Lecture 14 Move Semantics

#CS106L

- **Lvalues vs. Rvalues**
- **Move Constructor and Assignment**
- **`std::move`**
- **Swap and Insert**
- **Perfect Forwarding**

---

1. copy but can't move:

  - the object is const/reference
  - the object has a user-defined copy constructor but no move constructor(has a deleted move constructor)
  - the object is returned from a function by value and not a NRVO candidate

2. move but can't copy:

  - the object is not const
  - the object has a user-defined move constructor but no copy constructor(has a deleted copy constructor)
  - the object is returned from a function by rvalue reference
  - the object is a move-only types(e.g. `unique_ptr`, `std::future`, and `std::optional`)

3. An l-value is an expression that has a name (identity)

- can find address using address-of operator (&var)
- can appear either left or right of an assignment
- lifetime is decided by scope

4. An r-value is an expression that does not have a name (identity)

- temporary values
- cannot find address using address-of operator (&var)
- can appear only on the right of an assignment
- ends on the very next line (unless you purposely extend it!)

5. NOTICE:

   - An l-value reference can bind to an l-value.
   - An r-value reference can bind to an r-value.
   - A const l-value reference can bind to either l or r-value.
   - An r-value reference is an alias to an r-value, but the r-value reference itself is an l-value.

```
        //l-value:v1,v2,v4,ptr,val,v1[2],*ptr

        //r-value:v1+v2,&val

        auto v4 = v1 + v2;

        ptr = &val;

        v1[2] = *ptr;
```

6. Why r-values are key to move semantics?

- An object that is an l-value is NOT disposable, so you can copy from, but definitely cannot move from.(namely, if an object might potentially be reused, you cannot steal its resources)

- An object that is an r-value is disposable, so you can either copy or move from.

- there exists some objects that can’t be copied (eg. stream)

7. special member functions

- Default constructor
- Copy constructor (create new from existing l-value)
- Copy assignment (overwrite existing from existing l-value)
- Move constructor (create new from existing r-value)
- Move assignment (overwrite existing from existing r-value)
- Destructor

8. What members need to be deep copied?

Any member that are handles to external resources.

- Handles to memory (pointers), files (filestreams), locks (mutexes)

9. l or r-value?

```
        const auto&& v4 = v1 + v2;
```

v4:l-value

10. `std::move`: unconditional cast to an r-value,`std::move` itself doesn’t move anything.When declaring move operations, make sure to `std::move` all members.

Move constructor:

```
MyVector<T>(MyVector<T>&& other)
```

```
:elems(std::move(other.elems)),logicalSize(std::move(other.logicalSize)),

allocatedSize(std::move(other.allocatedSize)) {

        other.elems = nullptr;

}
```

Move assignment:

```
MyVector<T>& operator=(MyVector<T>&& rhs) {

        if (this == &rhs) {

        delete[] elems;

        allocatedSize = std::move(rhs.allocatedSize);

        logicalSize = std::move(rhs.logicalSize);

        elems = std::move(rhs.elems);

        rhs.elems = nullptr;

        }

        return *this;

}
```

11. application:

Our `push_back` function before:

```
template <typename T>

void MyVector<T>::push_back(const T& element) {

        if (size() == allocatedSize) resize(…);

        elems[logicalSize++] = element;

}
```

Our `push_back` function now supports an r-value overload:

```
template <typename T>

void MyVector<T>::push_back(T&& element) {

        if (size() == allocatedSize) resize(…);

        elems[logicalSize++] = std::move(element);

}
```

write a generic swap function:

```
template <typename T>

void swap(T& a, T& b) noexcept {

        T c(std::move(a)); // move constructor

        a = std::move(b); // move assignment

        b = std::move(c); // move assignment

}

// by the way, this is std::swap
```

12. Rule of Five:If you explicitly define (or delete) a copy constructor, copy assignment, move constructor, move assignment, or destructor,you should define (or delete) all five.

The fact that you defined one of these means, one of your members has ownership issues that need to be resolved.

Q1:`emplace_back`?

`std::vector::emplace_back()` is a member function of the `std::vector` class in C++ that is used to insert an element at the end of the vector. It is similar to the `push_back()` function, but it can construct the element in-place, which means that the constructor of element is called directly, rather than creating a copy of the element.

Here's an example of how to use `emplace_back()`:

```
#include <vector>

#include <iostream>



struct MyStruct {

    int a;

    std::string b;

    MyStruct(int _a, std::string _b): a(_a), b(_b) {}

};



int main() {

    std::vector<MyStruct> myVec;

    myVec.emplace_back(5, "hello");  // construct MyStruct in-place

    myVec.emplace_back(10, "world"); // construct MyStruct in-place



    for (const auto & elem : myVec) {

        std::cout << elem.a << " " << elem.b << std::endl;

    }

    return 0;

}
```

The `emplace_back()` function takes any number of arguments, and they are passed as arguments to the constructor of the element type. The `emplace_back()` function can be useful when the elements are expensive to copy or when it is more efficient to construct the elements in-place.

Q2:How can a compiler tell whether we are allowed to move?

- The object's type/constness

- The object's storage duration: Temporary objects and rvalue references are move-able, whereas objects with automatic storage duration are not move-able.

- The object's presence of move constructor, move assignment, copy constructor and copy assignment.

Q3:how many times is each special member function called?

```
MyVector<string> findAllWords(size_t size) {

    MyVector<string> names(size, “Ito”);

    return names;

}



int main() {

    MyVector<string> names1 = findAllWords(54321234);

    MyVector<string> names2;

    names2 = findAllWords(54321234);

}
```

Counts without copy elision:

- Copy assignment x 1
- Copy constructor x 3
- Default constructor x 1
- Destructor x 6
- Fill constructor x 2

Counts with copy elision:

- Copy assignment x 1
- Default constructor x 1
- Destructor x 3
- Fill constructor x 2

Counts with copy elision and move semantics:

- Move assignment x 1
- Default constructor x 1
- Destructor x 3
- Fill constructor x 2