Subject: Order of member initialization
Posted by Tom1 on Tue, 20 Sep 2022 09:19:45 GMT
View Forum Message <> Reply to Message

Hi,

After quite some decades I feel like a newbie again. When I run:

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

using namespace Upp;

class A{
public:
 Array<int> &array;
 A(Array<int> &array_) : array(array_){
  Cout() << "Array Initial item count in A = " << array.GetCount() << "\n";
 }
};

class B : public A{
public:
 Array<int> array;
 B() : A(array){
  Cout() << "Array Initial item count in B = " << array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
}
```

I would expect to see item count zero in both constructors. However, in Windows on CLANGx64 it is 1 in A and 0 in B. This is true in both release and debug modes.

In MSBT22x64 it is correctly zero in release mode, but it can be any number in debug mode causing severe trouble.

Is there a way to re-order the initialization of class B so that the array gets initialized before class A?

Best regards,

Tom

Subject: Re: Order of member initialization

Posted by Tom1 on Tue, 20 Sep 2022 11:29:59 GMT
View Forum Message <> Reply to Message

Answering myself: The order of initialization is fixed. Inherited classes get first constructed and only thereafter the local member variables in the order of declaration.

The lesson learnt here is: A base class having member reference variables, referencing variables declared in inherited classes, is a very bad idea. The reference must be initialized when constructing the base class, but the variable cannot be initialized yet at this time. Very bad things happen.

Best regards,

Tom

Subject: Re: Order of member initialization
Posted by peterh on Tue, 20 Sep 2022 13:19:49 GMT
View Forum Message <> Reply to Message

Hi I am just a beginner learning (I have practice in C and Object Pascal) and find your example interesting, so I tried it.

It shows zero for A and for B here with CLANGx64 and MSVC22x64. So it seems to work here. (Windows10 x64, Upp build 16332)

So far I have read, the purpose of the initialization list is to initialize variables before any code is run and before the vtables are built.
This is why sometimes lists must be used, so the explanation in my book.

So I believe your example should work as intended.
I modified your example from reference to pointer and it runs equally well.
A pointer in theory can be initialized to an address where the target object does not yet exist (not recommended, but possible)
Now, a reference is essentially a constant self-dereferencing pointer that cannot be modified and must be initialized.
Here my pointer version:

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

using namespace Upp;

class A{
public:
 Array<int> *array;
 A(Array<int> &array_) : array(&array_){
  Cout() << "Array Initial item count in A = " << array->GetCount() << "\n";
 }
};
```

```
class B : public A{
public:
 Array<int> b_array;
 B() : A(b_array){
  Cout() << "Array Initial item count in B = " << b_array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
 Cout() <<"Fertig\n";
 Sleep(1000000);
}
```

Subject: Re: Order of member initialization
Posted by peterh on Tue, 20 Sep 2022 14:47:29 GMT
View Forum Message <> Reply to Message

I must correct myself, it doesnt work.
This code:

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

using namespace Upp;

class A{
public:
 Array<int> *array;
 A(Array<int> &array_) : array(&array_){
  Cout() <<"construct A\n";
  Cout() << "Array Initial item count in A = " << array->GetCount() << "\n";
 }
};

class B : public A{
public:
 Array<int> b_array;
 void *p=&b_array;
 B() : A(b_array){
  Cout() <<"construct B\n";
  Cout() << "Array Initial item count in B = " << b_array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
```

```
 Cout() <<"Fertig\n";
 Sleep(1000000);
}
```
produces this output:
```
construct A
Array Initial item count in A = 107746496
construct B
Array Initial item count in B = 0
Fertig
```

So constructor A is executed, before B was constructed.

It should however work, if A doesnt use or touch the referenced array before it was constructed.

Subject: Re: Order of member initialization
Posted by Tom1 on Tue, 20 Sep 2022 15:05:25 GMT
View Forum Message <> Reply to Message

peterh wrote on Tue, 20 September 2022 17:47I must correct myself, it doesnt work.
This code:
```
#include <Core/Core.h>

using namespace Upp;

class A{
public:
 Array<int> *array;
 A(Array<int> &array_) : array(&array_){
  Cout() <<"construct A\n";
  Cout() << "Array Initial item count in A = " << array->GetCount() << "\n";
 }
};

class B : public A{
public:
 Array<int> b_array;
 void *p=&b_array;
 B() : A(b_array){
  Cout() <<"construct B\n";
  Cout() << "Array Initial item count in B = " << b_array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
 Cout() <<"Fertig\n";
```

```
 Sleep(1000000);
}
```
produces this output:
```
construct A
Array Initial item count in A = 107746496
construct B
Array Initial item count in B = 0
Fertig
```

So constructor A is executed, before B was constructed.

It should however work, if A doesnt use or touch the referenced array before it was constructed.

Hi Peter,

Yes, it does work when the constructor of A does not use or pass the array on to be used by others. However, this may get difficult to control when projects are large.

Best regards,

Tom

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Tue, 20 Sep 2022 15:06:48 GMT
View Forum Message <> Reply to Message

I found out the same thing a while ago and then it made sense to me. The base class is initialized first, then the initializers. I think what I ended up doing to get around that, and I still don't know if it's correct, is that I would so something like create a context, a struct holding the Array as well as a function that would return *this, I'd inherit from it instead and then then I'd pass the ref function to the constructor of the second base class which would then reference it.

Looked something like this:

```
struct BaseA {
  Array<int> a;
  auto& BaseARef() { return *this; }
};

struct BaseB {
  BaseA& baseA;

  BaseB(BaseA& baseA) : baseA(baseA) {
    // do something with baseA.a;
  }
```

```
};

struct Whatever : BaseA, BaseB {
   Whatever() : BaseA(), BaseB(BaseARef()) {}
};
```

It was weird and I'm not sure if it's sound - but it did work, and if anyone knows the compiler better, feel free to let me know.

Subject: Re: Order of member initialization
Posted by peterh on Tue, 20 Sep 2022 15:48:45 GMT
View Forum Message <> Reply to Message

I cannot discuss this, multiple inheritance is too advanced for me.

This however seems to work.
The array is allocated by new and so it is constructed before all classes are constructed and instantiated.

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

using namespace Upp;

class A{
public:
 Array<int> &array;
 A(Array<int> &array_) : array(array_){
  Cout() <<"construct A\n";
  array.SetCount(0);
  Cout() << "Array Initial item count in A = " << array.GetCount() << "\n";
 }
};

class B : public A{
public:
 Array<int> &b_array;
 B() : A(*(new Array<int>)),b_array(A::array){
  Cout() <<"construct B\n";
  Cout() << "Array Initial item count in B = " << b_array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
 Cout() <<"Fertig\n";
 Sleep(1000000);
```

}

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Tue, 20 Sep 2022 16:27:50 GMT
View Forum Message <> Reply to Message

Quote:
I cannot discuss this, multiple inheritance is too advanced for me.

I hear you.

It got even crazier than that though. I realized I could also combine it with variadic templates and came up with this monstrosity:

```
#include <iostream>

using namespace std;

struct BaseA {
  int a { 100 };
  int b { 200 };
  int c { 300 };

  auto& BaseARef() { return *this; }
};

struct BaseB {
  BaseA& baseA;

  BaseB(BaseA& baseA) : baseA(baseA) {
    std::cout << baseA.a << "\n";
  }
};

struct BaseC {
  BaseA& baseA;

  BaseC(BaseA& baseA) : baseA(baseA) {
    std::cout << baseA.b << "\n";
  }
};

struct BaseD {
  BaseA& baseA;
```

```
  BaseD(BaseA& baseA) : baseA(baseA) {
     std::cout << baseA.c << "\n";
  }
};

template<typename ...Args>
struct Whatever : BaseA, Args... {
  Whatever() : Args(BaseARef())... {}
};

using Test = Whatever<BaseB, BaseC, BaseD>;

int main(void) {
   Test whatever;
}
```

which prints:

```
100
200
300
```

I mean it's kind of neat because you can use one struct as a data holder for the other ones that all of them have access to but I'm unsure whether I'd use it in production code. The other thing that's cool about it is you can kind of change the initialization order by swapping their positions around in the using statement:

```
using Test = Whatever<BaseD, BaseC, BaseB>;
```

prints

```
300
200
100
```

instead

Subject: Re: Order of member initialization
Posted by peterh on Tue, 20 Sep 2022 20:59:16 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Tue, 20 September 2022 11:19Hi,
Is there a way to re-order the initialization of class B so that the array gets initialized before class A?

So far I have read, the order of initialization is partially implementation dependent and not defined in C++.
There are some rules, but I do not know exactly yet.

I believe the problem is: The array is not constructed at all because it is never accessed within the scope, only its address is taken.
When it is constructed without arguments, then it should be initialized as an empty array.

When the array is initialized, it is constructed.

This seems to work:

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

using namespace Upp;

class A{
public:
 Array<int> &array;
 A(Array<int> &array_) : array(array_){
  Cout() << "constructing A\n";
  Cout() << "Array Initial item count in A = " << array.GetCount() << "\n";
 }
};

class B : public A{
public:
 Array<int> array;
 B() : A(array), array({}) { //<-------Initialize here
  Cout() << "constructing B\n";
  Cout() << "Array Initial item count in B = " << array.GetCount() << "\n";
 }
};

CONSOLE_APP_MAIN{
 B b;
	Sleep(1000000);
}
```

Subject: Re: Order of member initialization
Posted by Tom1 on Wed, 21 Sep 2022 07:55:16 GMT
View Forum Message <> Reply to Message

Hi,

Peter, this does not work for me:

```
 B() : A(array), array({}) { //<-------Initialize here
```

It still gives uninitialized array in DEBUG mode. But this is expected.

John, your intermediate class or struct is really an interesting idea and it surely can do the trick. However, I really need to fix (simplify) my own code to be able to read and understand it even after years have passed.

Thanks to both of you!

Best regards,

Tom

Subject: Re: Order of member initialization
Posted by peterh on Wed, 21 Sep 2022 09:49:29 GMT
View Forum Message <> Reply to Message

Yes, I did not stop testing after I wrote this and discovered problems.

Did you see my previous example some messages above, where the array is allocated from Heap?

```
class B : public A{
public:
 Array<int> &b_array;
 B() : A(*(new Array<int>)),b_array(A::array){
       .........
       }
```

This works securely, I believe, because the array is constructed and initialized to an empty array when it is allocated.
This ensures the array is constructed, before A and B are constructed.

I do however not know if C++ guarantees that A is constructed before B. This could be different with different compilers.
I do not find much about this in my books and online, and this could mean it is implementation and compiler vendor dependant.
However, if this can be guaranteed, then this should be a clean solution.

The most clean way is probably: create an empty array and give it to the B::B(....) constructor as an argument. Then B has no need to create and initialize the array.

Subject: Re: Order of member initialization
Posted by peterh on Wed, 21 Sep 2022 10:37:35 GMT
View Forum Message <> Reply to Message

I find this answer from stackoverflow very clear:

https:// stackoverflow.com/questions/2517050/c-construction-and-initi alization-order-guarantees

I hope it is correct.
When this is true then my example using "new Array<int>" should solve the given problem.
Of course, this requires a "delete" in the destructor too.

Subject: Re: Order of member initialization
Posted by peterh on Wed, 21 Sep 2022 13:11:03 GMT
View Forum Message <> Reply to Message

I think there comes clarity into this when we overwrite the array default constructor.
If DArray is used instead Array, then it becomes visible, when the array is constructed.

```
#define NL "\n"

template <typename T>
class DArray:public Array<T>{
 public:
 DArray(){Cout() << "Array constructed" << NL;}
};
#define Array DArray
```

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Wed, 21 Sep 2022 15:42:20 GMT
View Forum Message <> Reply to Message

Derp, I just realize I should have attempted to solve your problem instead of trying to just discuss order of member initialization.

OK as far as I know order of member initialization is done in the order it appears in the class, so

```
struct Test {
    int a;
    int b;
```

```
    int c;

    Test() : a(), b(a), c(b) {
    }
};
```

would be proper. What would be improper is if you had them appear in a different order:

```
struct Test {
    int c;
    int a;
    int b;

    Test() : a(), b(a), c(b) {
    }
};
```

so it's incumbent upon you once you use chains of dependencies in initializer lists that you don't go moving your member variables around without considering how they'll affect the initializers.

As for how to do the class you initially asked about, which I should have answered but got carried away, try this:

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

using namespace Upp;

class A {
public:
    A(Array<int>&& arr) : array { pick(arr) } {
        Cout() << "Array Initial item count in A = " << array.GetCount() << "\n";
        Cout() << array << EOL;
    }

protected:
    Array<int> array;
};

class B : public A {
public:
    B() : A { Array<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } } {
        Cout() << "Array Initial item count in B = " << array.GetCount() << "\n";
        Cout() << array << EOL;
    }
```

```
};

CONSOLE_APP_MAIN {
   B();
}
```

Pick semantics with move constructor will transfer ownership of the contents of the initialized array into the base array. and they'll be available in the derived class because the base was initialized first.

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Wed, 21 Sep 2022 17:00:08 GMT
View Forum Message <> Reply to Message

peterh Quote:
I find this answer from stackoverflow very clear:

https://stackoverflow.com/questions/2517050/c-construction-a nd-initialization-order-guarantees

I hope it is correct.

with inherited classes, that's what I thought too (that it proceeded left to right), but I didn't check.

https://en.cppreference.com/w/cpp/language/eval_order

Looks like they call them sequence points and they proceed in order - here's initialization lists:

Quote:
10) In list-initialization, every value computation and side effect of a given initializer clause is sequenced before every value computation and side effect associated with any initializer clause that follows it in the brace-enclosed comma-separated list of initializers.

I should acquaint myself with this stuff better, a lot of things have been changing and what was formerly Undefined Behavior is being addressed. I just learned by accident a few months ago that in C++17 there is type inference now even for template types so you no longer have to write:

```
Vector<int>    x { 1, 2, 3 };
Vector<String> y { "hello", "world" };
```

you can just write:

Vector x { 1, 2, 3 };
Vector y { "hello", "world" };

and it will fill in the type for you.

Subject: Re: Order of member initialization
Posted by peterh on Thu, 22 Sep 2022 22:26:46 GMT
View Forum Message <> Reply to Message

[/quote]

with inherited classes, that's what I thought too (that it proceeded left to right), but I didn't check.

https://en.cppreference.com/w/cpp/language/eval_order

Looks like they call them sequence points and they proceed in order - here's initialization lists:

Quote:
10) In list-initialization, every value computation and side effect of a given initializer clause is sequenced before every value computation and side effect associated with any initializer clause that follows it in the brace-enclosed comma-separated list of initializers.

Oh my god, I am german spoken and this is hard to understand in english.
The DeepL transator, (which is usually very good) makes it completely confused.

I hope this translates to this semantically:
1) A sequence point is a finalizing point where all preceding calculations and initializations are completely processed.
2) An initializer list is strictly evaluated from left to right, and each bracketed,comma separated initializer term ends with a sequencing point.

I hope I am not in error ; (I hope and assume this, because it makes some sense for me)

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Thu, 22 Sep 2022 22:53:39 GMT
View Forum Message <> Reply to Message

peterh wrote on Fri, 23 September 2022 00:26
Oh my god, I am german spoken and this is hard to understand in english.
The DeepL transator, (which is usually very good) makes it completely confused.

I hope this translates to this semantically:
1) A sequence point is a finalizing point where all preceding calculations and initializations are completely processed.
2) An initializer list is strictly evaluated from left to right, and each bracketed initializer term ends with a sequencing point.

I hope I am not in error ; (I hope and assume this, because it makes some sense for me)

No kidding, it's pretty hard to follow the whole thing, but your understanding agrees with my own.
But I'm left with other questions.
It says "brace-enclosed", so I wonder if paren-enclosed follow the member order in the class still.
The other thing I wonder about is if it also applies to brace-enclosed designated initializers e.g.

```
a = {
  .b = 1,
  .c = 2,
  .d = 3
};
```

One might ask where the problem comes from, so here's a pathological example:

```
a = {
  .b = 1,
  .c = 2,
  .d = a.b
};
```

The compiler might let you get away with that, but from what I'm told it's undefined behavior.

Subject: Re: Order of member initialization
Posted by peterh on Thu, 22 Sep 2022 23:13:53 GMT
View Forum Message <> Reply to Message

I do not know these constructs, sorry.
Is this similar to pascals "with"?:

```
with a do begin
 b=1;
 c=2;
 d=3;
end;
```

Anyway this is too advanced for me I would not write this and I have nowhere read this until now.
;-/

Subject: Re: Order of member initialization
Posted by jjacksonRIAB on Thu, 22 Sep 2022 23:22:14 GMT
View Forum Message <> Reply to Message

peterh wrote on Fri, 23 September 2022 01:13I do not know these constructs, sorry.
Is this similar to pascals "with"?:

```
with a do begin
 b=1;
 c=2;
 d=3;
end;
```

Anyway this is too advanced for me I would not write this and I have nowhere read this until now.
;-/

It's a kindof newish feature called designated initializers. If you have a struct

```
struct a {
  int b {};
  int c {};
  int d {};
};
```

before these initializers you'd have to bracket initialize it as

```
a = { 1, 2, 3 };
```

but now you can use the member names.

```
a = { .b = 1, .c = 2, .d = 3 };
```

so it looks a bit cleaner. I haven't programmed Pascal in a considerable amount of time but I'm assuming the keyword 'with' enters the scope which is very similar but likely more powerful.

Subject: Re: Order of member initialization
Posted by peterh on Thu, 22 Sep 2022 23:33:43 GMT

The "with" statement in pascal is handy to access many fields of the same record but not handy if fields are copied between records and risky if variables with the same name as the field names exists. So it is not loved too much even by pascal programmers.

I have to look into the syntax you demonstrated, this looks better.