The Nitrogen Manifesto

Lisa Lippincott
lippin@zocalo.net

Abstract

With recent improvements such as Carbon events and the hierarchical control manager, the Carbon API has usurped much of the role formerly filled by class libraries like MacApp.

But one of Carbon's strengths, its language-neutral design, is also a weakness. This neutrality leads it to use weakly typed interfaces, error codes, resource disposal functions, and other error-prone patterns. While these features make using Carbon possible in C, they make using Carbon cumbersome in C++.

I propose to create an open-source C++ interface to Carbon. I call this library Nitrogen.

History

Several past attempts to create C++ libraries for Macintosh programming have been organized as class libraries: they identify the central abstractions of the operating system and attempt to capture each in a C++ class.

I believe that the class libraries have suffered from the grand scale of their ambitions. Redesigning the operating system in this manner is laborious, and such an effort necessarily lags behind the development of the operating system. Documenting these class libraries is a task on par with rewriting Inside Macintosh. And users of these libraries often find themselves in situations unanticipated by the library writers; then they must learn both the underlying operating system and the quirks of its interaction with the library.

Nitrogen

The Nitrogen project takes a less ambitious course. The aim of the project is to translate the C-oriented Carbon interface into idiomatic C++ without redesigning it. Nitrogen provides a C++ interface to Carbon in much the same way as other libraries provide interfaces to Pascal, Lisp, or Ada.

Nitrogen stands on two legs. The first leg is *Cyanogen*, a set of rules governing the relationship between Nitrogen and Carbon. By meticulously following these rules, Nitrogen achieves a uniformity which makes it easier to design, easier to learn, and easier to predict. In most cases, a user who understands the Cyanogen rules and a portion of Carbon will be able to use the corresponding portion of Nitrogen, or, if that portion has not yet been written, predict the form it will eventually take with enough accuracy to implement a compatible interface.

The second leg upon which Nitrogen stands is the *Nitrogen Nucleus*, a library which addresses in a general way the problem of adapting legacy C interfaces to C++. The Nitrogen Nucleus supports and extends the uniformity imposed by the Cyanogen rules by making those rules easy to follow, and by providing a central implementation of the patterns found in the rules.

The bulk of Nitrogen consists of wrappers for Carbon types and functions. For each Carbon type or function in the global namespace, there is a type or function in the Nitrogen namespace with the same name. Where practical, a Nitrogen type is identical to the Carbon type; otherwise each is implicitly convertible to the other. A Nitrogen function is similar in effect, but often differs in signature from its Carbon namesake, and there are sometimes several Nitrogen functions or function templates overloading a single Carbon name.

Nitrogen in Action

Perhaps the best way to grasp the Nitrogen approach is to see it in action. Here is a function from a Nitrogen version of Apple's sample application, Converter:

```
namespace N = Nitrogen;

void ConvertCommandHandler( WindowRef window )
  {
   double fahrenheitTemp =
      N::Convert<double>(
         N::GetControlData< kControlEditTextCFStringTag >(
            N::GetControlByID( window, fahrenheitControlID ) ) );

   double celsiusTemp = ( fahrenheitTemp - 32.0 ) * 5.0 / 9.0;

   ControlRef celsiusField =
      N::GetControlByID( window, celsiusControlID );

   N::SetControlData< kControlStaticTextCFStringTag >(
      celsiusField,
      N::Convert< N::Owned<CFStringRef> >( celsiusTemp ) );

   N::DrawOneControl( celsiusField );
  }
```

This function illustrates the way Nitrogen can handle many of the mundane aspects of the Macintosh operating system without fundamentally changing the design of a program. In this function, Nitrogen checks for errors coming from all of the Carbon calls which signal errors; it releases the two temporary CFStrings used in the function; and it also improves the type-safety of this function by associating the type CFStringRef to the constants kControlEditTextCFStringTag and kControlStaticTextCFStringTag.

But where Nitrogen really shines is in the way it handles callback functions. While Nitrogen provides functions that take UPP parameters, it also overloads these functions with templates which handle the UPP construction and provide glue for catching exceptions and navigating weakly-typed interfaces. For example, here is the installation of a Carbon event handler:

```
Point IdealWindowSize
   ( const MyContent&, WindowRef );

N::Owned< EventHandlerRef > handler =
 N::InstallWindowEventHandler
    < kEventClassWindow,
      kEventWindowGetIdealSize,
      kEventParamDimensions,
      const MyContent&,
      N::EventParamNameList
         <kEventParamDirectObject>,
      IdealWindowSize >
    ( myWindow, myContent );
```

In this example, Nitrogen constructs glue which extracts the WindowRef direct object parameter from the Carbon event, casts and dereferences the void* parameter to produce const MyContent&, calls IdealWindowSize, stores the result into the dimensions parameter of the event, and converts any exception thrown to an error code, which it returns.

The Nitrogen Nucleus

The Nitrogen Nucleus is a subset of Nitrogen devoted to the task of adapting legacy C interfaces to C++. The principal areas addressed by the Nucleus are type conversions, error codes, management of resource ownership, and saving values for later restoration. While these facilities provided by the Nucleus are designed for use with legacy code, many of them prove useful in wider contexts.

Conversions

Legacy interfaces provide a wealth of conversion functions, and adapting libraries to each other, or even to the C++ standard library, creates even more. To organize these functions, Nitrogen adopts a convention that's easy for humans to remember and practical for templates to make use of: all simple conversions are performed by a set of function templates named Convert.

The basic Convert template is

```
template < class Output, class Input >
Output Convert( const Input& );
```

In most situations, the input type can be deduced from the function parameter, so one may write "Convert<Output>(input)" to perform a conversion.

The Convert functions use a functor template, Converter, to perform conversions; conversions are added by specializing this template. The default implementation of Converter attempts an implicit conversion. The Convert function template is overloaded to allow for more than one parameter; these additional parameters are used to initialize the Converter object.

The Nitrogen Nucleus provides specializations of Converter for range-checked conversions between numeric types and stream-based conversions to and from the standard string types.

Error Codes

The traditional C++ approach to error codes returned by legacy interfaces has been to define a class type in which to wrap the error code, and throw objects of that class as exceptions. Nitrogen participates in this tradition and expands upon it.

One problem with the traditional approach has been that one cannot catch specific error codes, because the error codes differ only in value, rather than in type. To alleviate this problem, the Nitrogen Nucleus uses a template, ErrorCode, to generate a distinct subclass of the wrapper class for each error code. Thus, for example, the class

```
ErrorCode< Nitrogen::OSStatus, fnfErr >
```

represents a Carbon file-not-found error, and is a subclass of Nitrogen::OSStatus.

Nitrogen also provides a function template for throwing these classes. The function

```
template < class ErrorClass >
void ThrowErrorCode( ErrorClass )
```

looks up the particular error code in a table and throws the appropriate subclass of ErrorClass. Unfortunately, it's not practical for the Nucleus to construct the table of error codes on its own, but calling

```
RegisterErrorCode<ErrorClass, code>()
```

ensures the registration of a particular code in the table. When passed an error code not registered in this way, ThrowErrorCode throws the base class.

It's not enough to produce exceptions from error codes. Sometimes it's also necessary to produce error codes from exceptions. Nitrogen uses a token class, TheExceptionBeingHandled, to represent the current exception and provides a specialization of Converter to act on it. Using these, one may write

```
Convert<DesiredType>
    ( TheExceptionBeingHandled() )
```

to produce an error code from the current exception. The converter uses two strategies to perform the conversion: first, it tries to catch the exception as const DesiredType& and return that value; if that fails, it uses a list of types that can be converted to DesiredType by the Convert function. If both strategies fail, it rethrows the exception.

Like the exception throwing mechanism, the exception conversion list can't be constructed by Nitrogen on its own. To ensure that a particular conversion is listed, one may call

```
RegisterExceptionConversion
          < Output, Exception >()
```

For times when an error code is strictly required, one may supply a default value for the converter to return instead of rethrowing the exception:

```
Convert<DesiredType>
   ( TheExceptionBeingHandled(),
     defaultValue )
```

The exception conversion mechanism is encapsulated in a class, ExceptionConverter, which can also be used for producing error messages or in other situations where a global translation scheme is inappropriate.

Ownership

Managing the disposal of resources is error-prone in a C program, and correctly using C-style resource disposal functions in an exception-rich environment is nearly impossible. So C++ programs instead use resource management classes whose destructors release resources. The exemplar of this technique is the template std::auto_ptr, which manages the disposal of resources with delete.

The Nitrogen Nucleus provides a similar template, Owned, for managing the disposal of resources through function calls. Specializations of the templates OwnedDefaults and Disposer specify the functions used to dispose resources of various types, so that Nitrogen knows, for example, that the destructor of an Owned<WindowRef> should call DisposeWindow.

As with auto_ptr, the copy constructor and assignment operators of Owned transfer ownership of the owned resource from one object to another — the object copied or assigned from is left holding nothing. This effect may be used to make function signatures more expressive: when Owned<T> is used as a function parameter, it indicates a transfer of ownership from the caller into the function; and when used as a function result, it indicates a transfer of ownership out of the function.

The static member function Owned<T>::Seize is used to claim ownership of a freshly created resource, and the member function Release is used to release ownership without transferring it to another object.

For situations where an object may not have a single owner, or where the unusual copying and assignment semantics of Owned are unacceptable, the Nitrogen Nucleus also provides a template Shared. Copying a Shared object dilutes, rather than transfers ownership; the destructor of the last Shared object referring to a resource disposes of the resource. Shared objects are created by transferring ownership from an Owned object, and Shared provides a member function Unshare which transfers ownership back into an Owned object, on the condition that the resource has only one owner.

Scoped Changes

In addition to the disposal of allocated resources, C++ programs use destructors as an exception-safe mechanism for reverting variables to previously-saved values. The Nitrogen Nucleus provides two templates, Scoped and Tentative, which embody this idiom. Each takes a reference-like type as its template parameter, and is constructed from a reference of that type.

Scoped is the simpler of the two templates. Upon construction, a Scoped object stores a copy of the variable to which it is given a reference. During its lifetime, the Scoped object acts as a reference to that variable — assignment to the Scoped object causes assignment to the variable, and reading from the Scoped object produces the current value of the variable. When the Scoped object is destroyed, the original value is restored to the variable.

The template Tentative similarly stores the original value of a variable and acts as a reference to that variable. But with a Tentative object, the reversion to the original value can be avoided. If the member function Commit is called before destruction, the original value will not be restored. Tentative thus allows a series of changes to be grouped into a transaction; if an exception prevents the program from reaching the call to Commit, the changes will be rolled back.

Many properties that may benefit from being saved and restored are not stored in simple variables; instead, one must call functions to get and set their values. For this reason, the template parameters of Scoped and Tentative are not limited to simple references. They may also be proxy types that behave like references.

The Nitrogen Nucleus provides a template, Pseudoreference, which produces such a proxy type from a getter-setter function pair.

Destruction Exceptions

The problem of exceptions coming from destructors is a particularly thorny one. Three things are clear about the problem. First, errors will happen in destructors, particularly when legacy interfaces are involved. Second, it's bad to let these errors go unreported. And third, it's bad to have a program terminate because exceptions collided during stack unwinding.

A palatable way to deal with this situation has yet to be found. So, following age-old programming tradition, the Nitrogen Nucleus provides a flexible mechanism for choosing between the unpalatable alternatives. Each class may choose its own destruction exception policy, or use a default policy chosen by macro flags. If no flags are set, the default policy passes the exceptions to a handler function, and the default handler ignores them.

Wrappers for Integral Types

Legacy interfaces sometimes use integral types to represent things which aren't integers at all. This practice not only compromises type-safety, but creates problems in overload resolution and template instantiation. The Nitrogen Nucleus provides class wrappers for integral types through the templates IDType, SelectorType, and FlagType. Each of these templates takes three parameters: a tag type, an underlying integral type, and a default value. The tag type serves only to distinguish an instantiation of the template from other instantiations with the same underlying type and default value.

IDType is the simplest of these templates. It is used for types, such as file reference numbers, that are only produced by functions and have no numeric properties other than ordering. There are implicit conversions from IDTypes to their underlying integral types and vice-versa, but conversions to and from other integral types or IDTypes are blocked. For situations in which the implicit conversion is inadequate, IDType provides a member function Get, which returns the value as the underlying type, and a static member function Make, which produces an IDType object.

The template SelectorType is used for types whose values are given by constants in header files, but that have no numeric properties. In addition to conversions to and from the underlying type, there are implicit conversions to (but not from) SelectorTypes from int, unsigned int, long, or unsigned long, as necessary to allow enumerators to be converted to the SelectorType.

FlagTypes are similar to SelectorTypes, but also allow bitwise operations and are convertible to bool.

Cyanogen

Cyanogen is the set of rules by which the Nitrogen interface relates to Carbon. The fundamental goal of Cyanogen is to define Nitrogen in a way which facilitates good C++ programming practices while following the fundamental design of Carbon. Its secondary goals are to make Nitrogen easy to expand, easy to learn, and easy to use.

Namespaces

All Nitrogen functions are defined in the Nitrogen namespace. Since many identifiers in Nitrogen have the same names as identifiers in Carbon, a using-directive "using namespace Nitrogen" is rarely an adequate way to access Nitrogen identifiers. The recommended practice is to declare "namespace N = Nitrogen" and write "N::" in front of Nitrogen identifiers.

Writing "N::" for operator functions is awkward, and Koenig lookup won't find operators in the Nitrogen namespace unless one of the parameters has a Nitrogen type. To make its operators more readily available, Nitrogen provides a namespace Nitrogen::Operators containing using-declarations nominating all nonmember operators defined in the Nitrogen namespace. Any scope which needs these operators can make them available with the declaration "using Nitrogen::Operators."

There is one way in which Nitrogen makes excursions into the global namespace. Where the Carbon headers define an identifier needed by Nitrogen to be a macro, Nitrogen replaces the macro with an equivalent C++ construct, such as a constant definition or inline function. Preprocessor tests are used to ensure that these redefinitions only occur if the macro is defined.

Types
Each type name used in Carbon should also appear in the Nitrogen namespace. In most cases, the Nitrogen name is simply a synonym for the Carbon name, introduced with a using-declaration. Carbon types are never redefined by Nitrogen in order to add constructors or ordinary member functions; these purposes can be more compatibly achieved by writing nonmember functions. Nitrogen also does not redefine types in order to add destructors; instead, the Owned template is used to represent resource ownership and the unadorned type is used to refer to resources without claiming ownership.

Nitrogen does redefine types in order to improve type-safety. In particular, where Carbon uses a numeric type for nonnumeric purposes, Nitrogen wraps this type in a class type, usually using the IDType, SelectorType, or FlagType template. Likewise, where Carbon simulates inheritance by typedef-names for void*, such as CFPropertyListRef or CFTypeRef, Nitrogen provides wrapper types. To the extent that it is practical, these types provide all the conversions and operators necessary for them to be interchangeable and miscible with the Carbon type. And to cover situations where the wrapper type does not fit seamlessly, the wrapper provides a static member function Make for creating instances of the class from the Carbon type and a member function Get that returns the object’s value in the Carbon type.

Nitrogen introduces such a wrapper for a numeric type whenever the Carbon usage indicates a distinct type, even when Carbon does not give that type a name. For these types, Nitrogen uses the name that most closely matches the names given by Carbon to functions and variables related to the type. If Carbon uses more than one variation on a name, Nitrogen uses typedef declarations to make all of the variations equivalent.

In general, Nitrogen does not introduce wrapper classes in order to declare operator functions; instead, it declares nonmember operator functions. But the operators [], (), and -> may only be declared as member functions. When one of these operators is appropriate, Nitrogen defines a wrapper type. But if the Carbon type permits, Nitrogen arranges for templates such as Owned to accept either the original Carbon type or the wrapper class.

Nitrogen also introduces wrapper classes for the Pascal string types Str255, Str31, and so forth. These are all typedef names for specializations of template class Str<length>. This template to mimics the properties of the array types, adding only copy-construction and assignment operations.

Functions
While Nitrogen is conservative in redefining types, it is liberal in redefining functions. Most Carbon functions will have one or more Nitrogen wrappers; only a few are brought into the Nitrogen namespace with a simple using-declaration. The majority of changes instituted by the wrappers are simple changes to the error reporting, return type, or parameter list of the function. But in some cases — particularly when a single Carbon function is burdened with a great variety of responsibilities — the wrapper functions can be considerably more complex.

Exceptions
All Nitrogen functions report errors by throwing exceptions.

Errors represented by OSStatus codes are thrown as Nitrogen::OSStatus or as a class derived therefrom. Nitrogen provides the function ThrowOSStatus to simplify throwing these exceptions: this function does nothing when passed noErr; it throws an object of type ErrorCode<Nitrogen::OSStatus, code> when passed a registered error code; and it throws an object of type Nitrogen::OSStatus otherwise.

Calling the function RegisterOSStatus<code>() ensures that a particular error code is registered. Before throwing an OSStatus exception, Nitrogen wrappers for Carbon functions ensure the registration of every error code listed in the Inside Macintosh chapter describing the Carbon function.

In accordance with these rules, the Nitrogen wrappers for MemError, ResError, and QDError do not return values, but instead throw exceptions. If a Carbon function reports errors through one of these mechanisms, its Nitrogen wrapper calls the error-throwing function to ensure that the error is not ignored.

If a Carbon function indicates an error without providing an error code, say by returning a null pointer, Nitrogen declares a class FunctionName_Failed, and the Nitrogen wrapper for that function throws an object of that type. Iteration functions such as FrontWindow and GetNextWindow are not considered to have failed when they return a null pointer to indicate the end of the iteration.

Function Results

Nitrogen functions always return their results.

When a Carbon function has multiple results, its Nitrogen wrapper returns a structure type containing all of the results. The structure is named FunctionName_Result; all of its data members are public and each bears the name given to the corresponding parameter in Inside Macintosh. If one of the results is clearly primary, as in ResolveAliasFile, the result structure will have a conversion operator that converts it to the primary result.

A few functions, such as SetPt or RectRgn, can be viewed in two ways: either as an assignment-like operation that modifies a parameter, or as a construction-like operation with a result. In these cases, Nitrogen provides two overloaded wrapper functions, one following each interpretation.

Some Carbon functions, such as CFBundleGetDataPointerForName, return a void* result which often needs to be cast to some other type. Others, such as GetWRefCon, return a long integer that is often simply a substitute for void*. Nitrogen overloads these functions with templates that perform the appropriate casts.

Function Parameter Lists

A Nitrogen wrapper function takes its parameters in the same order as its Carbon counterpart, but may give some parameters different types, give some default values, or even omit some parameters entirely.

Trailing parameters are given default values when the choice of default value is clear. But this mechanism is not available for parameters early in the list. Where no ambiguity arises, Nitrogen therefore provides overloaded wrapper functions that allow early parameters with clear default values to be omitted.

Where pointers are passed as parameters to Carbon functions, Nitrogen uses references if possible. Pointers are used for parameters that are arrays, and for parameters that may be null; references are used in all other cases.

For Carbon types that have Nitrogen wrappers, Nitrogen uses the Nitrogen wrapper type rather than the Carbon type. But when such a type is used in a non-const reference parameter, Nitrogen overloads the function to accept either type.

The use of integral types to represent fixed-point numbers is a particularly heinous practice of the Macintosh operating system, a practice stemming from the poor floating-point capabilities of a long-abandoned processor. Nitrogen breaks with this practice: where a Carbon function uses a fixed-point type, its Nitrogen wrapper uses double.

Nitrogen also uses the C++ types bool, void*, and std::size_t where Carbon uses other types for similar purposes. In particular, Nitrogen replaces long by void* in contexts where it is used to hold a generic pointer.

Ownership

A Nitrogen function that creates a resource of type T returns the created object in a result of type Owned<T>; and a Nitrogen function which consumes such a resource receives it in a parameter of type Owned<T>. The Nitrogen headers for such functions ensure that the necessary specializations of OwnedDefaults and Disposer are available.

For resources that are released by a function that may fail, the specialization of Disposer uses the default destruction exception policy to handle failures within the destructors of the Owned objects. Since destruction exceptions are problematic, one may take weight off of this mechanism by making explicit release calls for these resources whenever practical. Unlike the destructors of the Owned objects, the releasing functions always throw their errors.

Nitrogen takes a broad view of ownership. Anything that can be closed, deleted, disposed, freed, released, removed, or otherwise gotten rid of is governed by this rule. For reference-counted objects, this rule applies to each counted reference.

Some resources, such as files, are usually created with the intent that they live until some external process disposes of them. The member function Owned<T>::Release will set the resource free in this fashion. Care should be taken to delay this call until the resource is ready to stand on its own — for example, one generally should not release a file until its contents have been written. In this way, one may avoid leaving behind half-written files when an exception is thrown.

Strings

The Macintosh C++ programmer is confronted with five kinds of strings: Pascal-style strings with a length byte, C-style null-terminated strings, CFStrings, std::strings, and std::wstrings. The general Nitrogen practice is that wrappers for Carbon functions use the same kind of string as the Carbon function. To ease the burden of this practice, Nitrogen provides conversions between most pairs of string types.

Following Macintosh convention, function parameters of type const char* are taken to be C strings, and parameters of type const unsigned char* are taken to be Pascal strings. While many Carbon functions produce Pascal strings, these cannot be function results, so Nitrogen functions instead return Pascal strings using the template Str. This template is not used as a parameter type.

Where Carbon functions use a pointer and length to refer to text, Nitrogen provides two wrapper functions. One uses the pointer and length, and the other uses either std::string or std::wstring. The class std::string is used for single-byte text, or for text where the character size is governed by a Macintosh script code; std::wstring is used for UTF-16 text.

Selectors

Some Carbon functions use selectors — integer parameters that select from a variety of actions. Nitrogen always provides simple wrappers for these functions which follow this pattern. But where the wrapper can be improved by knowledge of the particular selector used, Nitrogen overloads those simple wrappers with function templates that take the selector as a template parameter. In this way, the template wrapper for GetControlData can provide a result of the appropriate type and the template version of Gestalt can provide default values for some selectors.

Where selectors are used as template parameters, Nitrogen provides a traits class template, FunctionName_Traits, which describes the variable portions of the function template. If the result type varies, it declares a type name Result, and for each parameter that varies in type, it declares a type name ParameterName_Type.

Callbacks

Nitrogen takes a multilayered approach to Carbon functions that have callback parameters, overloading the function name to provide several degrees of convenience.

As a first layer, Nitrogen provides a function that takes a universal procedure pointer as a parameter and provides the usual Nitrogen facilities. Using the first-layer wrapper for InstallEventHandler, one might write

```
InstallEventHandler
  ( eventTarget, myUPP,
    typeCount, eventTypes, userData )
```

As a second layer, Nitrogen provides a function template that, rather than taking a UPP as a function parameter, takes a function pointer of the corresponding type as a template parameter. From this function pointer, it constructs a UPP of static lifetime using the template StaticUPP. Using this wrapper, one may write

```
pascal OSStatus
Handler( EventHandlerCallRef,
         EventRef,
         void * );

InstallEventHandler< Handler >
  ( eventTarget, typeCount,
    eventTypes, userData )
```

As a third layer, Nitrogen provides glue that puts a Carbon-style face on a Nitrogen-style function. The Cyanogen rules are applied to the callback function type to derive a callback type in the Nitrogen style — that is, one which throws exceptions, returns its results, and so forth. In addition, if there is a void* “user data” parameter passed through to the callback, it is moved to the front of the callback parameter list, and it is allowed to be of any data pointer or reference type.

The third layer of overloading is a function template that takes the user data type and a corresponding function pointer as template parameters, and provides the necessary Carbon-to-Nitrogen glue and UPP. Taking advantage of this glue, one may write

```
void Handler( WindowRef,
              EventHandlerCallRef,
              EventRef );

InstallEventHandler<WindowRef,Handler>
  ( eventTarget, typeCount,
    eventTypes, myWindow )
```

There may be cases in which overloading the layer two wrapper with a layer three wrapper would create an ambiguity; in such cases, the layer three wrapper replaces the layer two wrapper.

As a fourth layer, Nitrogen provides glue that navigates weakly-typed mechanisms for passing parameters and returning results. Callbacks such as Carbon event and Apple event handlers provide selectors to the operating system in order to read their parameters and return their results. The fourth-layer Nitrogen wrapper for a function of this kind is templated on these selectors, on any user data type, and on the function pointer. It uses these parameters to construct glue that extracts parameters and returns results, in addition to performing all the work done by the layer three glue. It is this fourth-layer interface that is used in this example:

```
Point IdealWindowSize
   ( const MyContent&, WindowRef );

InstallWindowEventHandler
    < kEventClassWindow,
      kEventWindowGetIdealSize,
      kEventParamDimensions,
      const MyContent&,
      N::EventParamNameList
         <kEventParamDirectObject>,
      IdealWindowSize >
    ( myWindow, myContent );
```

Proxies

To facilitate the use of the Scoped and Tentative templates, Nitrogen provides a proxy type for each getter-setter function pair in Carbon. The copy constructors and assignment operators of these types mimic the behavior of references. The names of these proxies are constructed by removing the word “Get” from the name of the getter function.
Nitrogen also provides proxies for abstractions, such as MenuItem and ControlPart, that are implicit in the Carbon interface as function parameter groups. The name of such a proxy is a singular noun phrase derived by removing the parts of the function names that describe their action. Those functions are then overloaded to accept either the original parameter group or the proxy type.

Finally, where Carbon provides facilities for accessing the elements of a collection, Nitrogen provides a proxy that, to the extent made possible by Carbon, acts as a reference to a C++ container. The names for these container proxies are plural noun phrases derived by removing words such as “First” or “Next” from the names of the functions used to access elements. For this purpose, Cyanogen considers “Infos” to be an acceptable plural of “Info.”

Conversions

Nitrogen rationalizes all the conversions in Carbon under the Convert rubric. In particular, Nitrogen aims to provide a full set of conversions between string types, between numeric types and string types, and between

Core Foundation types and standard C++ types.

The conversions between Core Foundation collections and C++ containers use the Convert template recursively to convert elements, and thus may be used to convert a wide variety of types to and from CFPropertyLists.

Header Files
The organization of header files in Nitrogen mirrors that of Carbon. If a Carbon identifier is made available, directly or indirectly, by including HeaderName.h, then both the Carbon identifier and the corresponding Nitrogen identifier are made available by including Nitrogen/HeaderName.h. Each Nitrogen header is guarded against multiple inclusion by a macro NITROGEN_HEADERNAME_H.

A Call to Action
Nitrogen is a work in progress; most of it hasn't yet been written. And by its nature, it is unlikely to ever be completed — like the libraries that have come before it, Nitrogen will always lag behind Apple's development of Carbon. But the design of Nitrogen is intended to make these gaps easy to bridge, and to make clear the direction Nitrogen will take across these gaps. And I believe that these properties make Nitrogen well-suited to an open-source effort.

It is my hope that programmers who have become accustomed to Nitrogen will find that the easiest way to deal with these gaps is to extend Nitrogen themselves, following the Cyanogen rules. I intend to collect that work, integrate it into the main Nitrogen code line, and make it available to everyone.

As Nitrogen expands to embrace all of Carbon, it will also expand to embrace a wider variety of C++ idioms. And to make this expansion possible, the Nitrogen Nucleus and the Cyanogen rules will need to grow to meet new challenges. I intend to make this growth a public effort.

To these ends, I am assembling a network of volunteers. Together, we will make the Macintosh a better place for C++.