J@ArangoDB

{ "subject" : "ArangoDB", "tags": ["multi-model", "nosql", "database"] }

C++ Constructors and Memory Leaks

Nov 18th, 2015

Preventing leaks in throwing constructors

The easiest way to prevent memory leaks is to create all objects on the stack and not using dynamic memory at all. However, often this is not possible, for example because stack size is limited or objects need to outlive the caller's scope.

Another way to prevent memory leaks and leaks of other resources is obviously to employ the RAII pattern. How can it be used safely and easily in practice, so memory leaks can be avoided?

This post will start with a few seemingly working but subtly ill-formed techniques that a few common pitfalls. Later on it will provide a few very simple solutions for getting it right.

None of the solutions here are new or original.

I took some inspiration from the excellent constructor failures GotW post. That doesn't

cover smart pointers and is not explicitly about preventing preventing memory leak, so I put together this overview myself.

Naive implementation

Let's pretend we have a simple test program main.cpp, which creates an object of class *MyClass* on the stack like this:

main.cpp

```
1  #include <iostream>
2  #include "MyClass.h"
3
4  int main () {
5    try {
6      MyClass myClass;
7      std::cout << "NO EXCEPTION" << std::endl;
8    }
9    catch (...) {
10     std::cout << "CAUGHT EXCEPTION" << std::endl;
11   }
12 }
```

The above code creates the *myClass* instance on the stack, so itself will not leak any memory. When the creating of the *myClass* instance fails for whatever reason, the instance newer existed so the memory for holding a *MyClass* object will be freed automatically. If object creation succeeds and the object goes out of scope at the end of the *try* block, then the object's destructor will be called and resources can be freed, too.

Obviously this is already good, so let's keep it as it is and have a look at the implementation of *MyClass* now. This class will manage two heap objects of type *A*, which are created using the helper function *createInstance*:

MyClass.h

```
1  #include <iostream>
2  #include "A.h"
3
4  struct MyClass {
5    A* a1;
6    A* a2;
7
8    MyClass ()
9      : a1(createInstance()),
10       a2(createInstance()) {
```

```
11
12   std::cout << "CTOR MYCLASS" << std::endl;
13  }
14
15  ~MyClass () {
16   std::cout << "DTOR MYCLASS" << std::endl;
17   delete a1;
18   delete a2;
19  }
20 };
```

For completeness, here is class *A*. It won't manage any resources itself:

A.h

```
1  #include <iostream>
2
3  struct A {
4   A () {
5    std::cout << "CTOR A" << std::endl;
6   }
7   ~A () {
8    std::cout << "DTOR A" << std::endl;
9   }
10 };
11
12 // helper method for creating an instance of A
13 A* createInstance (bool shouldThrow = false) {
14  if (shouldThrow) {
15   throw "THROWING AN EXCEPTION";
16  }
17  return new A;
18 }
```

During this complete post, the code of *A.h* will remain unchanged.

Compiling and running the initial version of main.cpp will produce the following output:

output of naive implementation

```
1 CTOR A
2 CTOR A
3 CTOR MYCLASS
4 NO EXCEPTION
5 DTOR MYCLASS
6 DTOR A
7 DTOR A
```

Valgrind also reports no memory leaks. Are we done already?

Introducing exceptions

No, because everything still went well. Let's introduce exceptions into the picture and check what happens then.

Let's first introduce an exception in the constructor of *MyClass*. We'll make the *createInstance* function throw on second invocation (we do this by passing a value of *true* to it):

constructor throwing an exception

```
1 MyClass ()
2   : a1(createInstance()),
3     a2(createInstance(true)) {
4
5   std::cout << "CTOR MYCLASS" << std::endl;
6 }
```

Running the program will now emit the following:

output of naive implementation, with exception

```
1 CTOR A
2 CAUGHT EXCEPTION
```

As we're throwing in the initializer list already, we don't even reach the constructor body. This is no problem, but worse is that the destructor for class *MyClass* is not being called at all. Valgrind therefore reports the memory for first *A* instance as leaked.

By the way, the destructor for the *MyClass* instance is intentionally not being called as the object hasn't been fully constructed and logically never existed.

Will it help if we move the heap allocations from the initializer list into the constructor body like this?

using the constructor body instead of the initializer list

```
1 MyClass () {
2   std::cout << "CTOR MYCLASS" << std::endl;
3   a1 = createInstance();
4   a2 = createInstance(true);
```

```
5 }
```

Unfortunately not. Still no destructor invocations:

output of constructor body variant

```
1 CTOR MYCLASS
2 CTOR A
3 CAUGHT EXCEPTION
```

Remember: an object's destructor won't be called if its constructor threw and the exception wasn't caught. That also means releasing an object's resources solely via the destructor as in implementation above will not be sufficient if resources are allocated in the constructor and the constructor can throw.

What can be done about that?

Obviously all resource allocations can be moved into the constructor body so exceptions can be caught there:

catching exceptions in constructor of MyClass

```
1  MyClass () {
2    std::cout << "CTOR MYCLASS" << std::endl;
3    a1 = createInstance();
4
5    try {
6      a2 = createInstance(true);
7    }
8    catch (...) {
9      // must clean up a1 to prevent a leak
10     delete a1;
11     // and re-throw the exception
12     throw;
13   }
14 }
```

While the above will work, it's clumsy, verbose and error-prone. If more objects need to be managed this will make us end up in deeply nested try...catch blocks.

try...catch for the initializer list

But wait, wasn't there a try...catch feature especially for initializer list code? Sounds like it

could be useful. Maybe we can use this instead so we can catch exceptions during initialization?

There is indeed something like that: exceptions thrown from the initializer list can be caught using the following special syntax:

catching exceptions thrown in the initializer list

```
1  MyClass ()
2    try : a1(createInstance()),
3          a2(createInstance(true)) {
4
5      std::cout << "CTOR MYCLASS" << std::endl;
6    }
7    catch (...) {  // catch block for initializer list code
8      std::cout << "CATCH BLOCK MYCLASS" << std::endl;
9      delete a1;
10   }
```

Running the program with the above *MyClass* constructor will also do what is expected: when creating the second *A* instance, the initializer list code will throw, invoking its catch block. Again code execution won't make it into the constructor body, and we don't see the destructor code in action.

The output of the program is:

output of initializer list variant

```
1 CTOR A
2 CATCH BLOCK MYCLASS
3 DTOR A
4 CAUGHT EXCEPTION
```

Valgrind does not report a leak, so are we done now?

No, as the above code has a severe problem. It worked only because we knew the second invocation of *createInstance* would fail.

But in the general case, either the first call or the second call can fail. If the first call fails, then the initializer hasn't initialized any of the object's members, and it would be unsafe to delete any object members in the initializer's catch block. If the second *createInstance* call fails, then the initializer has created *a1* but not *a2*. To prevent a leak in this case, we should delete *a1*, but we better don't delete *a2* yet.

But how do we tell in the catch block at what stage the initializer list had thrown? There is no natural way to do this correctly without introducing more state. And without that, we have the choice between undefined behavior when deleting the not-yet-initialized object members, and memory leaks when ignoring them.

Not using pointers at all

Note that if we wouldn't have used pointers for our managed *A* objects, then we could have used the fact that destructors for all initialized object members are actually called when object construction fails.

However, simple pointers don't have a destructor, so the objects they point to remain and the memory is lost.

So one obvious solution for preventing memory leaks is to not use pointers, and get rid of all `new` and `delete` statements.

In some situations we can probably get away with making the managed objects regular class members of the class that manages them:

not using pointers

```
1  struct MyClass {
2    A a1; // no pointer anymore!
3    A a2; // no pointer anymore!
4
5    MyClass ()
6      : a1(),
7        a2() {
8
9      std::cout << "CTOR MYCLASS" << std::endl;
10   }
11
12   ~MyClass () {
13     std::cout << "DTOR MYCLASS" << std::endl;
14     // no delete statements needed anymore!
15   }
16 };
```

Now if any of the *A* constructors will throw an exception during initialization, everything will be cleaned up properly. Now we can make use of the destructor of *A*. If *A* instances are not pointers but regular objects, the destructors for already created instances will be called normally, and no destructors will be called for the not-yet-initialized *A* instances. That's how

it should be. We don't get this benefit with regular pointers, which don't have a destructor.

As an aside, we got rid of the `delete` statements in the destructor and may even get away with the default destructor.

Obviously this is an easy and safe solution, but it also has a few downsides. Here are a few (incomplete list):

- when compiling *MyClass*, the compiler will now need to know the definition for class *A*. You can't get away with a simple forward declaration for class *A* anymore as in the case when the class only contained pointers to *A*. So this solution increases the source code dependencies and coupling.
- instances of managed objects (e.g. *A*) will need to be created when the managing object (e.g. *MyClass*) is created. There is no way to postpone the object creation as in the case of when using pointers.
- in general, the lifetime of the managed objects is tied to the lifetime of the managing object. This may or may not be ok, depending on requirements.

Using smart pointers (e.g. std::unique_ptr)

In many cases the superior alternative to all the above is using one of the available smart pointer classes for managing resources.

The promise of smart pointers is that resource management becomes easier, safer and more flexible with them.

Really useful smart pointers (this excludes `std::auto_ptr`) are part of standard C++ since C++11, and to my knowledge they can be used in all C++11-compatible compilers and even in some older ones. Apart from that, smart pointers are available in Boost for a long time already.

In the following snippets, I'll be using smart pointers of type `std::unique_ptr` as it is the perfect fit for this particular problem. I won't cover `shared_ptr`, `weak_ptr` or other types of smart pointers here.

When using an `std::unique_ptr` for managing the resources of *MyClass*, the *MyClass* code becomes:

using std::unique_ptr

```
1  #include <memory>
```

```
2
3  struct MyClass {
4    std::unique_ptr<A> a1;
5    std::unique_ptr<A> a2;
6
7    MyClass () :
8      a1(createInstance()),
9      a2(createInstance(true)) {
10
11     std::cout << "CTOR MYCLASS" << std::endl;
12   }
13
14   ~MyClass () {
15     std::cout << "DTOR MYCLASS" << std::endl;
16   }
17 };
```

With a unique_ptr, we can still create resources when needed, either in the initializer list, the constructor or even later. The resources can still be created dynamically using new (as is still done by function *createInstance*). When we're not taking the resources away from the unique_ptrs, then they will free their managed objects automatically and safely. We don't need to bother with delete.

And we don't need to bother with nested try...catch blocks either. If anything goes wrong during object creation, any already assigned unique_ptrs will happily release the resources they manage in their own destructors.

It does not matter if the above code throws an exception in the first invocation of *createInstance*, in the second or not at all: in every case any allocated resources are released properly, and still there is no need for any explicit exception handling or cleanup code. This is what a smart pointer will do for us, behind the scenes.

Simply compare the following two code snippets, which both create three instances of *A* while making sure no memory will be leaked if the initialization goes wrong:

solution using smart pointers

```
1 std::unique_ptr<A> a1(createInstance());
2 std::unique_ptr<A> a2(createInstance());
3 std::unique_ptr<A> a3(createInstance());
4
5 // now do something with a1, a2, a3
6 // managed objects will be released automatically when
7 // the unique_ptrs go out of scope
8 // note: they may go out of scope unintentionally if
```

```
9 // some code below will throw an exception...
```

solution using nested try...catch blocks

```
1  A* a1 = nullptr;
2  A* a2 = nullptr;
3  A* a3 = nullptr;
4
5  a1 = new A;
6  try {
7    a2 = new A;
8    try {
9      a3 = new A;
10   }
11   catch (...) {
12     delete a2;
13     throw;
14   }
15 }
16 catch (...) {
17   delete a1;
18   throw;
19 }
20
21 // now do something with a1, a2, a3
22 // objects a1, a2, a3 will not be released automatically
23 // when a1, a2, a3 go out of scope. any user of a1, a2, a3
24 // below must make sure to release the objects when they
25 // go out of scope or when an exception is thrown...
```

Obviously the smart pointer-based solution is less verbose, but it is also safer and hard to get wrong. It is especially useful for initializing and managing dynamically allocated object members, because as we've seen most of the other ways to do this are either subtly broken or much more complex.

Apart from that, we can take the managed object from out of a `unique_ptr` and take over responsibility for managing its lifetime.

Further on the plus side, a class definition that contains `unique_ptr`s can be compiled with only forward declarations for the managed types. However, when the `unique_ptr` is a regular object member, at least the class destructor implementation will need to know the size of the managed type so it can call `delete` properly.

The downside of using smart pointers is that they may impose minimal overhead when compared to the pure pointer-based solution. However in most cases this overhead should

be absolutely negligible or even be optimized away by the compiler. It may make a difference though when compiling without any optimizations, but this shouldn't matter too much in reality.

Posted by jsteemann Nov 18th, 2015 C++, Development

Tweet

« ArangoDB-PHP driver improvements Using bind parameters in the AQL editor »

About ArangoDB

- ArangoDB homepage
- ArangoDB repository on Github

Recent Posts

- Handling Binary Data in Foxx 3.0
- How Much Memory Does an STL Container Use?
- Compiling an Optimized Version of ArangoDB
- Using the Address Sanitizer (ASAN) in ArangoDB Development
- Compiling a Debug Version of ArangoDB
- Fastest String-to-uint64 Conversion Method?

Categories

- AQL (37)
- ArangoDB (66)
- ArangoShell (9)
- Bash (2)
- C++ (15)
- Concepts (8)
- Development (14)
- ES6 (4)
- Foxx (7)
- Git (1)
- Graphs (1)
- Indexes (7)
- JavaScript (9)
- Linux (12)

- Logstash (1)
- MySQL (1)
- Node.js (1)
- Parsing (2)
- Performance (25)
- PHP (3)
- Redis (1)
- Schemas (3)
- Traversals (1)
- TravisCI (3)
- V8 (1)
- WAL (2)

Monthly Archives

- 2016 (12)
  - JUN (8)
  - JAN (4)
- 2015 (46)
- 2014 (18)

Stack Overflow