# Post-Conversion Migration

## EnergyPlus C++

Stuart G. Mentzer
Objexx Engineering, Inc.

# Strings

Fortran strings (and Fstring):

- Fixed length
- Comparisons ignore trailing space

Migration to C++ std::string is usually best

Can do gradually but ...

All at once is less work and disruption

- Plan is to do this soon

# Reference (POINTER)

Fortran POINTER and ObjexxFCL Reference:

- Reattachable unlike C++ references
- Leaky by design

Migration to C++ references and smart pointers

Fix leaks in remaining uses

# Optional Arguments

Fortran OPTIONAL argument semantics differ from C++ arguments with default values:

- Can omit argument and ask if present
- Can be intermixed with non-OPTIONAL args (keyword)

ObjexxFCL Optional:

- Wrapper template with default “omitted” value
- Use _ to denote omitted argument in calls

# Optional Arguments: Fortran

```
REAL FUNCTION add( a, b, c )
  REAL :: a
  REAL, OPTIONAL :: b
  REAL :: c
  IF ( PRESENT( b ) ) THEN
    add = a + b + c
  ELSE
    add = a + c
  END IF
END FUNCTION
...
  x = add( w, c = y )
```

# Optional Arguments: C++

```
float add( float a, Optional_float b, float c )
{
  if ( present( b ) ) {
    return a + b + c;
  } else {
    return a + c;
  }
}
...
  x = add( w, _, y );
```

# Optional Arguments: C++ Migration

Ugly with many optionals: `foo( a, _, _, _, z )`

Performance hit to create Optional wrappers

Migration to:

- Overloaded functions
- Default arguments

OO code will tend to reduce the # args

# Member Arrays

Members of objects in arrays treated like arrays: `array.member(i)`

ObjexxFCL has mechanisms for this:

```
array.ma( &Type::member )( i )
```

Custom array methods: `array.member()(i)`

Not pretty (or performance friendly)

Best avoided and migrated out

# Naming

C++ identifiers are case sensitive

C++ tends to use ClassName and object_name

EnergyPlus: CamelCase for types & objects

Naming guideline?

Refactoring editor can safely rename

# SELECT CASE Mapping

Fortran SELECT CASE semantics differ from C++ **switch** so they convert to messy **if** blocks

```
{ auto const SELECT_CASE_var( UpdateType );
if ( SELECT_CASE_var == iGetZoneSetPoints ) {
   CalcZoneAirTempSetPoints();
} else if ( SELECT_CASE_var == iPredictStep ) {
   PredictSystemLoads( ShortenTimeStepSys, ... );
}}
```

# SELECT CASE Mapping: After

Fortran SELECT CASE semantics differ from C++ **`switch`** so they convert to messy **`if`** blocks

```
if ( UpdateType == iGetZoneSetPoints ) {
   CalcZoneAirTempSetPoints();
} else if ( UpdateType == iPredictStep ) {
   PredictSystemLoads( ShortenTimeStepSys, ... );
}
```

# I/O Flags

IOFlags usage can be simplified:

```
{ IOFlags flags; flags.ADVANCE( "No" );
gio::write( unit, fmt, flags ); }
```

to:

```
IOFlags flags_na; flags_na.na_on();
…
gio::write( unit, fmt, flags_na ); // Reuse flags
```

# Comments

Commented out code is not converted to C++

```
//  DO Loop2=1, NumOfNodeConnections
//    IsValid=.FALSE.
//    IF (Loop1 == Loop2) CYCLE
//    IF (IsInlet .OR. IsOutlet) EXIT
//  ENDDO
```

Non-code comments use Fortran terminology (DO, …)

Policy to clean up comments as you go

# Tools to Goose Up C++

Safer vectors

Intrusive smart pointers for better speed

Key-indexed vectors

# Safer Vectors

`std::vector` doesn't bounds check `v[i]`

- VC++ does by default in debug builds
- Clang and GCC have a mechanism to do it

Solution: wrap `std::vector` and do it yourself:

`vector0`: Bounds checks in debug build

`vector1`: Same but 1-based indexing

# Intrusive Smart Pointers

## Faster and safer than shared_ptr/weak_ptr

- Like Boost intrusive_ptr (but a little faster)

## Use them in class hierarchies

```
template< typename T >
class owning_ptr
{
   …
private:
   T * p_;
};
```

```
class
ReferenceCount
{
   …
private:
   mutable Size
count_;
};
```

```
class
Pointee :
public ReferenceCount
{
   …
};
```

# Smart Pointer Policy

Do you want them? Probably: Need a policy

- std::shared_ptr  Shared ownership
- std::weak_ptr  Non-owning / Can't delete
- std::unique_ptr  One owner

Smart pointers add some overhead

Classes that use RAII and don't expose pointers probably don't need smart pointers

# Code Refinement Tools

## clang-format / astyle / Uncrustify

- Don’t need yet: C++ is clean for now
- Wrap lines? Or let IDEs soft wrap?

## clang-modernize: C++11 migration

- Need to see if too modern for VC++ or other compilers
- Will be packaged in Ubuntu 14.04 (Trusty)

## Clang-Tidy

## iwyu (include-what-you-use)

# clang-modernize: for Loops

converts:

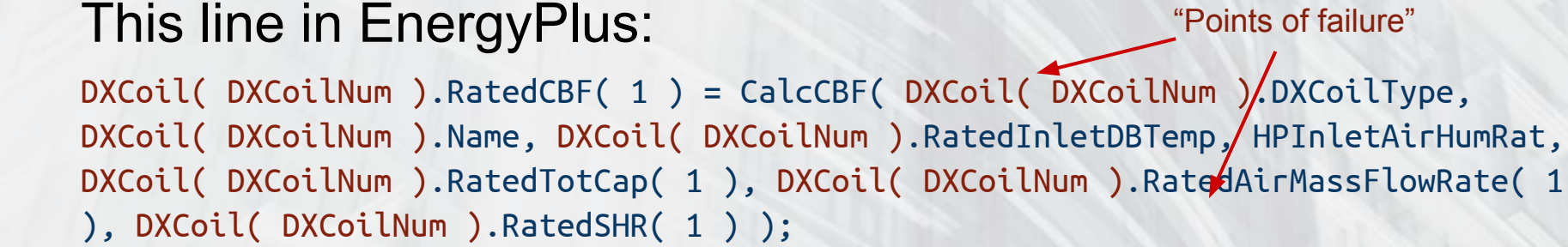
```
for ( vector< MyType >::iterator it = v.begin();
 it != v.end(); ++it )
  cout << *it;
```

to:

```
for ( auto & elem : v )
  cout << elem;
```

# Refactoring for Clarity, Safety, & Speed

This line in EnergyPlus:




```
DXCoil( DXCoilNum ).RatedCBF( 1 ) = CalcCBF( DXCoil( DXCoilNum ).DXCoilType,
DXCoil( DXCoilNum ).Name, DXCoil( DXCoilNum ).RatedInletDBTemp, HPInletAirHumRat,
DXCoil( DXCoilNum ).RatedTotCap( 1 ), DXCoil( DXCoilNum ).RatedAirMassFlowRate( 1
), DXCoil( DXCoilNum ).RatedSHR( 1 ) );
```

Can be refactored to these 2 lines:

```
auto & coil( DXCoil( DXCoilNum ) ); // Reuse this throughout the block

coil.RatedCBF( 1 ) = CalcCBF( coil.DXCoilType, coil.Name, coil.RatedInletDBTemp,
HPInletAirHumRat, coil.RatedTotCap( 1 ), coil.RatedAirMassFlowRate( 1 ),
coil.RatedSHR( 1 ) );
```

Redundancy: Clutter, bug risk, & (sometimes) slow

# Refactoring for Clarity, Safety, & Speed

Unless you have a very smart optimizer this style is definitely a performance waste (as well as clutter and bug risk):

```
if ( ( PlantLoop( LoopNum ).LoopSide( LoopSideNum ).
Branch( BranchNum ).Comp( CompCounter ).TypeOf_Num
 != TypeOf_PumpVariableSpeed ) &&
( PlantLoop( LoopNum ).LoopSide( LoopSideNum ).Branch(
BranchNum ).Comp( CompCounter ).TypeOf_Num
 != TypeOf_PumpBankVariableSpeed ) ) {
```

# QA

Address all `//Autodesk` comments & tickets

Eliminate all compiler warnings

Assert pre/post-conditions and invariants

Unit testing: Googletest

Static testing: clang++/clang-analyzer + ?

# Questions