Subject: About Nuller and Null
Posted by Tom1 on Sat, 10 Oct 2020 08:58:09 GMT
View Forum Message <> Reply to Message

Hi,

I failed to find much documentation about Nuller and Null. Then I just looked around the source and tried to put together a generic macro to support Nuller/Null in a class.

Could someone with deeper understanding confirm if my following NULLSUPPORT -macro covers all the relevant aspects of supporting Null for a class?

```
#define NULLSUPPORT(x)\
 CLASSNAME(const Nuller&){ SetNull(); }\
 void SetNull(){ x=Null; }\
 bool IsNullInstance() const { return IsNull(x); }

class A{
public:
 typedef A CLASSNAME;

 NULLSUPPORT(a);

 int a;
 int b;

 A(){
  a=0;
  b=0;
 }
};
```

Best regards,

Tom

Subject: Re: About Nuller and Null
Posted by mirek on Sat, 10 Oct 2020 17:32:05 GMT
View Forum Message <> Reply to Message

Seems fine to me.

Subject: Re: About Nuller and Null
Posted by Tom1 on Sat, 10 Oct 2020 18:25:40 GMT

Hi Mirek,

Thanks for looking into this. I really have trouble and feel insecure about returning Null references. The access to Array and Vector containers comes as references. So, when I create a function returning those references, I need to be able to return Null if the container does not have a suitable object to return for a request.

However, returning a Null reference is not trivial. And possibly also forbidden in C++. Then, I looked at using pointers instead and found that C++ references have the following limitation:

"There shall be no references to references, no arrays of references, and no pointers to references. " (ISO C++)

Finally (after quite a few hours) I came up with the following solution: Using: " return (A&)Null; " to return a Null reference. How dangerous is this? (I also added the check: " this==&(classname&)Null " to IsNullInstance() in order to cover this case.

In contrast to the previous code the following compiles with CLANG too and seems to work as expected:

```
#include <Core/Core.h>

using namespace Upp;

#define NULLSUPPORT(classname, variable)\
 classname(const Nuller&) { variable=Null; }\
 void SetNull() { variable=Null; }\
 bool IsNullInstance() const { return this==&(classname&)Null || IsNull(variable); }

class A{
public:
 int a;
 int b;

 NULLSUPPORT(A,a)

 void Clear(){ a=b=0; }

 A(){
  a=1;
  b=2;
 }

 void Serialize(Stream &s){
  s % a % b;
 }
```

```
 String ToString() const { return IsNullInstance() ? String("Null") : String("A[") << a << ", " << b <<
"]"; }
};

// Testing:

Array<A> av;

A& GetA1(int x){
 if((x<0)||(x>=av.GetCount())) return (A&)Null;
 return av[x];
}

CONSOLE_APP_MAIN{
 av.Add().a=1;
 av.Add().a=2;
 av.Add().a=3;
 av.Add().a=4;

 for(int i=-1;i<6;i++){ A &a=GetA1(i); Cout() << a << "\n"; }
 return;
}
```

But is this safe? If not, is there a decent way to do it?

Best regards,

Tom

Subject: Re: About Nuller and Null
Posted by Tom1 on Sat, 10 Oct 2020 22:02:45 GMT
View Forum Message <> Reply to Message

Hi,

How about this? I did not benchmark the performance, but at least this is not relying on testing for a Null reference. As you can see, the 'Optional' is named in the foot steps of std::optional which is available in C++17 for the same purpose. (However, std::optional does not seem to support passing reference variables.)

```
#include <Core/Core.h>

using namespace Upp;

template <typename T>
struct Optional : public Tuple2<bool, T>{
```

```
 typedef Tuple2<bool, T> Base;
 Optional(T data) : Base(true, data) { }
 Optional() : Base(false, (T)Null) { }
 inline bool IsOK(){ return (bool)Base::a; }
 inline T Get(){ return (T)Base::b; }
};

class A{
public:
 int a;
 int b;

 A(){
  a=1;
  b=2;
 }

 String ToString() const { return String("A[") << a << ", " << b << "]"; }
};

// Testing:

Array<A> av;

Optional<A&> GetA2(int x){
 if((x<0)||(x>=av.GetCount())) return Optional<A&>();
 return Optional<A&>(av[x]);
}

CONSOLE_APP_MAIN{
 av.Add().a=1;
 av.Add().a=2;
 av.Add().a=3;
 av.Add().a=4;

 for(int i=-1;i<6;i++){
  Optional<A&> result=GetA2(i);
  Cout() << (result.IsOK() ? AsString(result.Get()) : "Null") << "\n";
 }
}
```

Best regards,

Tom

Subject: Re: About Nuller and Null
Posted by mirek on Sat, 10 Oct 2020 23:39:01 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Sat, 10 October 2020 20:25Hi Mirek,

Thanks for looking into this. I really have trouble and feel insecure about returning Null references. The access to Array and Vector containers comes as references. So, when I create a function returning those references, I need to be able to return Null if the container does not have a suitable object to return for a request.

However, returning a Null reference is not trivial. And possibly also forbidden in C++. Then, I looked at using pointers instead and found that C++ references have the following limitation:

"There shall be no references to references, no arrays of references, and no pointers to references. " (ISO C++)

Finally (after quite a few hours) I came up with the following solution: Using: " return (A&)Null; " to return a Null reference. How dangerous is this? (I also added the check: " this==&(classname&)Null " to IsNullInstance() in order to cover this case.

In contrast to the previous code the following compiles with CLANG too and seems to work as expected:

```
#include <Core/Core.h>

using namespace Upp;

#define NULLSUPPORT(classname, variable)\
 classname(const Nuller&) { variable=Null; }\
 void SetNull() { variable=Null; }\
 bool IsNullInstance() const { return this==&(classname&)Null || IsNull(variable); }

class A{
public:
 int a;
 int b;

 NULLSUPPORT(A,a)

 void Clear(){ a=b=0; }

 A(){
  a=1;
  b=2;
 }

 void Serialize(Stream &s){
  s % a % b;
```

```
 }

 String ToString() const { return IsNullInstance() ? String("Null") : String("A[") << a << ", " << b <<
"]"; }
};

// Testing:

Array<A> av;

A& GetA1(int x){
 if((x<0)||(x>=av.GetCount())) return (A&)Null;
 return av[x];
}

CONSOLE_APP_MAIN{
 av.Add().a=1;
 av.Add().a=2;
 av.Add().a=3;
 av.Add().a=4;

 for(int i=-1;i<6;i++){ A &a=GetA1(i); Cout() << a << "\n"; }
 return;
}
```

But is this safe? If not, is there a decent way to do it?

Best regards,

Tom

I am totally cofused what are you trying to achieve here...

Both Null and Nuller are never supposed to be used outside of "assigning Null syntax sugar" context.

I think you might be overthinking something here.

Mirek

Mirek

Subject: Re: About Nuller and Null
Posted by Tom1 on Sun, 11 Oct 2020 08:34:47 GMT

Hi,

I sure have been overthinking, and then some! :)

I just wanted to basically return a null pointer (instead of a pointer to the result) when a function cannot solve a valid result. Then by checking for a null pointer, I could determine if the function succeeded or not.

When working on container classes based on e.g. Vector or Array classes, I would obtain the result as a reference to the item. Or the solution might fail, in which case I would return a null reference. But null references are not allowed or their behavior is undefined. So using null references is likely just asking for trouble.

Then I (naively) figured out Upp::Null and Nuller are just right for the purpose. However, it seems this is not the case. I cannot easily/safely return a Null object in place of a reference. The problems I have encountered while trying to work around the issue include:

- 'warning: returning a reference to a local or temporary object' when returning a T(Null) for an object
- returning null references are generally undefined and should not exist in C++
- There shall be no pointers to references in C++, which prevents changing my function to return pointers and null pointers alternatively

After quite some hours of tinkering, I came up with the Tuple2<bool,T> based solution to get a feeling of returning a pointer/null.

```
template <typename T>
struct Optional : public Tuple2<bool, T>{
 typedef Tuple2<bool, T> Base;
 Optional(T data) : Base(true,data) { }
 Optional() : Base(false,(T)Null) { }
 inline operator bool() const { return (bool)Base::a; }
 inline operator T(){ return (T)Base::b; }
 inline bool IsOK() const { return (bool)Base::a; }
 inline T Get(){ return (T)Base::b; }
 inline bool IsNullInstance() const{ return !IsOK(); }
};

// Usage:
//
// Optional<T> func(){
// if(success) return Optional<T>(value);
// else return Optional<T>();
// }
//
// In the calling function:
//
```

```
// Optional<T> result = func();
// if(!result) Cout() << "Failed, returned null\n";
// else Cout() << "Success, returned " << result << "\n";
```

Please note that this can return real references, if T is a reference.

If you see any flaws in this approach, or have a cleaner way to do it, please let me know.

Best regards,

Tom

Subject: Re: About Nuller and Null
Posted by mirek on Sun, 11 Oct 2020 09:22:29 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Sun, 11 October 2020 10:34

Please note that this can return real references, if T is a reference.

Well, Null is about "value null", or "empty value". Definitely was not meant to be used with NULL references or pointers...

If you insist on returning a reference to something and you want to use Null value as error, you can always do something like

```
const Foo& GetData(...)
{
   static Foo null_data = Null;
...
   if(error) return null_data;
...
}
```

I mean, instead of inventing something to contain NULL reference, just return a reference to Null value...

Subject: Re: About Nuller and Null
Posted by Tom1 on Sun, 11 Oct 2020 10:34:22 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Thanks for fixing my thoughts. Now that you pointed it out, using a static initialized to Null as the return value is definitely the clear and easy way out. I wonder why I did not think of that...

Thanks and best regards,

Tom
