# ObjexxFCL

## EnergyPlus C++

Stuart G. Mentzer
Objexx Engineering, Inc.

# FCL = Fortran Compatibility Library

ObjexxFCL enables the C++ to retain the look and feel of the Fortran

- Fortran-ic arrays and strings
- Fortran intrinsic functions
- Fortran-style i/o

# ObjexxFCL 4.0

New 4.0 version with EnergyPlus

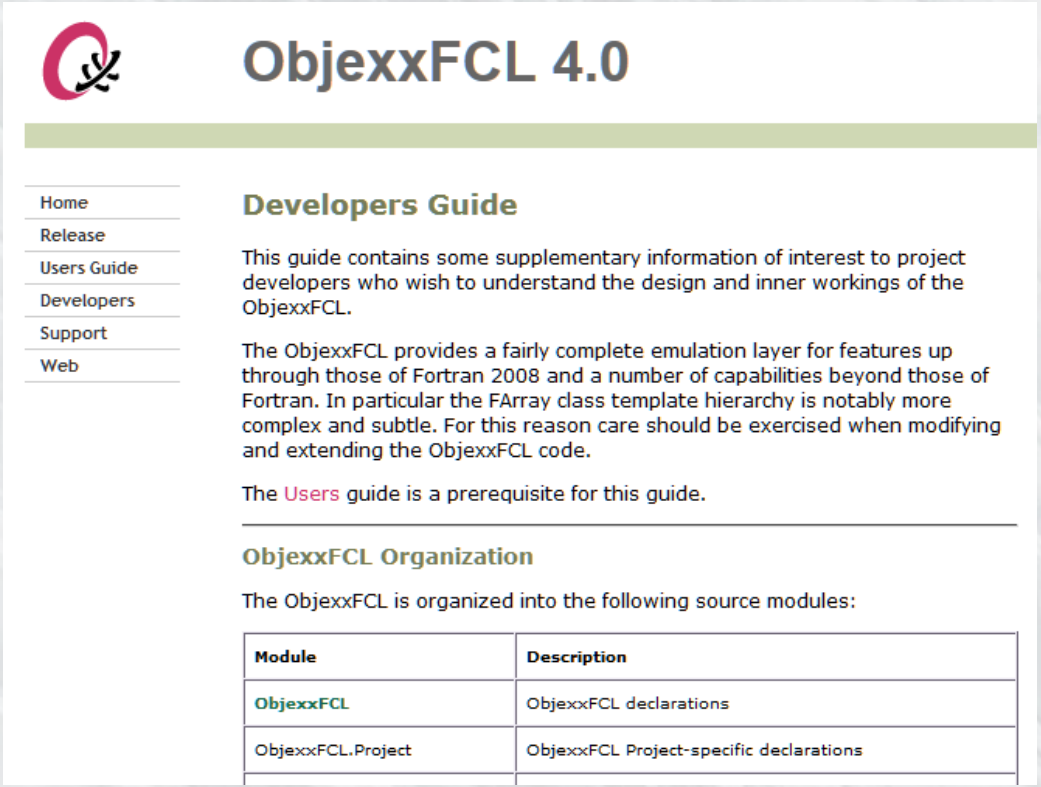
Expanded and improved Fortran 90+ support

- Array slices/sections
- Member array usage
- More intrinsic functions
- New higher level i/o support
- Global i/o stream “units”

# Getting Started

Documentation →

Browse EnergyPlus



ObjexxFCL 4.0

Home
Release
Users Guide
Developers
Support
Web

## Developers Guide

This guide contains some supplementary information of interest to project developers who wish to understand the design and inner workings of the ObjexxFCL.

The ObjexxFCL provides a fairly complete emulation layer for features up through those of Fortran 2008 and a number of capabilities beyond those of Fortran. In particular the FArray class template hierarchy is notably more complex and subtle. For this reason care should be exercised when modifying and extending the ObjexxFCL code.

The Users guide is a prerequisite for this guide.

### ObjexxFCL Organization

The ObjexxFCL is organized into the following source modules:

| Module | Description |
|---|---|
| ObjexxFCL | ObjexxFCL declarations |
| ObjexxFCL.Project | ObjexxFCL Project-specific declarations |

# Arrays: Fortran

Fortran arrays:

- Multidimensional
- Column-major memory order
- Array index range flexibility
- Array slicing (sections)
- Member array usage
- F77-style array passing tricks
- Fast!

```
REAL :: A(3,3), B(3,3)

A = B  ! Whole array op's

A(i,j) = 42.0d0 !
Elements

A(i,:) = B(k,:) ! Slices
```

# Arrays: C++

C++ has no multidimensional array type

- `A[N][N]` or vector<vector> are not sufficient
- Need an array library for engineering applications
- Boost.MultiArray, Blitz++, uBLAS, …
- Each has different attributes and strengths
- None are drop-ins for Fortran arrays

# Arrays: ObjexxFCL

ObjexxFCL FArray:

- Multidimensional
- Column-major order
- Array index range flexibility
- Array slicing (sections)
- Member array usage
- F77-style array passing tricks
- Fast and tuneable to very fast
- Drop-in replacement for Fortran arrays

```
FArray2D_double A(3,3);
FArray2D_double B(3,3);

A = B;  // Whole array op's

A(i,j) = 42.0; // Elements

A(i,_) = B(k,_); // Slices
```

# Arrays: ObjexxFCL: Types

| Type | Description | Rank |
|---|---|---|
| FArray*N*D | Contiguous array | *N* |
| FArray*N*A | Argument (proxy) array of contiguous array of any Rank | *N* |
| FArray*N*P | Proxy array of contiguous array of any Rank | *N* |
| FArray*N*S | Rank *N* slice of contiguous or slice array of Rank ≥ *N* | *N* |
| MArray*N* | Member array: Proxy of members of objects in an array | *N* |

* These are templates: `FArray3D< int > M( 3, 3, 3 );`

# Arrays: Beyond Fortran

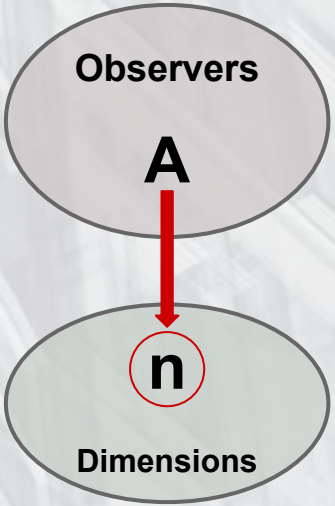
FArray:

- All arrays are allocatable
- Automatic sizing via Dimension expressions
- All lookups are index range checked in debug build
- Constructor variety to simplify create + initialize
- Data-preserving resize operations
- Raw memory access for use with numeric libraries

# Automatic Array Sizing

```
Dimension n; // Don't know n yet
FArray2D_float A( n, n );
…
n = 100; // n calls A.update()
// A's dimensions are all set
// A is allocated to 100 x 100

// Dimension expressions too
FArray1D_float sorted( n * n );
```



# Array Passing: C++ Style

```
FArray2D_int J( 3, m );
foo( J ); // Pass whole array

void foo( FArray2_int & I ) // Pass by reference
{...}
```

Passing same rank/dimension arrays => Fast

FArray2 base => Can pass var. 2D array types

# Array Passing: Fortran 77 Style

```
FArray2D_int J( 3, m );
foo( J( 1, n ) ); // Pass element at start of column

void foo( FArray1A_int col ) // Create proxy array
{
   col.dim( 3 ); // Set dimensions of array “view”
   ...
}
```

F2C++: Argument Arrays used for **explicit shape** and **assumed size** arrays
They are contiguous array proxies

# Array Passing: Fortran 90+ Style

```
FArray2D_int J( 3, m );
foo( J ); // Pass whole array

void foo( FArray2S_int I ) // Creates proxy
{ // I is “assumed size” here
   ...
}
```

F2C++: Slice Arrays used for **assumed shape** arrays
They can accept contiguous and slice arrays

# Strings: Fstring

Fstring has Fortran string semantics

- Fixed length
- Comparisons ignore trailing space
- Live (proxy) substrings: name( 5, 12 )
- Intrinsics: len, len_trim, trim, index, ...

Interoperates with std::string and “C strings”

Most uses best migrated to std::string

# Fortran I/O Support Coverage

Global unit-based i/o: open, read, write, print, inquire, backspace, rewind, close

Formatted & list-directed i/o

Full FORMAT semantics

Internal read/write

* Issue with line termination yet to be solved

# I/O: Formatting

Fortran format syntax supported at a high level

Unit-based i/o: Fortran + C++ stream flavor:

```
gio::write( unit, "(F15.7)" ) << x;
```

Non-global lower-level i/o system also available:

```
stream << F( x, 15, 7 ) << std::endl;
```

Can mix this with native C++, Boost.Format, …

# I/O: Controls

`IOFlags` objects used for control, inquire, status, and error flags

```
IOFlags flags;
gio::inquire( "in.epw", flags );
WeatherFileExists = flags.exists();
IOStatus = flags.ios();
```

# Other Parts Used in EnergyPlus

Intrinsic functions

Optional arguments

Reference (POINTER)

Character and String helper functions

Bit operations

# Other Parts *Not* Used in EnergyPlus

byte types

Safe C-style arrays and strings

ChunkVector (for huge vectors)

Required argument wrapper

# Intrinsic Functions

In various ObjexxFCL headers:

- Associated class header
- Time_Date.hh
- random.hh
- numeric.hh
- FArray.functions.hh
- FArrayS.functions.hh
- ...

# Optional Arguments

Fortran OPTIONAL ≠ C++ default arguments:

- Test for presence
- No default value
- Don’t need to be at end of arguments

EnergyPlus uses OPTIONAL heavily:

- Doesn’t always check PRESENT
- Optional catches these bugs in debug builds

# Optional Arguments: Declarations

```
void
SetVarSpeedCoilData(
    int const WSHPNum,
    bool & ErrorsFound,
    Optional_int CompanionCoolingCoilNum = _,
    Optional_int CompanionHeatingCoilNum = _
);
```

Default values needed in C++ to flag “not present” =>
Optional arguments must be moved to end of arg list
Arg reordering done for a number of EnergyPlus functions

# Optional Without Reordering

```
void
foo(
    int const WSHPNum,
    Optional_int CoilNum = _,
    bool & Errors
);
```

← Not legal

But this is →

```
void
foo(
    int const WSHPNum,
    Optional_int CoilNum = _,
    Required_bool Errors = _
);
```

Required looks like Optional but requires arg to be present

- Checked in debug builds

Not pretty / Not used in EnergyPlus / Could be useful

# Optional Arguments: Usage

```
void
SetVarSpeedCoilData(
    int const WSHPNum,
    bool & ErrorsFound,
    Optional_int CompanionCoolingCoilNum,
    Optional_int CompanionHeatingCoilNum
)
{ ...
    if ( present( CompanionHeatingCoilNum ) ) {
        // Use CompanionHeatingCoilNum
    }
}
```

# Optional: Technical Details

Optional is a template wrapper proxy class

Proxy => Optional created during call so:

- Pass by value
- Small performance cost

Optional converts to its value when needed

Some usage needs explicit conversion:

```
opt()( args )
opt().member
```

# Reference (Fortran POINTER)

Fortran POINTER ≠ C++ reference or pointer:

- Reattachable unlike C++ reference
- Reference syntax unlike C++ pointer

```
struct IntegerVariables
{
    Reference_int Which;
    …
}
```

# Reference: Usage

```
Reference_int IntValue; // Declaration
IntValue >>= iValue; // Attach
IntValue >>= jValue; // Reattach
IntValue.attach( kValue ); // Reattach
IntValue.detach(); // Detach
```

# Reference: Notes

Like Fortran POINTER, Reference is “leaky”:

- Directly allocated memory not auto-deleted
- EnergyPlus like most code just leaks these

Proxy like Optional:

- Auto-converts to contained object
- Need explicit conversion in some usage: ref().member

Suggest mig. to C++ reference or smart pointer

# Performance: Arrays

FArrays are designed for high performance

- Lookups: inline, non-virtual, highly optimized methods
- Linear indexing support for hot spot tuning
- Internal operations done with linear indexing
- Hidden array copying avoided
- Slice arrays drill into memory for high speed

# Performance: Arrays: Linear Indexing

```
for ( int k = 1; k <= N; ++k )
   for ( int j = 1; j <= N; ++j )
      for ( int i = 1; i <= N; ++i )
         A(i,j,k) *= fac(i);
```

Multi-index lookup is slow

A bunch of adds and multiplies to compute memory offset

```
for ( int k = 1, l = 0; k <= N; ++k )
   for ( int j = 1; j <= N; ++j )
      for ( int i = 1; i <= N; ++i, ++l )
         A[l] *= fac(i);
```

Element access is usu. regular

Linear indexing is easy & fast

(This is what a good Fortran optimizer is doing)

# Performance: Core Non-Array Code

Fstring has good performance

Intrinsic function performance should be good

Formatted i/o is not fast:

- C++ streams are notoriously slow
- Emulation of Fortran FORMAT makes it slower
- Can replace by C-style i/o in i/o bound hot spots

# Performance: Fortran Emulation

Emulating Fortran POINTER, OPTIONAL, and array passing tricks has mild overhead

Usually only significant in hot spots and high call count functions where we can tune it away

# Questions