Change the beginning of the MessageReceived() function of ColorWell.cpp to this:

```
void
ColorWell::MessageReceived(BMessage *msg)
{
    // Handle simple roColour-style drag and drop
    if (msg->WasDropped())
    {
        rgb_color *c;
        ssize_t size;
        if (msg->FindData("RGBColor", B_RGB_COLOR_TYPE,
            (const void **)&c, &size) == B_OK)
        {
            SetValue(*c);
            return;
        }
    }

    if (!msg->HasSpecifiers())
        BControl::MessageReceived(msg);
}
```

That little chunk of code is all that we need to enable the user to drop colors onto a ColorWell control and it will change to that color. The code for dragging colors to other targets involves a bit more. We'll need to implement `MouseDown()`.

```
void
ColorWell::MouseDown(BPoint pt)
{
    SetMouseEventMask(B_POINTER_EVENTS, 0);

    // Create a color bitmap to show the color drop. This version of the
    // BBitmap constructor allows for BBitmaps which can accept and be
    // drawn upon by BViews.
    BRect r(0, 0, 15, 15);
    BBitmap *bitmap = new BBitmap(r, B_RGB32, true);
    BView *view = new BView(r, "", 0, 0);

    bitmap->Lock();
    bitmap->AddChild(view);
    view->SetHighColor(fColor);
    view->FillRect(view->Bounds());
    view->Sync();
    bitmap->Unlock();

    BMessage msg;
    msg.AddPoint("click_location", pt);
    msg.AddData("RGBColor", B_RGB_COLOR_TYPE, &fColor,
               sizeof(rgb_color));

    DragMessage(&msg, bitmap, BPoint(12, 12));
}
```

There are three main sections of code to our implementation of `MouseDown()`. The first section is the call to `SetMouseEventMask()`. This function exists for our convenience and can only be called from within `MouseDown()`. Our view will continue to receive mouse event messages until the user releases the mouse button, dramatically reducing the amount of code we need to write to handle the drag. The second section creates a small `BBitmap` which will add feedback during the drag operation. The last part creates the drag message, adds the color and click location to it, and initiates the drag operation.

## ***Manipulating BBitmaps***

Of special note in our `MouseDown()` code is the way that the `BBitmap` is created and modified. It is possible to create a `BBitmap` which accepts a `BView` just like a `BWindow`. `BViews` attached to a `BBitmap` draw on the bitmap instead of the screen. This is wildly convenient because we don't have to use an external library or learn another API to get some nice graphics. There are three constructors which enable this:

```
BBitmap(BRect bounds, color_space mode, bool acceptViews = false,
        bool needsContiguousRAM);
BBitmap(const BBitmap *source, bool acceptViews = false,
        bool needsContiguousRAM);
BBitmap(const BBitmap &source, uint32 flags);
```

The third version takes a series of flags. While there are others defined in `Bitmap.h`, here are the more useful ones:

<b>Flag</b>	<b>Description</b>
B_BITMAP_CLEAR_TO_WHITE	The bitmap is initialized to white when created.
B_ACCEPTS_VIEWS	The bitmap will accept child BViews.
B_BITMAP_IS_CONTIGUOUS	The physical memory allocated for the bitmap will be contiguous. This only really matters to drivers doing direct memory access.
B_BITMAP_NO_SERVER_LINK	This is a Haiku-only extension in which the bitmap is created without any connection to the app_server. This has the benefit of being very lightweight, but BViews cannot draw them with DrawBitmap().

It is wise to understand some of the behind-the-scenes action which takes place when working with BBitmaps. For performance reasons, when a BBitmap is created, the app\_server actually allocates the memory in an area shared with your program's BApplication object. While this speeds up BView::DrawBitmap() calls, it also forces your program to have a BApplication object to use them. Also, when you allocate a bitmap which can accept BViews, the app\_server actually creates a BWindow counterpart on its side. Because of this, be conservative with the number of bitmaps which accept children. Otherwise, your application will crash. If you need to draw a lot of bitmaps, consider using an external graphics library.

### ***Concluding Thoughts***

After this thorough examination of Haiku's drag-and-drop facilities – especially the negotiated version – it may seem like this is more work than it is worth. In practice, adding drag and drop support to your programs is very little work because the negotiated version is used only very rarely. The simple method really is simple and giving the user the option to directly manipulate the interface of your program is a great way to save time for your target audience.

# Programming with Haiku

## Lesson 21

Written by DarkWorm



## Haiku Replicants

*"Fiery the angels rose, and as they rose deep thunder roll'd. Around their shores: indignant burning with the fires of Orc."*

Replicants in the world of Haiku are nothing short of amazing, but they have nothing to do with *Blade Runner*. Instead, they are very similar to a component technology. For those unfamiliar, components are an object-oriented programming concept where a program consists of objects with a generic interface to their specific features. Built into the Haiku API is a means of interacting with a BView-based object that does not require knowing anything about it. Back in 1995, BeOS could embed a web browser into the desktop without creating a major security risk like Windows 95. Sadly, not much was done with replicants after Be introduced them, but more has been done with them by the Haiku developers in recent years. We'll take some time in this lesson to see how they are put together.

## Archiving and Instantiation

One of the topmost classes in the hierarchy of the Haiku API is the BArchivable class. Its purpose is to provide an interface which enables child classes to save their state into a BMessage which can be sent to a target or flattened and saved to disk. In case you haven't noticed the trend over the course of these lessons, BMessage is both a floor wax and a dessert topping.

Only three functions must be implemented to make an object archivable: Archive(), Instantiate() and a version of the object's constructor which takes a BMessage as its only parameter. Archive() saves the object's state into a BMessage and Instantiate() loads its state from a BMessage. Archiving is easiest to understand in the context of an existing control, so let's examine some new code for the ColorWell class to explain.

```
status_t
ColorWell::Archive(BMessage *data, bool deep) const
{
    status_t status = BControl::Archive(data, deep);
    data->AddString("class", "ColorWell");

    return status;
}
```

The Archive() method does four things: call the inherited method to make sure parent classes can save any appropriate properties, store the class' name into the class property of the BMessage, place any state information of the object into the BMessage, and save any extra information if deep is true.

In our example here, we don't have to store any extra information. The BControl version of Archive() saves the control's value. Our ColorWell class stores the value as both an rgb\_color structure and an integer. When we unarchive a ColorWell instance, we'll just convert the integer value – which is saved by BControl::Archive() – to the rgb\_color and we'll have the exact same state as what was saved into the message.

More complicated objects may place other properties into the BMessage archive. Unfortunately, there is no defined naming protocol for these properties, so we have to be careful about name choices to avoid collisions. The easiest way is to save properties using the class name like this:

```
msg->AddString("MyClassName:MyStringProperty","SomeString");
msg->AddBool("MyClassName:SomeOtherProperty",someBooleanFlag);
```

The deep flag is often ignored by simpler objects. Its purpose is to signal to the object to go above and beyond the call of duty to save its state. For example, if a BView is given a deep archive request, it also saves the states of its children.

Now that we have examined how to save the state of an object, loading is next. We will define the other two methods here:

```
ColorWell::ColorWell(BMessage *data)
: BControl(data)
{
    // The inherited BControl constructor will pull the integer value
    // of the color out of the message for us, so all we have to do
    // is update fColor. Instead of copying and pasting code, we'll
    // call SetValue() to do all of the heavy lifting for us. This
    // kind of trick eases debugging and code maintenance.
    SetValue(Value());
}
```

```
BArchivable *
ColorWell::Instantiate(BMessage *data)
{
    if (validate_instantiation(data, "ColorWell"))
        return new ColorWell(data);

    return NULL;
}
```

Instantiate() is boilerplate code. validate\_instantiation() is just a check to make sure that the BMessage holds the information for a ColorWell object. Assuming that the check passes, it returns a ColorWell instance using the unarchiving constructor mentioned above. This constructor just loads data from the BMessage in the same way that data was saved in Archive().

To instantiate an archived object, call instantiate\_object() like this:

```
BArchivable *archivable = instantiate_object(msg);

MyDesiredClass *object = NULL;
if (archivable)
    object = dynamic_cast<MyDesiredClass*>(archivable);
```

The Be Book advocates using the preprocessor macro cast\_as(), but this is deprecated. Use a dynamic\_cast.

Although it's possible to unarchive just about any archive message, there is no defined protocol for discovering what is kept in an archive – Be expected developers to follow predefined protocols, such as what is used for replicants.

## Creating a Replicant

A replicant is simply an archivable BView control. Assuming that each child control can be archived and instantiated properly, all that is needed to create a replicant is to add an instance of the BDragger class.

```
BDragger(BRect frame, BView *target, int32 resizeMode = B_FOLLOW_NONE,
         uint32 flags = B_WILL_DRAW);
```

BDragger objects are the key to creating replicants. They are a child class of BView and have a target BView, much like BScrollView. They come with a few conditions that are expected to be met:

1. A BDragger expects its target to be its parent, child, or sibling in the view hierarchy.
2. If the BDragger instance is the child of its target, it must be the target's only child view and its frame must be only as big as the drag handle – 7 pixels.
3. If the BDragger instance is the target's parent, then its frame must be at least as big as its target.

Here is an example of how simple it is to make a replicant. This is the source code to the main view from the Haiku demo application OverlayImage.

```
/*
 * Copyright 1999-2010, Be Incorporated. All Rights Reserved.
 * This file may be used under the terms of the Be Sample Code License.
 *
 * Authors:
 *           Seth Flexman
 *           Hartmuth Reh
 *           Humdinger           <humdingerb@gmail.com>
 */

#include "OverlayView.h"

#include <Catalog.h>
#include <InterfaceDefs.h>
#include <Locale.h>
#include <String.h>
#include <TextView.h>

#undef B_TRANSLATE_CONTEXT
#define B_TRANSLATE_CONTEXT "Main window"

const float kDraggerSize = 7;

OverlayView::OverlayView(BRect frame)
:
  BView(frame, "OverlayImage", B_FOLLOW_NONE, B_WILL_DRAW)
{
  fBitmap = NULL;
  fReplicated = false;

  frame.left = frame.right - kDraggerSize;
  frame.top = frame.bottom - kDraggerSize;
```

```

BDragger *dragger = new BDragger(frame, this, B_FOLLOW_RIGHT |
                                     B_FOLLOW_BOTTOM);
AddChild(dragger);

SetViewColor(B_TRANSPARENT_COLOR);

fText = new BTextView(Bounds(), "bgView", Bounds(), B_FOLLOW_ALL,
                     B_WILL_DRAW);
fText->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
AddChild(fText);
BString text;
text << B_TRANSLATE(
    "Enable \"Show replicants\" in Deskbar.\n"
    "Drag & drop an image.\n"
    "Drag the replicant to the Desktop.");
fText->SetText(text);
fText->SetAlignment(B_ALIGN_CENTER);
fText->MakeSelectable(false);
fText->MoveBy(0, (Bounds().bottom - fText->TextRect().bottom) / 2);
}

```

```

OverlayView::OverlayView(BMessage *archive)
:
  BView(archive)
{
  fReplicated = true;
  fBitmap = new BBitmap(archive);
}

```

```

OverlayView::~~OverlayView()
{
  delete fBitmap;
}

```

```

void
OverlayView::Draw(BRect)
{
  SetDrawingMode(B_OP_ALPHA);
  SetViewColor(B_TRANSPARENT_COLOR);

  if (fBitmap)
    DrawBitmap(fBitmap, B_ORIGIN);
}

```

```

void
OverlayView::MessageReceived(BMessage *msg)
{
  switch (msg->what) {
  case B_SIMPLE_DATA:
  {
    if (fReplicated)
      break;

    entry_ref ref;
    msg->FindRef("refs", &ref);

```

```

        BEntry entry(&ref);
        BPath path(&entry);

        delete fBitmap;
        fBitmap = BTranslationUtils::GetBitmap(path.Path());

        if (fBitmap != NULL) {
            if (fText != NULL) {
                RemoveChild(fText);
                fText = NULL;
            }

            BRect rect = fBitmap->Bounds();
            if (!fReplicated) {
                Window()->ResizeTo(rect.right, rect.bottom);
                Window()->Activate(true);
            }
            ResizeTo(rect.right, rect.bottom);
            Invalidate();
        }
        break;
    }
    case B_ABOUT_REQUESTED:
        OverlayAboutRequested();
        break;

    default:
        BView::MessageReceived(msg);
        break;
}
}

```

```

BArchivable *OverlayView::Instantiate(BMessage *data)
{
    return new OverlayView(data);
}

```

```

status_t
OverlayView::Archive(BMessage *archive, bool deep) const
{
    BView::Archive(archive, deep);

    archive->AddString("add_on", "application/x-vnd.Haiku-OverlayImage");
    archive->AddString("class", "OverlayImage");

    if (fBitmap) {
        fBitmap->Lock();
        fBitmap->Archive(archive);
        fBitmap->Unlock();
    }

    return B_OK;
}

```

```

void
OverlayView::OverlayAboutRequested()
{
    BAlert *alert = new BAlert("about",
        "OverlayImage\n"
        "Copyright 1999-2010\n\n\t"
        "originally by Seth Flaxman\n\t"
        "modified by Hartmuth Reh\n\t"
        "further modified by Humdinger\n",
        "OK");

    BTextView *view = alert->TextView();
    BFont font;
    view->SetStylable(true);
    view->GetFont(&font);
    font.SetSize(font.Size() + 7.0f);
    font.SetFace(B_BOLD_FACE);
    view->SetFontAndColor(0, 12, &font);

    alert->Go();
}

```

Once you've constructed the target BView and the associated BDragger and placed them appropriately into the view hierarchy, there isn't anything else you need to do. The user can drag the BDragger handle and it will replicate the original view. Of course, this doesn't do much good unless you can drop it somewhere. This is where another class, BShelf, comes in.

BShelf is a BHandler which attaches to a BView and accepts replicants. There are three versions of the constructor:

```

BShelf(BView *view, bool allowDragging = true, const char *name = NULL);
BShelf(entry_ref *ref, BView *view, bool allowDragging = true,
    const char *name = NULL);
BShelf(BDataIO *stream, BView *view, bool allowDragging = true,
    const char *name = NULL);

```

The last two versions of the constructor initialize the shelf to a file where the shelf will save any replicants. If allowDragging is true, the user is allowed to move replicants around once they have been placed on the shelf. Otherwise, they stay where they were initially placed. The name of the shelf is important – if it has a name, the system will check to make sure that any replicants dropped onto it have a shelf\_type field which matches its name. Any replicants which do not have a matching name will be rejected.

Aside from the constructor, working with the BShelf is very straightforward. It is possible to programmatically add, remove, and count replicants attached to a shelf. The Save() method makes it possible for a shelf to have a state which is persistent from one program execution to another.

## ***Concluding Thoughts***

Long before the Windows Sidebar and Google Gadgets existed, BeOS had replicants. Like drag and drop support and scripting support, they are a wonderful technology which has been underutilized. Consider how your programs may use it. More than likely, you'll end up making work a little easier and a little more fun for your target audience.

This lesson also concludes our building of a full-featured, fully-implemented control. Most developers do not write their code this completely. Your code does not necessarily have to be, either. Pick and choose from the features which your program requires and as long as it works well, no one will know the difference or even care.