Move semantics

09 June 2021 07:07

1. ## C++ provides constructor, copy constructor, = by default if user hasn't supplied
   This copy constructor, = does shallow copy .. Even if user-defined to execute deep-copy, copying is expensive

2. ## C++ provides default move constructor, move = if class doesn't have defined copy/move methods
   default move constructor/move = does same thing as copy const/=??
   If u want move constructor/move = that does the move write it urself??

3. ## Return from function
   Return from function is executed as a copy.
   - Return obj is copy constructed into a temporary object.
   - temporary object is assigned to caller var by copy assignment

   Return from function is executed as a move.
   - Return obj is move constructed into a temporary object.
   - temporary object is assigned to caller var by move assignment

   Rationale:
   The move constructor and move assignment are called when those move functions have been defined, and the argument for construction or assignment is an r-value.

   Return by reference/address ok if the variable is not in function local stack

4. ## Move semantics
   Instead of copy, do a move/transfer of ownership.. Only one obj owns the resource b4/af move1. **Efficient than copy**

5. ## When to use copy/move semantics
   Copy
   - Construct an object / assignment where the argument is an l-value, the only thing we can reasonably do is copy the l-value.
   - We can't assume it's safe to alter the l-value, because it may be used again later in the program.

   Move
   - Construct an object / assignment where the argument is an r-value, then we know that r-value is just a temporary object of some kind.
   - Instead of copying it (which can be expensive), we can simply transfer its resources (which is cheap) to the object we're constructing or assigning.
   - This is safe to do because the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again!

6. ## Move functions should always leave both objects in a well-defined state
   I

7. ## C++ specification states **automatic objects** returned from a function by value can be moved **even if they are l-values** -> most likely when returned object's class definition has move constructs defined.
   In move-enabled classes, it is sometimes desirable to delete the copy constructor and copy assignment functions to ensure copies aren't made.

8. ## std::move(..)
   http://bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html
   - In C++11, std::move is a standard library function that casts (using static_cast) its argument into an r-value reference, so that move semantics can be invoked
   - std::move doesn't move
     The idiomatic use of std::move is to ensure the argument passed to a function is an rvalue so that you can move from it (choose move semantics). By function I mean an actual function or a constructor or an operator (e.g. assignment operator). i.e. function moves.
   - std::move() gives a hint to the compiler that the programmer doesn't need this object any more
   - Advise: std::move() to not be used on any persistent object you don't want to modify. If used state of any objects may not be the same after they are moved!
   - Trigger move versions of standard library functions if available by using std::move.
   - Working
     - First of all std::move is a template with a forwarding reference argument which means that it can be called with either a lvalue or an rvalue, and the reference collapsing rules apply.
     - Because the type T is deduced, we did not have to specify when using std::move.
     - Then all it does is a static_cast.

9. ## std::forward<typename>(..)
   https://isocpp.org/blog/2018/02/quick-q-whats-the-difference-between-stdmove-and-stdforward
   The idiomatic use of std::forward is inside a templated function with an argument declared as a forwarding reference, used to retrieve the original value category, that it was called with, and pass it on further down the call chain (perfect forwarding)

   http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2951.html
   Working
   - The type T is not deduced, therefore we had to specify it when using std::forward.
   - Then all it does is a static_cast.

```cpp
struct Y
{
  Y(){}
  Y(const Y &){ std::cout << "Copy constructor\n"; }
  Y(Y &&) noexcept { std::cout << "Move constructor\n"; }
};

struct X
{
  Y a_;
  Y b_;

  template<typename A, typename B>
  X(A && a, B && b) :
    // retrieve the original value category from constructor call
    // and pass on to member variables
    a_{ std::forward<A>(a) },
    b_{ std::forward<B>(b) }
  {
  }
};

template<typename A, typename B>
X factory(A && a, B && b)
{
  // retrieve the original value category from the factory call
  // and pass on to X constructor
  return X(std::forward<A>(a), std::forward<B>(b));
}

int main()
{
  Y y;
  X two = factory(y, Y());
  // the first argument is a lvalue, eventually a_ will have the
  // copy constructor called
  // the second argument is an rvalue, eventually b_ will have the
  // move constructor called
}

// prints:
// Copy constructor
// Move constructor
```

=`e

10. s
    I
11. F
    I
12. F
    I
13. F
    I
14. F
15. F
    I
16. F
    I
17. F
    I
18. F
    I
19. F
    I
20. F
    I
21. D
    I
22. F
    I
23. F
    I
24. F
    I
25. F
26. F
    I
27. F
    I
28. F
    I

29. F
    I
30. F
    I
31. F
    I
32. D
    I
33. F
    I
34. F
    I
35. F
    I
36. F
37. F
    I
38. F
    I
39. F
    I
40. F
    I
41. F
    I
42. F
    I
43. D
    I
44. F
    I
45. F
    I
46. F
    I
47. F
48. F
    I
49. F
    I
50. F
    I
51. F
    I
52. F
    I