std::move()

A mini-lecture series

CSE498 Collaborative Design (W) - Secure and Efficient C++ Software Development

04/02/2025

Kira Chan

https://cse.msu.edu/~chanken1/

Move semantics vs value semantics

- When you use an assignment operator, you *copy* over the values
- This is called **value semantics**
- int *b* = 20
- int *a* = b
- // both *a* and *b* are set to 20, each with their own copy

- What if you don't need *b* anymore?
  - If *b* is a really large object, then we just wasted time copying over the resource

Std::move()

- Std::move() allows you to take ownership of an object
- It effectively sets an L-value into an X-value
  - Recall L-value is one with a memory address or has a name
  - X-value is one that is “eXpiring”, and thus will disappear when out of scope
- Basically, same as:

  (Object&&) m_obj;
  - This is called an r-value reference

C: > msys64 > ucrt64 > include > c++ > 14.2.0 > bits > C move.h > {} std > move<_Tp>(_Tp &&)

```
117
118   /**
119    *  @brief  Convert a value to an rvalue.
120    *  @param  __t  A thing of arbitrary type.
121    *  @return The parameter cast to an rvalue-reference to allow moving it.
122    */
123   template<typename _Tp>
124     _GLIBCXX_NODISCARD
125     constexpr typename std::remove_reference<_Tp>::type&&
126     move(_Tp&& __t) noexcept
127     { return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
128
```

Move constructor

```
Object(Object&& other) noexcept
{
    // take other.size if exist
    m_data = other.m_data; // yoink their data
    Other.m_data = nullptr; // set their data to be nullptr, otherwise?
}
```

Example use case

- “Swap value of var A and var B”

- Void swap(a, b)
  - temp = a
  - a = b
  - b = temp

- Void swap_move(a, b)
  - temp = std::move(a)
  - a = std::move(b)
  - b = std::move(temp)

Move constructor and assignment operator

- We learned how to do the constructor
  - This is all sunny for construction of an object
  - Object n_obj = std::move(o_obj)
  - Is the same as: Object n_obj(std::move(o_obj))
- But what happens when we want to just assign?
  - Only gets call if we assign a variable into an *existing* variable
    - Object n_obj = new Object()
    - n_obj = std::move(o_obj)

Need to define move assignment operator

```
//again, take in a temporary object
Object& operator=(Object&& other) noexcept {
    // Check if it is the same object
    if (this != &other) {
        //What happened to my old data? (Memory leak)
        // Need to make sure we delete the old data FIRST
        delete m_data;

        // same as move constructor
        m_data = other.m_data; // yoink their data
        Other.m_data = nullptr; // set their data to be nullptr
        }
    Return *this;
}
```

When not to use move

- Never return std::move()
  - Prevents compilers from doing Return Value Optimization (RVO), a form of **copy elision**
  - RVO is mandatory after C++17 even if you turn optimisation level to 0
  - Optimisation-level 0 doesn’t actually turn off all optimisations, just most
  - Pay a runtime overhead cost

Std::move() doesn’t actually move

- There is no actual moving of data under the hood
- The ownership of the object just transfers

References

- https://en.cppreference.com/w/cpp/utility/move
- https://www.youtube.com/watch?v=ehMg6zvXuMY
- https://www.youtube.com/watch?v=OWNeCTd7yQE
- https://stackoverflow.com/questions/3413470/what-is-stdmove-and-when-should-it-be-used
- https://stackoverflow.com/questions/3106110/what-is-move-semantics
- https://en.cppreference.com/w/cpp/language/value_category

Person of the Day
Donald Knuth

- Recipient of the 1974 ACM Turing Award
- Known for his work with analysis of algorithms
- Author TeX, a typesetting language
- Author of *The Art of Computer Programming*
- Strongly dislikes patents to trivial solutions in software