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:

```cpp
#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

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coo

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X