3. Big Refactor, making World

Created by Anton Hritsan, last modified on Sep 15, 2021

Table of content

Goal

Right now, we can deal with one object in world. Adding two more will not make any complexity in code, but what about hundreds of them? One of developer responsibility is taking into account technical duties in code base. And it's a perfect time to refactor your code before making it complex.

Patterns

You need to implement at least next list of patterns:

- Factory
  - for making object independently
- Singleton\Locator\(for lazy people, extern pointer to class)
  - For accessing Application instance
- [very optional] Entity Component System
  - For storage parts as own components

Factory

Factory - one the most underestimated and overestimated pattern. Pattern gives you an opportunity to make objects without knowing but it has numerous amount of disadvantages too. You can make it by own or using internet pages to find the best implantation for you. As far as this pattern is key one and mandatory to use in this training, you can see my sample of implementation.

› Factory exaplanation

What is the classical way of Factory pattern? By given identificator - you have to get a new instance of this type. For example:

Simple sample

```
void* FactoryCreate( int id )
{
    switch( id )
```

```
    {
    case 0:
        return new int();
    case 1:
        return new Application();
    case 2:
        return new Camera();
    }

    return nullptr;
}


int main()
{
    int* pInt = ( int* )FactoryCreate( 0 );
    Application* pApplication = ( Application* )FactoryCreate( 1 );
    Camera* pCamera = ( Camera* )FactoryCreate( 2 );
}
```

it is really ugly and not obvious to use. First of all, need to remember relation between id==type, second, factory function MUST know about type to create it(need include everything before use it!).

We can get rid of most problems by using deep c++ usage

Improvement one: using pointer to function

Current factory function can be used not directly, so we can get opportunity to make function more or less common

Simple sample

```
void* FactoryCreate( int id )
{
    switch( id )
    {
    case 0:
        return new int();
    case 1:
        return new Application();
    case 2:
        return new Camera();
    }

    return nullptr;
}


void Application::Init( const char* commandLine, const HINSTANCE& hInst )
{
    using pFactoryFunc = void* ( * )( int );
    pFactoryFunc = &FactoryCreate;
```

```
    int* pInt = ( int* )pFactoryFunc( 0 );
    Application* pApplication = ( Application* )pFactoryFunc( 1 );
    Camera* pCamera = ( Camera* )pFactoryFunc( 2 );
```

Improvement two: Adding template to get rid of 'id'

template function has THE SAME signature even with different type of template parameter.

Simple sample

```
template< typename type >
void* FactoryCreate()
{
    return new type();
}


void Application::Init( const char* commandLine, const HINSTANCE& hInst )
{
    using pFactoryFunc = void* ( * )();
    pFactoryFunc pFuncInt = &FactoryCreate<int>;
    pFactoryFunc pFuncApplication = &FactoryCreate<Application>;
    pFactoryFunc pFuncCamera = &FactoryCreate<Camera>;

    int* pInt = ( int* )pFuncInt();
    Application* pApplication = ( Application* )pFuncApplication();
    Camera* pCamera = ( Camera* )pFuncCamera();
```

Wait, look at this code one more time. template function has THE SAME signature but produces different result! This is pure 'type erasure' pattern!

Improvement three - complex type erasure usage

As far as type erase has unified signature, we can use it to make relation between 'id' and pointer to func.

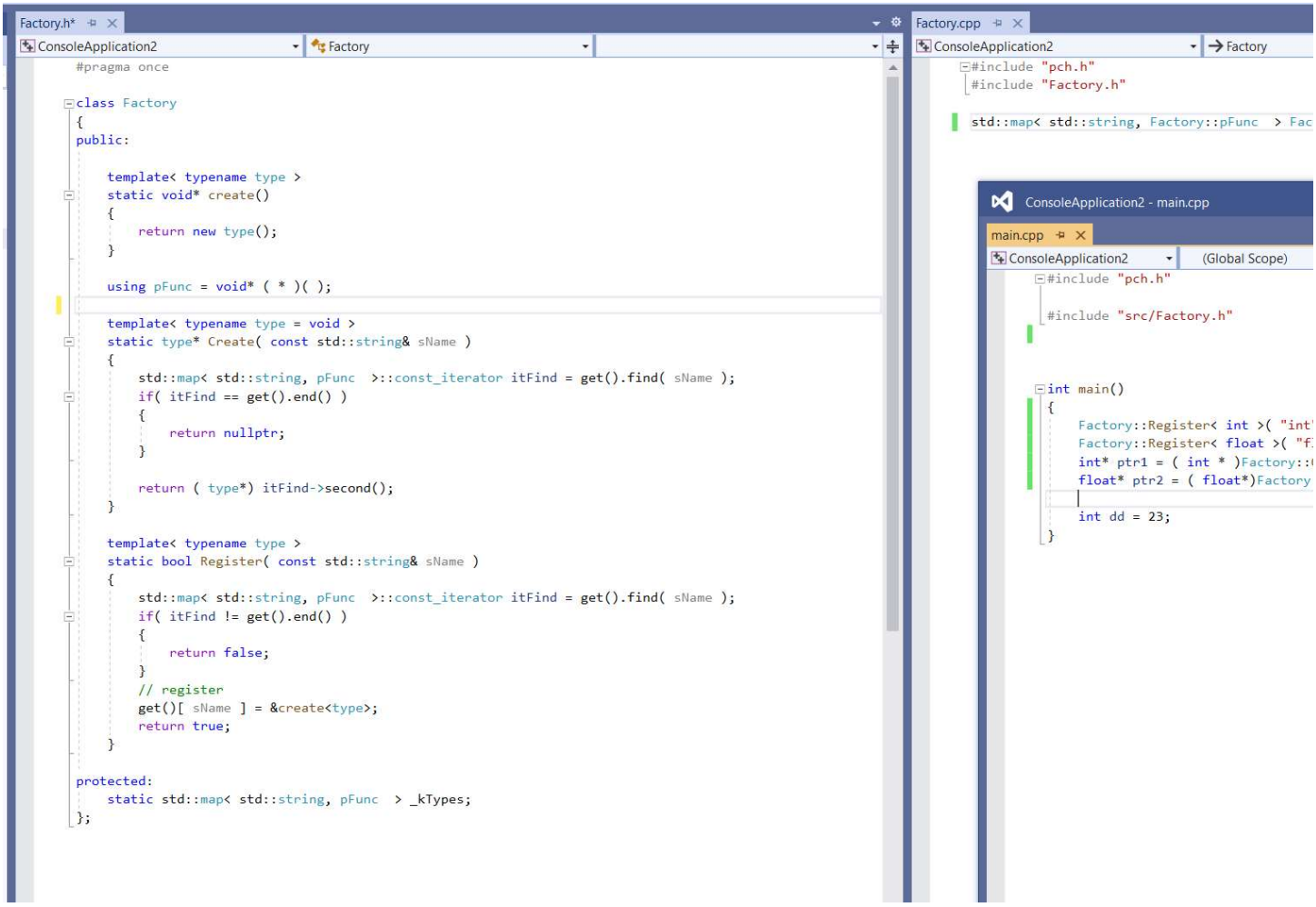
Simple sample

```
template< typename type >
void* FactoryMaker()
{
    return new type();
}
using pFactoryFunc = void* ( * )( );

// global!
std::unordered_map< std::string, pFactoryFunc > FactoryMap = {};

void* FactoryCreate( const std::string& sName )
{
    decltype( FactoryMap )::const_iterator itFind = FactoryMap.find( sName );
    if( itFind != FactoryMap.end() )
```

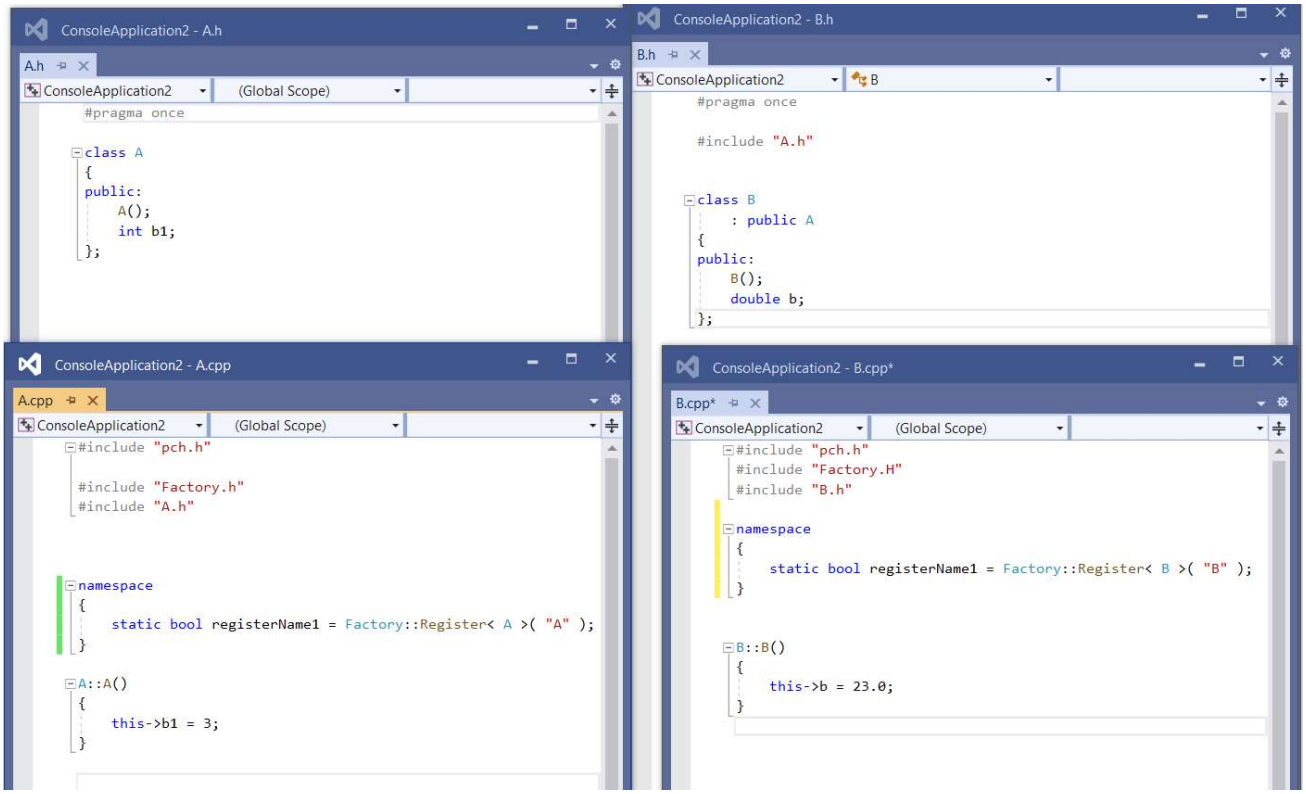
```
    {
        // call template function for registered type!
        return itFind->second();
    }

    // NOT registered type!
    return nullptr;
}

void Application::Init( const char* commandLine, const HINSTANCE& hInst )
{
    // register types
    FactoryMap[ "int" ] = &FactoryMaker<int>;
    FactoryMap[ "Application" ] = &FactoryMaker<Application>;
    FactoryMap[ "Camera" ] = &FactoryMaker<Camera>;

    int* pInt = ( int* )FactoryCreate("int");
    Application* pApplication = ( Application* )FactoryCreate( "Application" );
    Camera* pCamera = ( Camera* )FactoryCreate( "Camera" );
```

If we know pointer for FactoryCreate function , we can make desired type independently of factory and place where we are getting result! Yey! this is much better factory pattern then first raw idea! But we still to register types somewhere. Before making final result, let's cover such functional into logical unit.

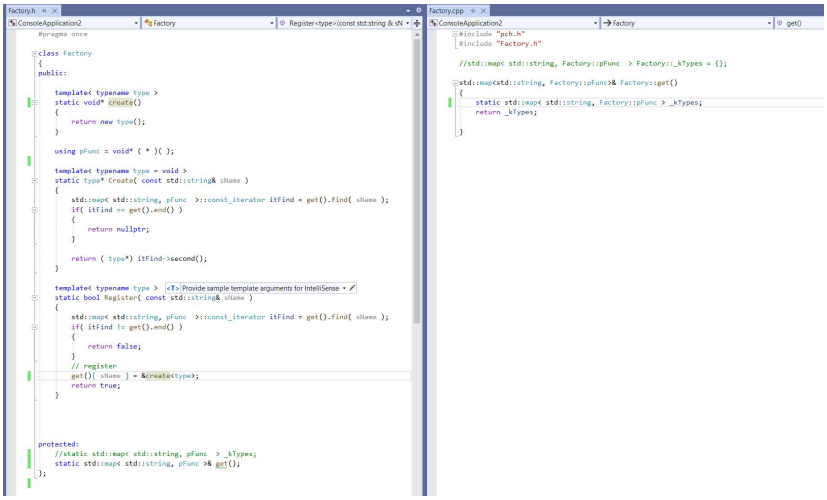
To avoid very long codes, next samples in screenshots



As you see, I've put everything in to class with static access for each var\func. Let's change POD types into some simple hierarch like



As you see, I've added self initing by one more hack of c++. Anonymous namespaces guarantees us unique id per .obj file. As there is initing static bool function - it will call static function in Factory. This is pure hack that works in pre-init time of application, but there is one critical problem. C++ doesn't guarantee an order of initing global vars. It means, this code CAN work and SHOULD NOT at the same time. Why? Static map in Factory class can be inited AFTER calling Factory::Register function from B.obj and there will be a 100% crash. To deal with it, we should use c++ strict rules one more time. We need to provide a functional to guarantee static map inited at any time. We can't alter compilator, but we can make function that returns static variable - does it sound like a fix? Yes, any call to such function will be iron proof for us of existing and inited variable. Sample below



Amazing, we have all function done. Our Factory usage can be like

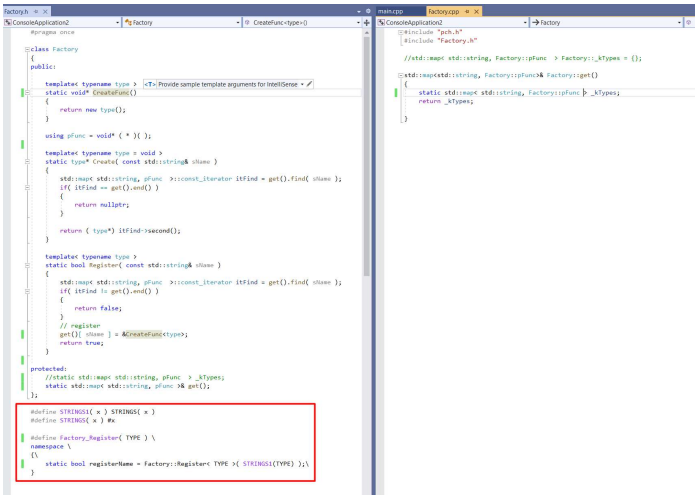
main.cpp | Factory.cpp

ConsoleApplication2 | (Global Scope)

```
#include "pch.h"

#include "src/Factory.h"
#include "src/A.h"


int main()
{
    A* ptr1 = Factory::Create<A>( "A" );
    A* ptr2 = Factory::Create<A>( "B" );

    int dd = 23;
}
```

Note: I haven't included B.h in the main.cpp. But I can obtain an instance to it!

One last point I wanna consider in this code. Registering... this is ugly way, no excuses. Let's convert it so something useable. What if Simplify registering to macros? Sounds like a plan. Simple and naïve implantation



usage and final result will be

Full sample project with code here

Singleton\Locator

You should implement it by your own. No jokes, these patterns are common in development, so you should not have any troubles with it.

Configs

Setup world in code, not the best idea because without re-compilation - you can't change anything. So with that approach there will be cried game designer in the world. Much better idea to add not compilation files with description of object next to the binary program. In our case, you will use .json format for such configurations. You can use whatever you want json parser, an a sample, I attached in document an archive with one of the very popular one. A full documentation about json format and possible usage of libraries you definitively can find in free internet. Check out at least this PAGE to be sure that your knowledge is really true and right.

an archive with jsoncpp library below

You need to add it in to solution and make it work. I will not describe in details such process because it's a minimal knowledge to dealing with Visual Studio, solutions, and projects(do not try to steal sources from library, by definition you must respect licenses). Feel free to use internet help if you're in trouble with it.

Sort of conclusion. In order to describe your world, you will use .json format with metadata inside. One important things need to be mentioned, you need to accommodate your metadata into Factory usage(this is MANDARY to have !)

Class hierarchy and architecture decision

First and main point - you can do what ever you want! Your own idea for implementing architecture - is the best idea ever. Few mandatory points for architecture

1. Object must be created anonymously by Factory Pattern
2. Configuration file must be read only once at start of Application class
3. World class must use ONLY base interface for Objects(you can name it whatever you want as well)
4. Ray functional has to be separated into logical unit\structure as far as this structure has to be scaled up in time

Please notice, Right after this section of training, all code samples will be incomplete and cover only necessary info for tasks. This is point no return, if you design bad architecture - you can really struggle next. Asking for a help - this is okay, and your assigned trainer will ask few times about your approach for architecture and class hierarchies. Make it cool!

Tasks

Draw a UML representation of your architecture

actually, it should be done BEFORE you have your code completed

Describe several objects in json configuration and draw them

minimal information for each objects in world:

- position in 3d space
- radius for sphere

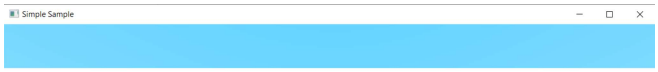
Minimum information about 'camera' in config:

- position
- direction
- [optional] FOV value(Field Of View)

Make 'Ground' object

To fake it, you can use a BIG sphere(for example with radius 100.0f)

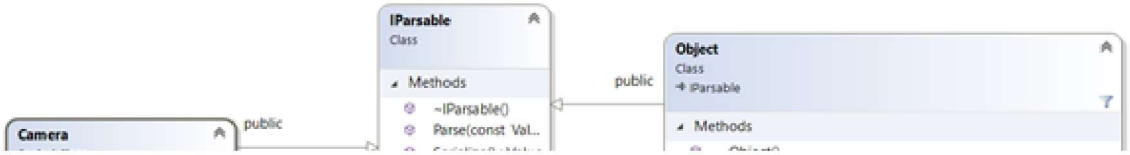
Sample of possible result



Sample of architecture

If you're not good at making scalable architecture, you can use this hint as reference. I highly recommend you to make your own.

- Sample of diagram here
  - Common! Don't do it! Make it own!
    - I'm sure you don't need this sample of diagram, otherwise proceed to next link


IParsable
Class
Methods
~IParsable()
Parse(const Val...
public
Object
Class
IParsable
Methods
public
Camera

No labels