Subject: The problem with 'Null'
Posted by gridem on Thu, 19 Mar 2009 07:04:25 GMT
View Forum Message <> Reply to Message

I found that Upp uses the following practice: instead of creating already prepared object it creates the object with default constructor and than fill the necessary members in later calls. But using such approach the programmer should distinguish between init and non init state. One of the possible solution: apply 'Null' to the fields and than check by using IsNull.

The simple types (int, long etc) and Value already have such posibility. String also can use this but using another functionality:
String::GetVoid() return 'super' empty string that may be treated as 'Null'. The problem is that IsNull(String()) and IsNull(String::GetVoid()) return both true. This may be workarounded but it's not a good solution. But for the Vector<T> the workaround is more complex: the programmer should use One<Vector<T> >. The problem may occur in situation when function should return result or error. In the following example:
String LoadFile(...)
the solution exist: return String::GetVoid() on error. But what I can do when I must return Vector:
Vector<Templates> GetTemplateList()
Empty list denotes the there are no templates. But how can I return error without using terrible One<Vector<T> >?

Subject: Re: The problem with 'Null'
Posted by gridem on Thu, 19 Mar 2009 07:42:02 GMT
View Forum Message <> Reply to Message

I found a dirty trick:

```
template<typename T>
struct PickedVector : Vector<T>
{
 PickedVector()
 {
  Vector<T>() = *this;
 }
};

template<typename T>
const Vector<T>& VectorNull()
{
 return Single<PickedVector<T> >();
}

template<typename T>
bool IsNull(const Vector<T>& v)
{
 return v.IsPicked();
}
```

```
Cout() << "Is Null: " << IsNull(Vector<int>()) << EOL;
Cout() << "Is Null: " << IsNull(VectorNull<int>()) << EOL;

output:
Is Null: false
Is Null: true
```
and picked state may be treated as Null.

Subject: Re: The problem with 'Null'
Posted by cbpporter on Thu, 19 Mar 2009 07:45:23 GMT
View Forum Message <> Reply to Message

gridem wrote on Thu, 19 March 2009 09:04But using such approach the programmer should distinguish between init and non init state. One of the possible solution: apply 'Null' to the fields and than check by using IsNull.
I think that with U++ design there is no such thing as init and non-init state. Pretty much all classes are in "init" state after the call of a constructor, even the default one. Such objects are initialized and ready to use, but generally hold no extra information. A String() will have zero length, a Vector() will have zero elements, a Button() will have no picture, text and other properties that diverge from default, but otherwise is ready to be inserted into a parent.

So basically the default constructor creates object in an already prepared state. I think the reason that we use default constructor in this way rather than constructor with parameters to set all the meaningful data is largely related to the intrinsic construction rules of C++. Since we don't put everything on the heap, the objects construction can't be deferred to the allocation moment and is constructed at the runtime point equivalent to the declaration. At this point you often don't have all the information required to call a constructor with parameters.

There are of course exceptions. Things like Size and Point will not be initialized by default construction and can result in bugs if the user is not aware of this fact. Here denoting the absence of an initialization would be useful, but I think these classes are supposed to be lightweight and this is why they are not initialized. Adding unfertilized state detection would make them more heavy weight and less efficient. Having a vector of 10 points would initialize all of them to default, and in most cases, you would reinitialize them in code again. By leaving them uninitialized you only get the meaningful initialization, but you risk bugs if you forget it.

As for the One<Vector <T> > solution, I think is not a very good one. Not only do you have to test if the Vector is not Null and loose the guarantee that Vector is initialized, but you also increase the level of indirection. Such tricks might look like something good, but my experience tells me that if you are going to write programs with hundreds of thousands of line of code and use less straight forward solutions, you'll end up causing more problems than you solve.

Wouldn't a call to IsEmpty or something equivalent on you return value solve the problem? Maybe I'm not understanding exactly what you are trying to achieve. Maybe if you can give a real world example where U++ initialization scheme is not working out and you need to determine

initialization status, then I could offer a better solution.

Subject: Re: The problem with 'Null'
Posted by mirek on Thu, 19 Mar 2009 08:16:34 GMT
View Forum Message <> Reply to Message

gridem wrote on Thu, 19 March 2009 03:04I found that Upp uses the following practice: instead of creating already prepared object it creates the object with default constructor and than fill the necessary members in later calls. But using such approach the programmer should distinguish between init and non init state. One of the possible solution: apply 'Null' to the fields and than check by using IsNull.

The simple types (int, long etc) and Value already have such posibility. String also can use this but using another functionality:
String::GetVoid() return 'super' empty string that may be treated as 'Null'. The problem is that IsNull(String()) and IsNull(String::GetVoid()) return both true. This may be workarounded but it's not a good solution. But for the Vector<T> the workaround is more complex: the programmer should use One<Vector<T> >. The problem may occur in situation when function should return result or error. In the following example:
String LoadFile(...)
the solution exist: return String::GetVoid() on error. But what I can do when I must return Vector:
Vector<Templates> GetTemplateList()
Empty list denotes the there are no templates. But how can I return error without using terrible One<Vector<T> >?

Null is for "full values".

Note that String::GetVoid is deliberately defined "Null" because e.g. for LoadFile, most code can safely ignore the error as long as they get "Null" file.

As for your vector example, I would simply use

bool GetTemplateList(Vector<Templates>& r);

Mirek

Subject: Re: The problem with 'Null'
Posted by mirek on Thu, 19 Mar 2009 08:20:30 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 19 March 2009 03:45
There are of course exceptions. Things like Size and Point will not be initialized by default

construction and can result in bugs if the user is not aware of this fact.

Well, that depends on definition of "initialized"

Is normal scoped

```
{
    int x;
```

initialized or not? I believe in sense it is - all resources required for it are allocated and it is ready to be used. Of course, contract does not define any particular initial value for it

(I agree this is mostly just pointless word-play).

Mirek

Subject: Re: The problem with 'Null'
Posted by gridem on Fri, 20 Mar 2009 07:16:50 GMT
View Forum Message <> Reply to Message

Thank you very much for detailed answers! It's very useful for me to understanding intrinsics of Upp.

But let's me to understand my opinion. I think that Null approach is not narrow-minded but generic. It uses for Value, it uses for many simple types.

Because I use a lot the templates, overloaded functions and code generation I have to utilize the generic methods for every types that I want to use. And I use Null as some kind of parameter state that I can treat as:
1. Init/Non init
2. Error on function return.
3. Default values to call the function to distinguish it from nondefault values:

```
void some_fun(int a, int b = Null, bool c = Null, ...)
{
  if (b == Null)
    b = some_complex_calculated_value(a);
  ...
}
```

etc

So to see this approach for String's and Vector's I have to do workaround. May be I should not use the generics but another mechanism?

Subject: Re: The problem with 'Null'
Posted by mirek on Fri, 20 Mar 2009 08:52:06 GMT
View Forum Message <> Reply to Message

gridem wrote on Fri, 20 March 2009 03:16Thank you very much for detailed answers! It's very useful for me to understanding intrinsics of Upp.

But let's me to understand my opinion. I think that Null approach is not narrow-minded but generic. It uses for Value, it uses for many simple types.

Because I use a lot the templates, overloaded functions and code generation I have to utilize the generic methods for every types that I want to use. And I use Null as some kind of parameter state that I can treat as:
1. Init/Non init
2. Error on function return.
3. Default values to call the function to distinguish it from nondefault values:

```
void some_fun(int a, int b = Null, bool c = Null, ...)
{
  if (b == Null)
    b = some_complex_calculated_value(a);
  ...
}
```

etc

Two notes:

You cannot really define Null for bool, as it has only 2 values..

It is more correct to use "IsNull" instead of "== Null".

Quote:So to see this approach for String's and Vector's I have to do workaround. May be I should not use the generics but another mechanism?

Well, I can see where you are heading, but I do not really like that path

My only apology at this moment is that U++ is "practice driven", and in the whole history (which now spans abuot 10 years), we never missing IsNull for containers...

BTW, as you have noticed, there is the small issue with String Null - empty string is considered Null.

I agree this is sort of controversial decision. Indeed, a couple of years ago, we identified it as mistake and tried

```
IsNull(String()) == false
IsNull(String(Null)) == true
String(Null) == String()
```

variant. Well, what happened is that in practice, this was found to be rather unfortunate. I guess the primary problem is that it is very convenient and natural, when working with databases and GUI, that all empty String gui fields are inserted as Nulls. With above, you would need to have additional GUI buttons to say whether the field is empty or whether it is null.

Similar issues can be found across the code. That is why we went back to

IsNull(String()) == true

As a sidenote, this equivalence was originally taken from Oracle.

Mirek

Subject: Re: The problem with 'Null'
Posted by gridem on Sun, 22 Mar 2009 08:15:57 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 20 March 2009 11:52

Two notes:

You cannot really define Null for bool, as it has only 2 values..

It is more correct to use "IsNull" instead of "== Null".

Oh, my mistake: IsNull of course instead of == Null.

luzr wrote on Fri, 20 March 2009 11:52

Well, I can see where you are heading, but I do not really like that path

My only apology at this moment is that U++ is "practice driven", and in the whole history (which now spans abuot 10 years), we never missing IsNull for containers...

BTW, as you have noticed, there is the small issue with String Null - empty string is considered Null.

I agree this is sort of controversial decision. Indeed, a couple of years ago, we identified it as mistake and tried

IsNull(String()) == false
IsNull(String(Null)) == true
String(Null) == String()

variant. Well, what happened is that in practice, this was found to be rather unfortunate. I guess the primary problem is that it is very convenient and natural, when working with databases and GUI, that all empty String gui fields are inserted as Nulls. With above, you would need to have additional GUI buttons to say whether the field is empty or whether it is null.

Similar issues can be found across the code. That is why we went back to

IsNull(String()) == true

As a sidenote, this equivalence was originally taken from Oracle.

Mirek

Thank you for your explanations. I caught the main idea. So for SQL programming it's possible to introduce 'local IsNull':

```
template<typename T>
bool IsNullSql(const T& t)
{
  return IsNull(t);
}

bool IsNullSql(const String& s)
{
  return s.IsEmpty();
}
```

Subject: Re: The problem with 'Null'
Posted by mirek on Mon, 23 Mar 2009 22:55:21 GMT
View Forum Message <> Reply to Message

Quote:
Thank you for your explanations. I caught the main idea. So for SQL programming it's possible to introduce 'local IsNull':

```
template<typename T>
bool IsNullSql(const T& t)
{
  return IsNull(t);
}

bool IsNullSql(const String& s)
{
```

```
  return s.IsEmpty();
}
```

But it really is not only about SQL. I agree that it is a little bit controversial, we know it, but the convention is really quite practical.

Mirek

Subject: Re: The problem with 'Null'
Posted by gridem on Tue, 24 Mar 2009 07:41:35 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 24 March 2009 01:55

But it really is not only about SQL. I agree that it is a little bit controversial, we know it, but the convention is really quite practical.

Mirek

I see. I think that I will use simple workaround for IsNull. At release mode the template inline functions should not reduce perfomance at all.

Subject: Re: The problem with 'Null'
Posted by mirek on Tue, 24 Mar 2009 10:04:02 GMT
View Forum Message <> Reply to Message

gridem wrote on Tue, 24 March 2009 03:41luzr wrote on Tue, 24 March 2009 01:55

But it really is not only about SQL. I agree that it is a little bit controversial, we know it, but the convention is really quite practical.

Mirek

I see. I think that I will use simple workaround for IsNull. At release mode the template inline functions should not reduce perfomance at all.

IMO, you might find that in most cases, you will not need it.. (just as we did

Mirek
