Deletion of physics process, model and cross section objects

KOI, Tatsumi

U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Construction of objects of physics process, model and cross section happens in

- PhysicsList
  - new and register
- Process, model and cross section
  - new and use
- Others including users codes

These objects must be deleted

- by **who** and at **when**

Some of them are shared

- Introduce complexity

Multithreading library

- enhanced potential concerns

An example
Geant4/physics_lists/lists/include/LBE.icc

SLAC

```
779     else if (particleName == "neutron") {
780       // elastic scattering
781       G4HadronElasticProcess* theElasticProcess = new
G4HadronElasticProcess;
782       theElasticProcess-
>AddDataSet(G4CrossSectionDataSetRegistry::Instance()-
>GetCrossSection
DataSet(G4ChipsNeutronElasticXS::Default_Name()));
783       G4HadronElastic* elastic_neutronChipsModel = new
G4ChipsElasticModel();
784       elastic_neutronChipsModel-
>SetMinEnergy( 19.0*CLHEP::MeV );
785       theElasticProcess-
>RegisterMe( elastic_neutronChipsModel );
786       G4NeutronHPElastic * theElasticNeutronHP =
new G4NeutronHPElastic;
787       theElasticNeutronHP->SetMinEnergy( theHPMin );
788       theElasticNeutronHP->SetMaxEnergy( theHPMax );
789       theElasticProcess-
>RegisterMe( theElasticNeutronHP );
790       theElasticProcess->AddDataSet( new
G4NeutronHPElasticData );
791       pmanager-
>AddDiscreteProcess( theElasticProcess );
```

A question,

who should delete

“theElasticNeutronHP”,

which is instantiated in

LBE::ConstructHad()

and when

The object is newed in LBE::ConstructHad()

registered theElasticProcess

The process is added (registered) to

ProcessManager of Neutron (G4Neutron::Neutron)

G4HadronicInteractionRegistry::Clean()

SLAC

```
void G4HadronicInteractionRegistry::Clean()
{
  size_t nModels = allModels.size();
  //std::cout << "G4HadronicInteractionRegistry::Clean() start " <<
nModels
  //        << " " << this << std::endl;
  for (size_t i=0; i<nModels; ++i) {
    if( allModels[i] ) {
      const char* xxx = (allModels[i]->GetModelName()).c_str();
      G4int len = (allModels[i]->GetModelName()).length();
      len = std::min(len, 9);
      const G4String mname = G4String(xxx, len);
      //std::cout << "G4HadronicInteractionRegistry: delete " << i << "  "
      //                << allModels[i] << " " << mname
      //                << " " << this << std::endl;
      if(mname != "NeutronHP") {
        delete allModels[i];
      }
      // std::cout << "done " << this << std::endl;
    }
  }
  allModels.clear();
  //std::cout <<"G4HadronicInteractionRegistry::Clean() is done
"<<std::endl;
}
```

Currently, it suppose to delete in "G4HadronicInteractionRegistry::Clean()"

However, the object (theElasticNeutronHP) will not delete in the method,,,,

But this is another story

Is this design reasonable?

Lifetime of object

- step
- track
- event
- run
- application

Boundary sometimes becomes unclear

Changing PhysicsList between run

- Mike pointed out problem in his Tuesday presentation

How to control deletion of instantiated object

Establish a rule and observe the rule

- the best way but unrealistic

Use reference of pointer instead copy of pointer

Introduce registry(ies) to manage them *Current situation*

- number of registries
- timing of registration and deletion
- break ideal control based on ownership

It may be better to separate issues

Deletion of object

- ownership
- lifetime

Deletion of **shared** object

- mechanism (for example introducing registries)
- ownership and lifetime of the mechanism

Deletion of object (and shared object) in multithreading library

- technical problem

Deletion in multithreading library

Multithreading library enhances complexity

Destructor may need to work differently between master and worker

G4Threading::IsWorkerThread() does not always work

This is not a ownership problem

Smart pointers in C++11

SLAC

unique_ptr<T>

A unique_ptr explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the std::move function can be used to transfer ownership of the contained pointer to another unique_ptr.

shared_ptr<T>

A shared_ptr maintains reference-counted ownership of its contained pointer in cooperation with all copies of the shared_ptr. The object referenced by the contained raw pointer will be destroyed when and only when all copies of the shared_ptr have been destroyed.

weak_ptr<T>

A weak_ptr is created as a copy of a shared_ptr. The existence or destruction of weak_ptr copies of a shared_ptr have no effect on the shared_ptr or its other copies. After all copies of a shared_ptr have been destroyed, all weak_ptr copies become empty.

auto_ptr<T> is deprecated in C++11

Consider to use this kind of supports from compiler

Use them smartly, otherwise introduce other problems