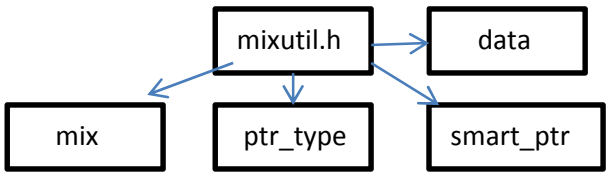
INTRODUCTION

The “mixutil.h” is a special purpose library designed for wrapping any C++ statements’ syntaxes that are deemed so lengthy in keywords. Besides that, it is also contained some features that seemed to be useful apart of merely wrapping any lengthy C++ keywords.

Due to the ongoing development process of the library, up to this point, the “mixutil.h” framework consists of the following sub namespaces:



The library framework is subdivided into several meaningful namespaces’ names, that consists of “mix”, the namespace’s name of itself, “ptr_type”, the namespace’s name which contains definition of any types of pointer, “smart_ptr”, the namespace’s name dedicated to any definition of smart pointers, and “data”, the namespace’s name that merely defined some useful data structures that later can be used in any programming tasks using C++.

To use the library, you better include the pre-declaration of any needed namespaces’ names in a separate header file called as “mixer.h”. The “mixer.h” header should contain only the forward declaration of any needed namespaces’ names provided by the “mixutil.h”.

For clarity, the following listing shows whatever things should be written to the “mixer.h” header.

```
#ifndef MIXER_H
#define MIXER_H
      using namespace mix;
      using namespace mix::ptr_type;
      using namespace mix::smart_ptr;
      using namespace mix::data;
#endif
```

The “using .. “ statements may be expanded a bit long or may be contracted a little short depending on what features is going to be used from the “mix” framework.

To the point, I’ll show you the main overview of program using the mix framework which incorporated into your C++ project.

```
#include “mixutil.h”
#include “mixer.h”


int main()
{


     return -1;

}
```

CLASSES’ FRAMEWORK

The “mix” namespace name is the primary namespace encapsulating the remaining, acts as an auxiliary to the entire “mix” framework structure. Its purpose is to deliver any miscellaneous features that might be needed somewhat in the construction of other namespaces definition.

The mix::ptr_type namespace

The “mix::ptr_type” namespace contains any type of pointers definition which types are in matched with the modern C++ smart pointers, those are std::unique_ptr and std::shared_ptr. The definition consists of both smart pointers’ types to a single object or to an array of objects, accompanied with the template parameter for the user specified “deleter” function object of their own implementations beside the standard predetermined “default deleters”.

```
// a single unique pointer type
template < class Ty, class Del1X = std::default_delete<Ty> >
using uniqueP = std::unique_ptr<Ty, Del1X>;
```

```
// a single shared pointer type
template < class Ty >
using shareP = std::shared_ptr<Ty>;
```

Usages:

```
// a single unique pointer to the Bucket object
uniqueP<Bucket> upBucket; // uses default deleter


/* same as above, but uses the customized function object
 deleter */

uniqueP<Bucket, del_unique<Bucket>> upBucket;


// a single shared pointer to the Bucket object
shareP<Bucket> spBucket; // uses default deleter


/* same as above, but uses custom deleter */
shareP<Bucket> spBucket = sp_create<Bucket,del_shared<Bucket>>(0);
```

NB: The zero '(0)' argument specified to the 'sp_create' template function doesn't have a specific meaning, it does only serve as an indication that an overloaded version of 'sp_create' is being called.

Or we can design a specific lambda expression as a customized function delete object specified to the 'sp_create' template parameter.

```
auto uDel = [](Bucket* _pb) {delete _pb; };

using fDel = _TYPE(uDel);

shareP<Bucket> shp = sp_create<Bucket,fDel>(0);
```

More information on user custom 'deleter' function with how to specify and how to implement it, are available through my blog website. Or you can 'googling' thru other website of the same kind of topic.

How to Create a Smart Pointer to any Array of [n] Elements of a Type Specified.

Basically, the above shown examples are only demonstrating a single smart pointer to a single object. How if we would to have smart pointer pointing to any array of [n] elements of any type specified?

Using the functionalities provided by the library “mixutil.h” and access to the respective namespace’s name, we can manifesting those requests into the real instances of their objects in C++.

Usages:

```
/* A unique pointer to the integer array of [n] elements,
      using its standard default deleter */
UNIQUE_ARRAY<int> upArray;


/* A shared pointer to the integer array of [n] elements,
      using its standard default deleter */

SHARED_ARRAY<int> shrArray;
```

Both the declared smart pointers to array are equivalent to the old style fashion of declaring a pointer to array of [n] integers.

For example: `int* p[] = { new int, new int, new int };`

To make them really point to the specific array, we needed factory creator for each instance of smart pointer declared as such as previously shown in the above.

Unique Factory

A Unique Factory object is purposely designed to yield a specific unique smart pointer to any array of [n] elements of a type specified in its template parameter. The following declares ‘ubFactory’ as a unique factory of ‘Bucket’ type.

```
// a unique factory uses default deleter
unique_array_ptr<Bucket> ubFactory;

// this one uses custom deleter function object
unique_array_ptr<Bucket,unique_del<Bucket*>> uArrayFactory;
```

Then, we need to have a unique smart pointer to point to the array of Bucket yielded by 'ubFactory'.

```
// a unique pointer to array of Buckets of 3 elems
UNIQUE_ARRAY<Bucket>&& upBucket = ubFactory.create(3);

/* same as above, but this one should paired with a unique factory that has
   custom deleter specified to it. */

UNIQUE_ARRAY<Bucket,unique_del<Bucket*>>&&
upBucket= uArrayFactory.create(3);
```

Shared Factory

A Shared Factory is the object designed to produce a shared smart pointer to any array of [n] elements of a type specified in its template parameter. The following shows how to declare a Shared Factory of Bucket type.

```
// a shared factory of Bucket type to produce 3 elements
Alloc_Share<Bucket> sharedFactory(3);

// A Shared Factory with user custom deleter
Alloc_Share<Bucket, del_shared_array<Bucket*>> uSharedFactory(3);
```

And again, we just create a shared smart pointer to match with the output produced by a 'sharedFactory' object.

```
// uses standard default deleter
SHARED_ARRAY<Bucket> shrBucks = sharedFactory.get_shared();

// uses custom deleter
SHARED_ARRAY<Bucket> uShrBucks = uSharedFactory.get_shared();
```

Initialize a Unique and Shared Smart Pointer

The type of smart pointers declared so far have to be initialized. Initializing a single smart pointer is a bit different with the initialization of smart pointer to array. These are valid for both a unique and a shared smart pointer.

```
Initialize a unique smart pointer to a Bucket Object.
uniqueP<Bucket> upBucket = up_create<Bucket>();
```

```
Initialize a shared smart pointer to a Bucket Object.
shareP<Bucket> shrBucket = sp_create<Bucket>();


Initialize a unique smart pointer to the array of Buckets.
UNIQUE_ARRAY<Bucket>&& upArray; // Bucket* pb[3] = { };

upArray = uArrayFactory.initialize( {Bucket("0"),Bucket("1"),Bucket("2")} );


Initialize a shared smart pointer to the array of Buckets.
SHARED_ARRAY<Bucket> shrBucks;

shrBucks = sharedFactory.initialize({Bucket(), Bucket(), Bucket()});
```