# Exercise Session 10

January 18, 2013

We would like to intragrate in the program `AllDynamic` the possibility to handle also the integrands with a factory. In particular we would like to collect some dynamic libraries with different integrands, using both classes functor and free functions. The number of the libraries is unknown and their loading still have to be dynamic. The chosen of the integrand is done in the data file, handled by GetPot, using the key associated in the registration process. Each library has to be independent from the others one and has to resister autonomously their products. In this way the adding of a new library is just a modification in the data file, the main program has not to be recompiled.

Modifications on the class **class** Factory are not allowed. In the sequel there are present some useful points that might be followed. Beware that each modification of the code might also affect the `Makefile`.

1. Do not use the class **class** udfHandler, using the new factory instead.
2. Using the function wrapper std :: function , of the standard library, to uniformly register the integrands into the factory.
3. Create a new proxy class for the registration of the integrands. Discuss why it is useful to split the builder function from the class proxy.
4. Remove the **extern** "C" from the header files containing the free functions. Create a new file, or use the corresponding source file, for the registration using the proxy previously introduced.
5. Introduce also a class functor, in a separate library, and use it.

To increase the robustness of using the quadrature rules with the factory, statically check if the template parameter ConcreteProduct inherits from AbstractProduct_type, defined into the factory. Furthermore the static member Build has to be converted in the corresponding type defined into the factory. Use the std :: traits and the static_assert for this purpose. Statically checked means that these features have to be checked at compile time.

**How to use the program** The structure of the folder `exercice` is the following.

```
.
‘-- src
    |-- QuadratureRule
    |   |-- AllDynamic
    |   |-- baseVersion
    |   ‘-- generic_factory
    ‘-- Utilities
        ‘-- include
```

To use the programm edit the `Makefile.inc` in the folder `exercice` specifying the absolute path of the folder `exercice` in the variable `PACS_ROOT`. Moreover beware the location of your compiler, specify it in the variable `CXX`. Create two empty folder called `include` and `lib` at the same level of `src`. Go in the directory `src/Utilities` and type `make install`. Now go in the directory `src/QuadratureRule/baseVersion` and type `make dynamic; make install`. In the directory `src/QuadratureRule/generic_factory` type `make install`. Go in the directory `src/QuadratureRule/AllDynamic` and type `make dynamic; make exec`. Before running the program set the `LD_LIBRARY_PATH`. You can use the command `make` to suggest the correct path for the command.

## Solution

The **class** Proxy, contained in `generic_factory/Proxy.hpp`, is written when the factory handles different types. But this case all the types are equal to std :: function<**double** (**double const** &)>, since we are using the function wrapper to handle free functions, lambda functions and functor classes. The new proxy is contained in the file `generic_factory/ProxyFunction.hpp` and reported below.

```
#ifndef PROXYFUNCTION_HPP_
#define PROXYFUNCTION_HPP_
#include <string>
#include <memory>
#include <type_traits>
#include "QuadratureRule.hpp"
#include "Factory.hpp"

namespace FunctionsFactory
{

  class BuilderFun {

  public:
    typedef NumericalIntegration::FunPoint FunPoint;

    BuilderFun ( FunPoint const & fun ) : M_fun( fun ) {}

    std::unique_ptr<FunPoint> operator () ( ) const
    {
        return std::unique_ptr<FunPoint>( new FunPoint (M_fun) );
    }

  private:

    FunPoint M_fun;

  };


  /*! A simple proxy for registering into a factory.

    It provides the builder as static method
    and the automatic registration mechanism.

    */
  class ProxyFunction
  {
  public:

    typedef NumericalIntegration::FunPoint FunPoint;

    typedef GenericFactory::Factory<FunPoint, std::string, BuilderFun> Factory_type;

    //! The constructor does the registration.
    ProxyFunction(std::string const & name, FunPoint const & fun)
    {
      // get the factory. First time creates it.
```

```
      Factory_type & factory(Factory_type::Instance());

      BuilderFun builder( fun );

      // Insert the builder. The & is not needed.
      factory.add(name, builder);
    }

  private:
    ProxyFunction(ProxyFunction const &)=delete; // only C++11
    ProxyFunction & operator=(ProxyFunction const &)=delete; // only C++11
  };

    }

#endif /* PROXYFUNCTION_HPP_ */
```

We have discorporated the **static** method Build from the proxy in a new class. In fact the proxy used for the quadrature rules can link the type, via the template parameter ConcreteProduct. The new builder stores the function so, when the **operator** () is called, it can return the function. For simplicity we have assumed that this proxy works only for the **class** Factory. The specialization of the factory for this case is in the file `AllDynamic/functionProxy.hpp`, reported below.

```
#ifndef H_FUNCTIONPROXY_HH
#define H_FUNCTIONPROXY_HH
#include <string>
#include "QuadratureRule.hpp"
#include "ProxyFunction.hpp"
#include "Factory.hpp"

//!Specialization of Factory and Proxy for Functions
namespace FunctionsFactory
{
    using NumericalIntegration::FunPoint;

    typedef GenericFactory::Factory < FunPoint, std::string, BuilderFun >
      FunctionsFactory;

}
#endif
```

The third parameter template, in the **class** Factory is now used indicating the new builder. The registration of the functions stored in `udf.*pp` are in the file `AllDynamic/myRules.cpp`, reported below.

```
#include "QuadratureRule.hpp"
#include "functionProxy.hpp"
#include "udf.hpp"

// Registration in the factory I use an unnamed namespace
namespace
{
    using NumericalIntegration::FunPoint;
    using namespace FunctionsFactory;

    ProxyFunction FSinCos( "fsincos", FunPoint( fsincos ) );

    ProxyFunction FPippo( "pippo", FunPoint( pippo ) );

    ProxyFunction FPluto( "pluto", FunPoint( pluto ) );

    ProxyFunction FP( "myfun", FunPoint( myfun ) );
}
```

We have registered, in an anonymous **namespace**, the four functions using the interface of the new proxy. We directly pass the wrapped objects and not each the free function itself. The Makefile should be modified to take into account the new library and its dependence with `libudf.so`. If we want to handle functor class we can use the same strategy. In the file `AllDynamic/myFunctor.*pp` we have introduced an example of standing alone library, which register a functor. The code is reported below.

```
#ifndef MYFUNCTOR_HPP
#define MYFUNCTOR_HPP

struct myFunctor
{
    myFunctor() = default;

    double operator() ( const double & x )
    {
        return x + M_a;
    }

private:

    double M_a{9.};

};

#endif
```

And the registration is done in the following source file.

```
#include "QuadratureRule.hpp"
#include "functionProxy.hpp"
#include "myFunctor.hpp"

// Registration in the factory I use an unnamed namespace
namespace
{
    using NumericalIntegration::FunPoint;
    using namespace FunctionsFactory;

    ProxyFunction FmyFunctor( "functor", FunPoint( myFunctor() ) );

}
```

Finally in the main program we read all the library defined in the data file and we compute the integral. An example is reported below.

```
#include <iostream>
#include <dlfcn.h>
#include <memory>
#include <vector>
#include "GetPot"
#include "numerical_integration.hpp"
#include "ruleProxy.hpp"
#include "functionProxy.hpp"

void printHelp();

template <typename FactoryType>
int checkObject( const std::string& , const FactoryType&, const std::string& );


int main(int argc, char** argv){

  using namespace Geometry;
  using namespace std;
  using namespace NumericalIntegration;
  using QuadratureRuleFactory::RulesFactory;
  using FunctionsFactory::FunctionsFactory;

  // Process options given to the program
  {
    GetPot key_input(argc,argv);
    if (key_input.search(2, "--help", "-h")){
      printHelp();
      exit(0);
    }
  }

  // Get the input file
  GetPot   cl("quadratura.getpot");
```

```
// Load library with the rules
string quadlib=cl("library","libmyrules.so");
void * dylib=dlopen(quadlib.c_str(),RTLD_NOW);
if (!dylib){
  cout<< "cannot find library" << quadlib <<endl;
  cout<< dlerror();
  exit(1);
}

// Now get the library with the functions to be integrated
const int libNumber = cl.vector_variable_size("udflib");
cout << "Number of libraries " << libNumber << endl;

vector<string> userdeflibs(libNumber);

for ( unsigned int i = 0; i < libNumber; ++i )
{
  userdeflibs[i] = cl("udflib", "./libmyFunctions.so", i );
}

vector<void *> dyFunlib(libNumber);

for ( int i = 0; i < libNumber; ++i )
{
    dyFunlib[i] = dlopen ( userdeflibs[i].c_str(),RTLD_NOW);
    if (!dyFunlib[i])
    {
      cout<< "cannot find library" << userdeflibs[i] <<endl;
      cout<< dlerror();
      exit(1);
    }
}

// Get the factory with the functions
FunctionsFactory & functionFactory( FunctionsFactory::Instance());

// Handle the library and get the integrand function
std::string fun_name=cl("integrand","myfun");

// Extract the function. I usa an auto_ptr to be sure to eventually delete the object
std::unique_ptr<FunPoint> f = functionFactory.create( fun_name );

if ( f == nullptr )
{
  int checked(1);
  for ( auto i : userdeflibs )  checked *= checkObject( fun_name, functionFactory, i );
  exit( checked / libNumber );
}

// Load all other info
double a=cl("a", 0.);
double b=cl("b", 1 );
int nint=cl("nint", 10);
string rule=cl("rule","Simpson");
// Get the factory with the rules
RulesFactory & rulesFactory( RulesFactory::Instance());
// Extract the rule. I usa an auto_ptr to be sure to eventually delete the object
QuadratureRuleHandler theRule=rulesFactory.create(rule);

if ( theRule == nullptr )
{
  const int checked =  checkObject( rule, rulesFactory, quadlib );
  exit(checked);
}

//  get the rule
cout<<"Integral between "<<a<<" and "<<b <<" with "<< nint <<" intervals"<<endl;
cout<<"Using rule "<<rule<<" and integrand "<<fun_name<<endl;

// Compute integral
Domain1D domain(a,b);
Mesh1D mesh(domain,nint);
Quadrature s(*theRule,mesh);
double approxs=s.apply(*f);
```

```
  cout<<" Result= "<<approxs<<endl;

  // Close rule library
  dlclose(dylib);

  for ( auto & i :dyFunlib ) dlclose( i );

}

void printHelp()
{
    using std::cout;
    using std::endl;
    cout<<"*** PROGRAM FOR COMPOSITE INTEGRATION  ***"<<endl<<endl;
    cout<<"*** Line Options ***"<<endl;
    cout<<"[-h  --help] This help"<<endl;
    cout<<"[InputFile=string] Input file name (quadrature.getpot)"<<endl<<endl;
    cout<<"*** File Options ***"<<endl;
    cout<<"[library=string] Quadrature rule library (libmyrules.so)"<<endl;
    cout<<"[udflib=string]  Library holding integrands (libudf.so)"<<endl;
    cout<<"[integrand=string] Integrand function name (myfun)"<<endl;
    cout<<"[rule=string] Quadrature rule name (Simpson)"<<endl;
    cout<<"                If rule=? the list of registered rules is given"<<endl;
    cout<<"[a=float] Left integration point (0)"<<endl;
    cout<<"[b=float] Right integration point (1)"<<endl;
    cout<<"[nint=int] Number of intervals (10)"<<endl;
}

template <typename FactoryType>
int checkObject( const std::string& objname,
                 const FactoryType& factory, const std::string& libname)
{
    using std::cout;
    using std::endl;

    if ( objname=="?" )
    {
        auto lista = factory.registered();
        cout<<" The following objects are registered in " << libname << endl;
        for ( const auto & i : lista )
        {
            cout << i << endl;
        }
        return 0; // exit without error status
    }
    else
    {
        return 2;
    }

}
```

And the data file is the following with the list of all the library loaded dynamically.

```
integrand=functor
udflib='./libmyFunctions.so ./libmyFunctor.so'
library=./libmyrules.so
rule=Simpson
#rule=Trapezoidal
```