Error handling policy

Background

For the discussion below we define the following terms:

> ***error reporting*** or ***logging****:* The process or mechanism for recording an error message.
>
> ***error notification***: The process or mechanism for notifying a user that an error has occurred and returning any messages that have been logged (reported)
>
> ***error status***: The process for indicating that an error has occurred.

The error handing mechanism used throughout the VAPOR 2.x code base (libraries, command line utilities, and vaporgui) is the MyBase class and it's static error setting/getting methods. These methods support error handling (reporting, notification, and status) via a combination of return code and configurable logging facility. Unfortunately, the error reporting and status facilities provided by MyBase are combined, and this has led to problems as discussed later on. C++ exceptions are not used (they were deemed to be not supported well enough when VAPOR development started 10+ years ago).

MyBase static methods:

Error reporting *and* error status:

```
static void    SetErrMsg(int errcode, const char *format, ...);

static void    SetDiagMsg(int errcode, const char *format, ...);

static const char  *GetErrMsg() {return(ErrMsg);}
```

Error notification:

```
static void SetErrMsgCB(ErrMsgCB_T cb) { ErrMsgCB = cb; };

static ErrMsgCB_T GetErrMsgCB() { return(ErrMsgCB); };

static void SetErrMsgFilePtr(FILE *fp) { ErrMsgFilePtr = fp; };

static const FILE  *SetErrMsgFilePtr() { return(ErrMsgFilePtr); };
```

Virtually all class objects in VAPOR libraries vdf and common inherit from MyBase.

Many, but not all class objects in params, flow, render "libraries" inherit from MyBase as well.

Key properties of 2.x mechanism

1. Formatted error messages are easy to construct.
2. Because methods are static global (out-of-scope) control is possible. E.g. An application can easily control the error notification with a single call to establish a file pointer to direct messages to, or provided an reporting callback function.
3. Failure of class constructors can be reported by checking the effectively global error code (via MyBase::GetErrCode()). This essentially works around the inability of constructors to return status, but unfortunately has resulted in unintended side effects.
4. Calling Set* methods on MyBase does not violate *const*. I.e. A *const* object or *const* method on an object can invoke MyBase::Set* methods.

Problems with current mechanism

1. The global scope of MyBase implies, combined with the fact that error reporting and error status are inseparable, means that side effects can, and often do occur. For example, an error can be encountered somewhere in the code, but not caught until some completely unrelated code segment calls MyBase::GetErrCode().

Options for improving error handling in 3.0

(1) Use C++ exceptions

This would mesh well with C++ programming practices, but would require massive code changes and would make the libraries difficult to use with other languages (e.g. C and Fortran)

(2) Make MyBase non-static

This would prevent side-effects, but would make it difficult for applications to easily control error reporting. E.g. A single static method could not be called that would send all messages to a file or a call back.

Global error code would not be possible (without sacrificing the ability to have const objects/methods)

(3) Adopt policies that would avoid current problems

Avoid use of MyBase's global error code for detecting errors (setting error status). This would imply guaranteeing that constructors never fail (easily done by adding a class initialization method when needed, e.g. open() or initialize()). Hence, method/function return values, not MyBase::GetErrCode(), would be the preferred mechanism for detecting errors.

Proposed 3.0 Error handling policy

Option (3) will be adopted for use in VAPOR with the guidelines described below.

Class objects and standalone functions:

MyBase's global error code should NOT be used for detecting errors (setting error status). Note: should we consider removing the error code set by SetErrMsg()?

All **non-private** class objects (objects that will be used outside of a single compilation unit) that are capable of incurring an error condition that prevent normal operation should inherit from MyBase.

Constructors must be no-fail, and therefore must not log error messages (i.e. call SetErrMsg()). An initialization method returning error status should be provided if warranted. Constructors may, however, log diagnostic messages with SetDiagMsg().

Error status and reporting: Any method/function that is capable of failure should indicate error status with a return code as described in the table below, and unless otherwise, noted should report an error message. **Note**: If the method is a class initialization method the return of an error indicates that the class instance is not in a usable state, and that results of calling subsequent methods on the class are undefined.

It is the responsibility of any calling function that invokes an error-code returning function or method to:

1. Propagate the error status up the call stack
2. Log any additional information via SetErrMsg() that might be helpful

Return type	**Description and action**
int	Unless otherwise documented method and functions that have integer return types are expected to return 0 upon success, and -1 on error. If a -1 is returned the method (or a nested method called by the returning method) should log an error message with MyBase::SetErrMsg()
pointers	Like integers, unless otherwise documented functions that return pointers are expected to return a non-NULL address on success, and a NULL on failure. If a NULL is returned the method (or a nested method called by the returning method) should log an error message with MyBase::SetErrMsg()
bool	Methods and functions that return boolean values must NOT log errors. These methods can be safely invoked knowing that they will never generate an error message by calling MyBase::SetErrMsg(). The return value may be used to indicate the success or failure of the method.
all other types	Any other return type indicates a no-fail method that always returns a usable value, and never calls MyBase::SetErrMsg()

Command line utilities

Error handling for command line utilities (e.g. wrvdfcreate, vdfcreate, etc) is relatively straightforward. Errors should be reported using the same mechanisms as used for class objects with the following important exception: **the main() function should return 0 on success and 1 on failure**. This is necessitated by the historical use of UNIX process exit codes.

Command line utilities are also responsible for user error notification. Error messages should minimally be written to stderr. The error message should be prefixed by the application's name, followed by a colon. For example:

```
wrfvdfcreate: Failed to open input file foo
```

Graphical User Interfaces

GUI's differ from command line utilities in a couple of notable ways:

1. User error notification should be performed via a "pop up". Error messages should not be directed to a file descriptor. However, an error log file may be helpful.
2. GUI's are, in most cases, expected to recover from error conditions and continue operation after user notification. Ideally the user should be able to take corrective action to address the error condition (e.g. select a different input file, or change a rendering setting).

OpenGL error handling

OpenGL errors require special attention. The OpenGL API is asynchronous, and very few OpenGL functions return status: errors conditions are not detected by the OpenGL library at the time an OpenGL function call is made, particularly when the function simply changes OpenGL's state. The mechanism provided by OpenGL for error checking is the OpenGL function glGetError(), which may be called at any time and will return the **most recent OpenGL error code**. This function **must** be called in a loop to get **all** of the error codes generated by the OpenGL library since the last time the error stack was drained with glGetError().

Two convenience methods are provided by the VAPOR code base: oglStatusOK(), and oglGetErrMsg(). The former returns a vector of **all** of the error codes since the last invocation. The latter takes a vector of error codes generated by oglStatusOK, and generates an error string by concatenating the error messages associated with the error code vector. T**hese convenience methods should, in general, be used in place of calling glGetError() directly.**

When to test for OpenGL errors?

The question of when to test for an OpenGL error via oglStatusOK() is not an easy one to answer. For one, the OpenGL specification doesn't specify what OpenGL calls will trigger checks for valid OpenGL state. The approach most programmers take is to sample the OpenGL state periodically (e.g. after making a bunch of OpenGL calls). Certainly, error state should be checked after swapping front and back buffers, as this triggers rendering. The vaporgui renderer base class currently calls oglStatusOK after invoking derived classes _paintGL methods. Thus, OpenGL error state is tested every time a specific renderer (e.g. DVR, Iso) is invoked. OpenGL code segments that are lengthy, complex, or known to be problematic may want to add additional tests.