Nuts and Bolts KIM-API

A guide to KIM-API portable models for the beginners, from a beginner!

C++ Primer

Common words you might encounter in KIM-API source

1. Reinterpret_cast
2. Template
3. Extern “C”
4. Static
5. PIMPL design

C++ Primer

Prerequisites:

1. Unix like env (preferably LINUX)
2. `g++` compiler
3. Cmake
4. Recommended: VS code + remote plugin
5. https://github.com/ipcamit/kim-api-tutorial

Or use KDP + VSCode

```

[sudo] docker pull ghcr.io/openkim/developer-platform:latest-minimal

docker run -it --name kim_dev -v `pwd`:/home/openkim/tutorial  ghcr.io/openkim/developer-platform:latest-minimal bash

```

“Attach to running container”

```
cd tutorial
```

“Just shut up and calculate”

🧮 **“Shut up and calculate.”**
— Richard Feynman
Just make it work. Don't get lost in theory when the compiler is yelling at you.

🛠️ **“If you keep unpacking everything, you’ll never get anything done... But if you don’t, you’ll do the wrong thing.”**
— Jim Keller

reinterpret_cast - I know what I am doing

What it does: low-level reinterpretation of the “bit pattern”

When to use (rarely & carefully):

- Interfacing with low-level hardware/memory.
- Converting between unrelated pointer types (use with extreme caution).
- Storing pointer addresses as integers (platform-dependent).

Why it's dangerous: No compile-time or runtime checks.

KIM-API use: convert objects to void for C-like uniform interface

```cpp
#include <iostream>

struct Data { int a; double b; };

int main() {
 Data myData = {10, 3.14};
 // Treat the Data object's memory as raw bytes
 char* bytePtr = reinterpret_cast<char*>(&myData);

 std::cout << "First few bytes of Data object:\n";
 for (int i = 0; i < sizeof(Data); ++i) {
   std::cout << std::hex << (int)(unsigned char)bytePtr[i] << " ";
 }
 std::cout << std::dec << std::endl;

 // DANGEROUS: Converting unrelated pointer types
 long addr = reinterpret_cast<long>(&myData);
 std::cout << "Address stored as long: " << addr << std::endl;

 return 0;
}
```

Templates - Ask compiler to write code for you

Function Templates: Create functions that work with various types

Compiler substitutes appropriate values at runtime

KIM-API uses: Dead code removal, optimization

```cpp
#include <iostream>
#include <string>

// Function template to find the maximum of two values
template <typename T> // "T" is a placeholder for any type
T maximum(T a, T b) {
 return (a > b) ? a : b;
}

int main() {
 std::cout << "Max(5, 10): " << maximum(5, 10) << std::endl; // T is int
 std::cout << "Max(3.14, 2.71): " << maximum(3.14, 2.71) << std::endl; // T is double
 std::cout << "Max('a', 'z'): " << maximum('a', 'z') << std::endl; // T is char
 // std::cout << "Max(5, 3.14): " << maximum(5, 3.14); // Error! T cannot be both int and double
 return 0;
}
```

Dead code removal

```
switch (GetComputeIndex(isComputeProcess_dEdr,
                       isComputeProcess_d2Edr2,
                       isComputeParticleVirial,
                       isShift))
{
 case 0:
   ier = Compute<false, false, false, false, false, false, false, false>(
 case 1:
   ier = Compute<false, false, false, false, false, false, false, true>(
 case 2:
   ier = Compute<false, false, false, false, false, false, true, false>(
…
```

Compute Dispatch: A templated Compute function, with all branch conditions.

Compiler generates a giant function without conditions.

extern “C”: Universal language compatibility

C++ compilers change function names for overloading, namespaces,and vice-versa) by name.

extern "C" ensures that names are left untouched

Tells the C++ compiler: "Use the C language's convention for naming (no mangling) for the following function(s) or variable(s)".

Purpose: Enables interoperability between C++ and C code.

KIM-API maps interface functions under extern “C” so that it can be called form C, Fortran

```cpp
// g++ -c extern.cpp -o extern.o
// nm extern.o

// Will be mangled by C++ compiler
void process_data(int data) {
   // Dummy implementation
   volatile int x = data;
}

// Overloaded version - will have a DIFFERENT mangled name
void process_data(double data) {
   // Dummy implementation
   volatile double y = data;
}

// *** Using extern "C" ***
// Tells C++ compiler NOT to mangle this name
extern "C" void process_data_for_c(int data) {
   // Dummy implementation
   volatile int z = data; //ignore volatile
}
```

Static: keep only one copy

1. Inside a Class: static Member Variable: Shared by all objects (one copy per class).

Python equivalent: @staticmethod

2. Inside a Function (Local Variable): Initialized only once. Value persists across calls. Lives for program duration.

KIM-API : Uses static to create a function map

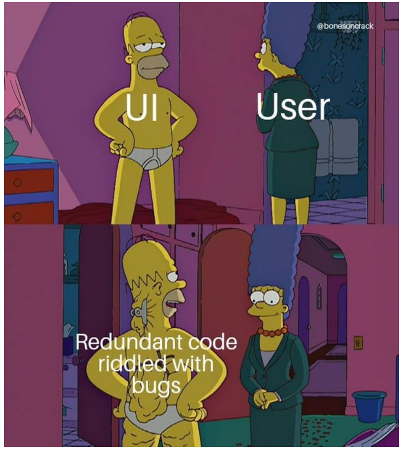
```cpp
#include <iostream>

class Thing {
public:
   // 1. Static Member Variable (Declaration)
   static int count;

   Thing() {
       ++count; std::cout << "Thing created! (Total things: " << count << ")" << std::endl;
   }
   ~Thing() {
       --count; std::cout << "Thing destroyed! (Total things: " << count << ")" << std::endl;
   }

   // 2. Static Member Function
   static int howMany() {
       return count; // Accesses the static 'count'
   }
};

// Definition and Initialization of the static member variable outside the class
int Thing::count = 0;

int main() {
   std::cout << "Initial count: " << Thing::howMany() << std::endl; // Call static function
   Thing t1; Thing t2;
   std::cout << "Current count: " << Thing::howMany() << std::endl;
   {Thing t3; // Create third object in a limited scope
    std::cout << "Count inside scope: " << Thing::howMany() << std::endl;} // t3 is destroyed here, destructor runs
   std::cout << "Count after scope: " << Thing::count << std::endl; // Can also access directly (if public)
   return 0;
}
```

PIMPL Pattern: Pointer to Implementation

A C++ technique to hide the private data members and internal workings of a class.

- Minimal public interface, single pointer to another, hidden "implementation".

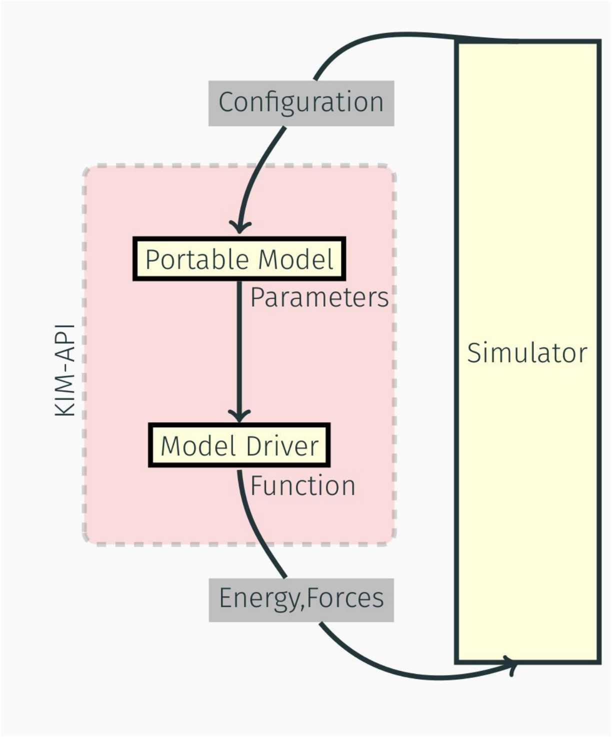
Hidden class: all the real private data and helper methods.



Quick Recap From Last Week …

Walkthrough a model driver

- Combination of Implementation + Parameters
- Implementation = Driver
- Parameters = Model
- Simulator IO = Compute Arguments



reinterpret_cast - I know what I am doing

What it does: low-level reinterpretation of the “bit pattern”

When to use (rarely & carefully):

- Interfacing with low-level hardware/memory.
- Converting between unrelated pointer types (use with extreme caution).
- Storing pointer addresses as integers (platform-dependent).

Why it's dangerous: No compile-time or runtime checks.

KIM-API use: convert objects to void for C-like uniform interface

Static: keep only one copy

1. Inside a Class: static Member Variable: Shared by all objects (one copy per class).

Python equivalent: @staticmethod . No self, this.

CppClass::func(x) == CppClass::func(this, x)

static CppClass::func(x) == CppClass::func(x)

2. Inside a Function (Local Variable): Initialized only once. Value persists across calls. Lives for program duration.

KIM-API : Uses static to create a function map

Adapter Pattern

Adapter Pattern in 3 steps

- Simulator calls Class::Compute (C signature).
- Compute fetches this from GetModelBufferPointer.
- Forwards to this->computeImpl, where the real work happens.

```
static int Destroy(KIM::ModelDestroy * const modelDestroy);
static int Refresh(KIM::ModelRefresh * const modelRefresh);
static int
 Compute(KIM::ModelCompute const * const modelCompute,
         KIM::ModelComputeArguments const * const modelComputeArguments);
```

Our Minimal LJ

Initialize the model:

1. Set Model Numbering
2. Set Model units
3. Set Influence Distance
4. Set Neighbor List
5. Set Species Codes
6. Register Functions
7. Read and Register Parameters

Our Minimal LJ

Initialize the model:

1. Set Model Numbering

```
modelDriverCreate->SetModelNumbering(KIM::NUMBERING::zeroBased);
```

2. Set Model units

```
modelDriverCreate->SetUnits(KIM::LENGTH_UNIT::A,
                            KIM::ENERGY_UNIT::eV,
                            KIM::CHARGE_UNIT::unused,
                            KIM::TEMPERATURE_UNIT::unused,
                            KIM::TIME_UNIT::unused);
```

3. Set Influence Distance
4. Set Neighbor List

```
modelDriverCreate->SetInfluenceDistancePointer(&cutoff);
modelDriverCreate->SetNeighborListPointers(1, &cutoff,
                           &modelWillNotReqNeighNoncontribPart);
```

5. Set Species Codes

```
KIM::SpeciesName KIMSpeciesCode(species.c_str());
modelDriverCreate->SetSpeciesCode(KIMSpeciesCode, 0);
```

6. Register Functions

```
KIM::ModelComputeFunction * compute = LennardJones612::Compute;
modelDriverCreate->SetRoutinePointer(KIM::MODEL_ROUTINE_NAME::Compute,
                                    KIM::LANGUAGE_NAME::cpp, true,
                           reinterpret_cast<KIM::Function*>(compute));
```

7. Read and Register Parameters

```
modelDriverCreate->GetParameterFileDirectoryName(&parmFileDir);
modelDriverCreate->GetParameterFileBasename(0, &paramFileName);
modelDriverCreate->GetNumberOfParameterFiles(&numberOfParamFiles);
```

Compute

To compute energies and forces

1. Receive compute arguments from the simulator
2. Validate which arguments are not NULL
3. Calculate appropriate quantities and assign the values back

```
modelComputeArguments->GetArgumentPointer(
        KIM::COMPUTE_ARGUMENT_NAME::partialEnergy, &energy);
```

```
Options:
KIM::COMPUTE_ARGUMENT_NAME::numberOfParticles
KIM::COMPUTE_ARGUMENT_NAME::particleSpeciesCodes
KIM::COMPUTE_ARGUMENT_NAME::particleContributing
KIM::COMPUTE_ARGUMENT_NAME::coordinates

KIM::COMPUTE_ARGUMENT_NAME::partialEnergy
KIM::COMPUTE_ARGUMENT_NAME::partialParticleEnergy
KIM::COMPUTE_ARGUMENT_NAME::partialForces
```

Things to remember when looking at MD code

1. PIMPL: All the actual code is in ModelDriverImplementation class
2. Compute dispatch: Compute function is usually in ModelDriverImplementation.hpp file (templated).
3. Need to compile a debug version of KIM-API for debugging
4. Consult kim.log in case of error

Questions?