M.5 — std::move_if_noexcept

ALEX AUGUST 25, 2021

(h/t to reader Koe for providing the first draft of this lesson!)

In lesson 20.9 -- Exception specifications and noexcept, we covered the `noexcept` exception specifier and operator, which this lesson builds on.

We also covered the `strong exception guarantee`, which guarantees that if a function is interrupted by an exception, no memory will be leaked and the program state will not be changed. In particular, all constructors should uphold the strong exception guarantee, so that the rest of the program won't be left in an altered state if construction of an object fails.

The move constructors exception problem

Consider the case where we are copying some object, and the copy fails for some reason (e.g. the machine is out of memory). In such a case, the object being copied is not harmed in any way, because the source object doesn't need to be modified to create a copy. We can discard the failed copy, and move on. The `strong exception guarantee` is upheld.

Now consider the case where we are instead moving an object. A move operation transfers ownership of a given resource from the source to the destination object. If the move operation is interrupted by an exception after the transfer of ownership occurs, then our source object will be left in a modified state. This isn't a problem if the source object is a temporary object and going to be discarded after the move anyway -- but for non-temporary objects, we've now damaged the source object. To comply with the `strong exception guarantee`, we'd need to move the resource back to the source object, but if the move failed the first time, there's no guarantee the move back will succeed either.

How can we give move constructors the `strong exception guarantee`? It is simple enough to avoid throwing exceptions in the body of a move constructor, but a move constructor may invoke other constructors that are `potentially throwing`. Take for example the move constructor for `std::pair`, which must try to move each subobject in the source pair into the new pair object.

```
// Example move constructor definition for std::pair
// Take in an 'old' pair, and then move construct the new pair's 'first' and 'second' subobjects from the 'old' ones
template <typename T1, typename T2>
pair<T1,T2>::pair(pair&& old)
  : first(std::move(old.first)),
    second(std::move(old.second))
{}
```

Now lets use two classes, `MoveClass` and `CopyClass`, which we will `pair` together to demonstrate the `strong exception guarantee` problem with move constructors:

```
#include <iostream>
#include <utility> // For std::pair, std::make_pair, std::move, std::move_if_noexcept
#include <stdexcept> // std::runtime_error

class MoveClass
{
private:
  int* m_resource{};

public:
  MoveClass() = default;

  MoveClass(int resource)
    : m_resource{ new int{ resource } }
  {}

  // Copy constructor
  MoveClass(const MoveClass& that)
  {
    // deep copy
    if (that.m_resource != nullptr)
    {
      m_resource = new int{ *that.m_resource };
    }
  }

  // Move constructor
  MoveClass(MoveClass&& that) noexcept
    : m_resource{ that.m_resource }
  {
    that.m_resource = nullptr;
  }

  ~MoveClass()
  {
    std::cout << "destroying " << *this << '\n';

    delete m_resource;
  }

  friend std::ostream& operator<<(std::ostream& out, const MoveClass& moveClass)
  {
    out << "MoveClass(";

    if (moveClass.m_resource == nullptr)
    {
      out << "empty";
    }
    else
    {
      out << *moveClass.m_resource;
    }

    out << ')';

    return out;
  }
};


class CopyClass
{
public:
  bool m_throw{};

  CopyClass() = default;

  // Copy constructor throws an exception when copying from a CopyClass object where its m_throw is 'true'
  CopyClass(const CopyClass& that)
    : m_throw{ that.m_throw }
  {
    if (m_throw)
    {
      throw std::runtime_error{ "abort!" };
    }
  }
};

int main()
{
  // We can make a std::pair without any problems:
  std::pair my_pair{ MoveClass{ 13 }, CopyClass{} };

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  // But the problem arises when we try to move that pair into another pair.
  try
  {
    my_pair.second.m_throw = true; // To trigger copy constructor exception

    // The following line will throw an exception
    std::pair moved_pair{ std::move(my_pair) }; // We'll comment out this line later
    // std::pair moved_pair{std::move_if_noexcept(my_pair)}; // We'll uncomment this line later

    std::cout << "moved pair exists\n"; // Never prints
  }
  catch (const std::exception& ex)
  {
      std::cerr << "Error found: " << ex.what() << '\n';
  }

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  return 0;
}
```

The above program prints:

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(empty)
destroying MoveClass(empty)
```

Let's explore what happened. The first printed line shows the temporary `MoveClass` object used to initialize `my_pair` gets destroyed as soon as the `my_pair` instantiation statement has been executed. It is `empty` since the `MoveClass` subobject in `my_pair` was move constructed from it, demonstrated by the next line which shows `my_pair.first` contains the `MoveClass` object with value `13`.

It gets interesting in the third line. We created `moved_pair` by copy constructing its `CopyClass` subobject (it doesn't have a move constructor), but that copy construction threw an exception since we changed the Boolean flag. Construction of `moved_pair` was aborted by the exception, and its already-constructed members were destroyed. In this case, the `MoveClass` member was destroyed, printing `destroying MoveClass(13) variable`. Next we see the `Error found: abort!` message printed by `main()`.

When we try to print `my_pair.first` again, it shows the `MoveClass` member is empty. Since `moved_pair` was initialized with `std::move`, the `MoveClass` member (which has a move constructor) got move constructed and `my_pair.first` was nulled.

Finally, `my_pair` was destroyed at the end of main().

To summarize the above results: the move constructor of `std::pair` used the throwing copy constructor of `CopyClass`. This copy constructor threw an exception, causing the creation of `moved_pair` to abort, and `my_pair.first` to be permanently damaged. The `strong exception guarantee` was not preserved.

std::move_if_noexcept to the rescue

Note that the above problem could have been avoided if `std::pair` had tried to do a copy instead of a move. In that case, `moved_pair` would have failed to construct, but `my_pair` would not have been altered.

But copying instead of moving has a performance cost that we don't want to pay for all objects -- ideally we want to do a move if we can do so safely, and a copy otherwise.

Fortunately, C++ has a two mechanisms that, when used in combination, let us do exactly that. First, because `noexcept` functions are no-throw/no-fail, they implicitly meet the criteria for the `strong exception guarantee`. Thus, a `noexcept` move constructor is guaranteed to succeed.

Second, we can use the standard library function `std::move_if_noexcept()` to determine whether a move or a copy should be performed. `std::move_if_noexcept` is a counterpart to `std::move`, and is used in the same way.

If the compiler can tell that an object passed as an argument to `std::move_if_noexcept` won't throw an exception when it is move constructed (or if the object is move-only and has no copy constructor), then `std::move_if_noexcept` will perform identically to `std::move()` (and return the object converted to an r-value). Otherwise, `std::move_if_noexcept` will return a normal l-value reference to the object.

Key insight

`std::move_if_noexcept` will return a movable r-value if the object has a noexcept move constructor, otherwise it will return a copyable l-value. We can use the `noexcept` specifier in conjunction with `std::move_if_noexcept` to use move semantics only when a strong exception guarantee exists (and use copy semantics otherwise).

Let's update the code in the previous example as follows:

```
//std::pair moved_pair{std::move(my_pair)}; // comment out this line now
std::pair moved_pair{std::move_if_noexcept(my_pair)}; // and uncomment this line
```

Running the program again prints:

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(13)
destroying MoveClass(13)
```

As you can see, after the exception was thrown, the subobject `my_pair.first` still points to the value `13`.

The move constructor of `std::pair` isn't `noexcept` (as of C++20), so `std::move_if_noexcept` returns `my_pair` as an l-value reference. This causes `moved_pair` to be created via the copy constructor (rather than the move constructor). The copy constructor can throw safely, because it doesn't modify the source object.

The standard library uses `std::move_if_noexcept` often to optimize for functions that are `noexcept`. For example, `std::vector::resize` will use move semantics if the element type has a `noexcept` move constructor, and copy semantics otherwise. This means `std::vector` will generally operate faster with objects that have a `noexcept` move constructor (reminder: move constructors are `noexcept` by default, unless they call a function that is `noexcept(false)`).

Warning

If a type has both potentially throwing move semantics and deleted copy semantics (the copy constructor and copy assignment operator are unavailable), then `std::move_if_noexcept` will waive the strong guarantee and invoke move semantics. This conditional waiving of the strong guarantee is ubiquitous in the standard library container classes, since they use std::move_if_noexcept often.

Next lesson
M.6 std::unique_ptr

Back to table of contents

Previous lesson
M.4 std::move

23 COMMENTS Newest

Tejero Joshua October 14, 2021 3:58 am

you said in the previous lesson that when an exception is thrown inside a class it destroys all member variable/function and doesn't call the class destructor.

but why on the output of 2nd-panel of code, the destructor of MoveClass is still called when the exception is thrown inside the CopyClass constructor?

Last edited 4 months ago by Tejero Joshua

0 Reply

Alex Author — Reply to Tejero Joshua — October 14, 2021 12:59 pm

I said when a constructor throws an exception, it doesn't call the class destructor (because the object technically isn't created until the constructor completes).

1 Reply

Tejero Joshua — Reply to Alex — October 17, 2021 6:21 pm

many thanks Alex.

0 Reply

Muralidhara September 27, 2021 10:08 pm

Though i have not made the move assignment constructor not throwing (noexcept) and using std::move_if_noexcept(a) to move the object, still move assignment constructor is called instead of copy assignment constructor !
What is wrong in the below code ?

#include <iostream>
#include <stdexcept>

using namespace std;

class A
{
public:
A(){cout << "In A() \n";}

A& operator=(const A& a)
{
cout << "In COPY A() ==== \n";

return *this;
}
A& operator=(A&& a)
{
cout << "In MOVE A() ==== \n";

return *this;
}
};

int main()
{
cout << "Hello World \n ";
A a{};
A b{};
b = std::move_if_noexcept(a);

return 0;
}

0 Reply

geoff August 22, 2021 2:34 pm

Small typo at the end of "The move constructors exception problem", in the 2nd sentence of this paragraph:
To summarize the above results: the move constructor of std::pair used the throwing copy constructor of CopyClass. This copy constructor throw an exception, causing the creation of moved_pair to abort, and my_pair.first to be permanently damaged. The strong exception guarantee was not preserved.

throw should "threw".

0 Reply

Nick June 25, 2021 9:14 am

The code example #includes <string> and <string_view> but these don't seem to be necessary.

0 Reply

Sam April 12, 2021 3:26 am

Does this mean we should always prefer std::move_if_no_except over std::move?

0 Reply

Alex Author — Reply to Sam — April 12, 2021 10:39 am

Great question. The author of a comment on https://stackoverflow.com/questions/28598488/stdmove-if-noexcept-calls-copy-assignment-even-though-move-assignment-is-noexc#:~:text=std%3A%3Amove%20is%20an,we%20are%20assigning%20to%20them states, "std::move is an unconditional cast to an rvalue-reference, whereas std::move_if_noexcept depends on the ways an object can be move-constructed - therefore it should only be used in places where we are actuallying constructing objects, not when we are assigning to them."

So it looks like the answer is no.

0 Reply

Mymoonsvabo — Reply to Alex — May 13, 2021 7:52 am

Hi,
If I understood correctly, the lesson says that std::move_if_noexcept will basically just check the argument's (in this case, my_pair) move constructor for the noexcept specifier.
Std::pair's move constructor defined at the top does not have the noexcept specifier, but the lesson states that the problem is in the fact that CopyClass doesn't have a noexcept move constructor.
Can you please clarify?
Shouldn't checking std::pair's constructors be the only check?
Does this mean that std::move_if_noexcept "digs" deeper into the argument class, beyond its constructors, until it finds out if all member objects' move constructor are noexcept?

1 Reply

Alex Author — Reply to Mymoonsvabo — May 17, 2021 8:36 am

You are correct -- std::pair's move constructor is not noexcept, so that is why we get a copy. Lesson updated.

0 Reply

Sam, — Reply to Alex — April 12, 2021 7:23 pm

Thanks for the quick response Alex!

0 Reply

Tomek February 16, 2021 7:07 am

The first lesson where I understand nothing

1 Reply

Mrp1 — Reply to Tomek — March 11, 2021 12:07 am

Same, man.

1 Reply

Rishi — Reply to Mrp1 — August 12, 2021 6:28 pm

Me too, but then I tried again 2 to 3 times and got it :)

Last edited 6 months ago by Rishi

1 Reply

Parul January 10, 2021 4:22 am

If a type has both potentially throwing move semantics and deleted copy semantics (the copy constructor and copy assignment operator are unavailable), then std::move_if_noexcept will waive the strong guarantee and invoke move semantics. This conditional waiving of the strong guarantee is ubiquitous in the standard library container classes, since they use std::move_if_noexcept often.

In this case exception will be thrown or not?

0 Reply

Rishi — Reply to Parul — August 12, 2021 6:29 pm

I think yes, the exception will be thrown and the source will be left broken

0 Reply

Marat September 8, 2020 8:02 am

I noticed that you specify move constructors as noexcept in previous lessons, but it's missing here.

0 Reply

Dong September 4, 2020 5:30 pm

- I run this example in Code::Block, The result shows an error "missing template arguments before my_pair"

```
std::pair my_pair{ MoveClass{ 13 }, CopyClass{} };
```

I do not understand what happens? Can you explain this problem? Thanks for you advices.

0 Reply

nascardriver Sub-admin — Reply to Dong — September 5, 2020 1:35 am

You need at least C++17 to run this example. Before that, you had to explicitly specify the template arguments

```
std::pair my_pair/* ... */
```

0 Reply

Fan — Reply to nascardriver — February 16, 2021 11:46 am

So that is deduction guides in action? Is there a lesson on that?

0 Reply

nascardriver Sub-admin — Reply to Fan — February 17, 2021 8:19 am

Yes that's template deduction. There's no lesson for it.

0 Reply

koe August 22, 2020 10:45 am

<3 thanks :)

typos:

- "Example move constructor declaration" -> definition
- the copy constructor of MoveClass doesn't need a self-assignment check
- "both a potentially throwing move semantics" -> /a\

0 Reply

nascardriver Sub-admin — Reply to koe — August 22, 2020 10:29 pm

Thanks, all fixed ;)

0 Reply