

**New C++ interface makes it easier to integrate FINCAD Developer with your C++ Applications**

Product Managers and Client Services staff here at FINCAD have talked with a number of customers and prospective customers who were using FINCAD Developer for developing C++ applications. Some common problems were identified by listening to their comments about using the software. New users were finding the XLOPER data structure difficult to work with and most C++ developers were writing some sort of custom wrapper to make the XLOPER data structure easier to use in their C++ applications while users who were working with other programming languages did not need to do this. There were four common reasons that developers wanted to create their own wrapper which made the XLOPER data structure easier to work with.

- 1) Memory management – Their custom wrapper classes avoided memory management issues by handling this transparently in the classes constructor and destructor methods
- 2) Exceptions – The custom wrappers allowed developers to throw an exception rather than return an error code when dealing with return objects that are XLOPER arrays (i.e. The C++ way)
- 3) Array Initialization – They provided an abstraction which made it easier to initialize array elements. (i.e. No more pointer arithmetic)
- 4) Type Checking – Function wrappers defined enumerated data types for switch inputs so that the compiler could catch some invalid inputs rather than catching them at runtime.

**Overview**

The latest release of FINCAD Developer (v10) includes a new wrapper library for C++ that provides these desired features. It eliminates the need for C++ developers to spend time writing this type of wrapper themselves and it makes the product easier to integrate with their applications. The purpose of this article is to provide a technical description of how the new interface works internally. It includes sample code from the new interface and descriptions of what is going on behind the scenes. For examples of using the new C++ API you can use the Function Finder application in the Windows version to generate a code example for any function. If you have already written your own wrapper class which provides similar functionality then reading this article will make it easier for you to compare your version to ours and decide which one you want to use going forward. If you are new to C++ programming then you may also find this article useful as a training tool. If you are new to using FINCAD Developer then this article will allow you to understand how the C++ interface works in more detail.

**Matrix Class**

The new C++ wrapper library defines a namespace called `fincad` which includes two classes. One is an exception class and the other is used to simplify initializing and passing arrays. Let's start by showing the high level definition for the array class. The array class is called `Matrix` and its definition is below.

```
class FC_EXPORT Matrix
{
public:
    Matrix();
    Matrix(unsigned short int rows, unsigned short int columns=1);
    ~Matrix();
    inline operator LPXLOPER() { return _xl; };
    unsigned int num_rows() const;
    unsigned int num_columns() const;
    void take_ownership(LPXLOPER xl);
    double & operator ()(const unsigned short i, const unsigned short j=1);
    const double & operator ()(const unsigned short i, const unsigned short j=1) const;

private:
    LPXLOPER _xl;
    bool _took_ownership;

    Matrix (const Matrix &);
    Matrix& operator=(const Matrix&);
}; //class Matrix
```

The `Matrix` class abstracts the way that you create and setup function inputs or outputs which are arrays. The arrays are stored in memory the same way as they were when you were using the XLOPER data structure in the older C interface. However, the complexity associated with setting up the data structure is handled by the `Matrix`

class. As a starting point let us look at how the Matrix class handles the memory allocation for arrays. Below is the source code for the constructor for the Matrix class.

```
Matrix::Matrix(unsigned short int rows, unsigned short int columns)
{
    _xl = (LPXLOPER) malloc(sizeof(XLOPER) * (rows * columns + 1));
    _xl->xltype = FCtypeMulti;
    _xl->val.array.lpparray = _xl + 1;
    _xl->val.array.rows = rows;
    _xl->val.array.columns = columns;
    const unsigned int num = rows * columns;
    for(unsigned short int i = 1; i <= num; i++)
    {
        _xl[i].xltype = FCtypeNum;
        _xl[i].val.num = 0.0;
    }
    _took_ownership=false;
}
```

**Memory Management and Array Initialization**

The Matrix::Matrix constructor allocates the memory necessary to store an array of the desired size. For a one dimensional array, the columns input argument is assigned a default value of 1 in the header for the class. So you can create a one dimensional array with a statement like “fincad::Matrix hl(9);” and a two dimensional array with a statement like “fincad::Matrix bondseries\_tbl(3,7);”. The constructor function then allocates the memory for you, assigns the pointer to a private member variable in the Matrix class and initializes the array elements to zero. Once an instance of the Matrix class has been created, you can get the array size information by using the num\_rows() and num\_columns() member functions. Indexes for the Matrix arrays start at one. To set the second element in a one dimensional array to a value use a syntax like “hl(2) = 36942;” and to initialize the second column in the first row use a syntax like “bondseries\_tbl(1,2) = 36009;”. The \_took\_ownership member of the Matrix class is used by the C++ wrapper functions to give control of memory for arrays that were allocated inside the FINCAD math library (i.e. return arrays) to the Matrix class. You will not need to use this member function since it is only needed internally by the C++ wrapper functions. The source for the destructor function for the Matrix class is below. The tilda (~) in front of the function name is part of the standard C++ naming convention for a routine that is called when an instance of the class goes out of scope. (i.e. is no longer in use)(aka a destructor method)

```
Matrix::~Matrix()
{
    if(_took_ownership) fc_free(_xl);
    else free(_xl);
    _xl = 0;
}
```

The destructor for the class calls an appropriate routine to free the memory for the array. If the memory was allocated inside the FINCAD math library then it does this using the fc\_free function otherwise the ordinary free function is used. So, the constructor and destructor together handle the memory management for you and avoid the need for you to do this explicitly yourself. There is also a default constructor for the Matrix class that takes no inputs. This is used by the C++ wrapper functions internally as a safety mechanism to ensure that the return arrays are not used prematurely before the internal function call has returned the results. Now that we have seen how the Matrix class handles memory management and makes it easy to setup the array elements.

**Exception Handling**

Let us look next at how the interface handles exceptions. The definition for the exception class is below.

```
class FC_EXPORT api_exception
{
public:
    api_exception(unsigned int error_code, const char * error_string, const char * function_name);
    virtual ~api_exception();
    const char *what() const
    {
        // return pointer to message string
        return _error_message->c_str();
    }
}
```

```

    }
    unsigned int error_code()
    {
        // return error code
        return _error_code;
    }
private:
    std::string* _error_message;
    unsigned int _error_code;
};

```

The C++ wrapper contains a function called FC\_CHECK\_XL\_AND\_THROW which uses the api\_exception class to throw an exception when an internal function call in the C++ wrapper for a given function returns an error. The function is part of the fincad namespace and the source for the FC\_CHECK\_XL\_AND\_THROW function is below.

```

//Checks for errors, if error found it throws error code and error description
inline
void FC_CHECK_XL_AND_THROW(const xloper * x, const char * function_name)
{
    if(XL_IS_ERROR(x))
    {
        char errorStr[1024];
        GetErrorString(x->val.err, 1023, errorStr);
        throw fincad::api_exception(x->val.err, errorStr, function_name);
    }
}

```

The FC\_CHECK\_XL\_AND\_THROW function calls the function GetErrorString internally. This GetErrorString routine is new in version 10. It takes an error code as an input and returns the error description that is associated with the given code. The GetErrorString function can also be used in C#, VB.NET and Java. Documentation for this new function is available on the error codes help page which can be accessed through the Function Finder menu (i.e. Help->Developer Reference->Error Codes).

The api\_exception class contains two additional member functions which I have not mentioned yet. The what() member function returns the exception string and the error\_code() member function returns the error code that is associated with the exception. Let us now look at a code example for one of the new C++ wrapper functions so that we can see how the FC\_CHECK\_XL\_AND\_THROW function is being used to create an exception when an internal function call returns an error.

```

// C++ wrapper for the function aaAMB
void AMB(double princ_m, double princ_v, double cpn, double d_m, double d_v, double yield,
FCSW13::type freq, FCSW98::type d_rul,
const fincad::Matrix & stat, const fincad::Matrix & hl, FCSW1::type acc, fincad::Matrix & result)
{
    LPXLOPER xlResult = aaAMB(princ_m, princ_v, cpn, d_m, d_v, yield, freq, d_rul,
const_cast<fincad::Matrix &>(stat), const_cast<fincad::Matrix &>(hl), acc);
    // check for errors and throw an exception (if an error was found)
    FC_CHECK_XL_AND_THROW(xlResult, "AMB");
    // throw an error in debug build if the result is not an array
    assert(XL_IS_ARRAY(xlResult));
    // need to take ownership of the memory away from the library
    // or the call to aaGlobalFreeAllA() will invalidate our result matrix
    result.take_ownership(xlResult);
    aaGlobalFreeAllA();
}

```

The result of the internal call (xlResult) is passed to the FC\_CHECK\_XL\_AND\_THROW function along with the function name. If the result of the function call is an error then the FC\_CHECK\_XL\_AND\_THROW function will throw an exception which will display the function name and the description text associated with the error code. The FINCAD math library has an internal garbage collection (i.e. memory management) scheme which frees memory objects used for calculating results during a function call. To prevent the wrapper from freeing the result for the function call, a call to a member function called take\_ownership() is used to remove the pointer for the results from the internal table used for garbage collection. The destructor member function for the Matrix class will handle the de-allocation of the memory when the results are no longer needed. The garbage collection algorithm

isn't used to do this for the return array in the new interface. This way you no longer need to call the `aaGlobalFreeAllA()` function explicitly like you did in the older interface which makes the new interface easier to use.

### Type Checking

As you may recall from the desired feature list at the beginning of this article, some developers who were writing their own interface for C++ were doing this in part so that they could define enumerated data types for switch inputs and allow the compiler to find some incorrect value assignments to input variables rather than having to find the problems at runtime. If you are not familiar with enumerated data types here is a definition from Wikipedia (a free online encyclopedia) "an enumerated type is a data type whose set of values is a finite list of identifiers chosen by the programmer. Enumerated types make program source more self-documenting than the use of explicit magic numbers. Typically the compiler will select a small integer to represent each enumeration value at run-time, but this representation is not always visible to the programmer." The sample C++ wrapper function above has several input parameters which are enumerated data types. These data types are defined in a new header file called `:"fcswitches.hpp"` which you can find in the `C:\Program Files\FINCAD\fcml10\include\fc` folder. There is a similar file called `FCSwitches.bas` that provides the same functionality for Visual Basic programmers.

Here is a short excerpt from the `fcswitches.hpp` header file

```
namespace switches{

struct FCSW1 {
    enum type {
        Act365Fixed = 1,
        Act360 = 2,
        Act365Act = 3,
        ISDA30360 = 4,
        ISMA30E360 = 5,
        Act365CDN = 6,
        ActAct = 7
    };
    static const type min;
    static const type max;
private:
    FCSW1();
};
```

Basically the enumerated values in the list are used much the same way as constants. An assignment of an enumerated element to a variable in C++ looks like below:

```
acc = FCSW1::Act360;
```

It is easier when looking at this code to see that the intended value being assigned is the one for the Actual 360 convention then it would be if the assignment looked like `"acc = 2;"`.

If you use the value 2 rather than `FCSW1::Act360` in an assignment you are using a magic number since it is a hard coded value which requires some additional knowledge to understand.

So, in summary, the new interface implements the four key features that were mentioned at the beginning of this article. It is no longer necessary to write your own custom wrapper to achieve the same result. You can now make your code easier to read by taking advantage of the enumerated data types that are available in v10 and you can throw meaningful exceptions in C++ which provide the error descriptions that are associated with the error codes. This means that you can rely on the math library for input validation rather than writing additional input validation code for each function you are using so that more detailed exception messages could be thrown. The new error lookup function used by the new C++ interface does this for you. These features significantly reduce the work required to integrate the FINCAD math library into your C++ applications.

### Disclaimer

Your use of the information in this article is at your own risk. The information in this article is provided on an "as is" basis and without any representation, obligation, or warranty from FINCAD of any kind, whether express or implied. We hope that such information will assist you, but it should not be used or relied upon as a substitute for your own independent research.

2006 © FinancialCAD Corporation. All rights reserved. FinancialCAD® and FINCAD® are registered trademarks of FinancialCAD Corporation. Other trademarks are the property of their respective holders. This email is for informational purposes only. FinancialCAD Corporation MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, IN THIS SUMMARY.