MOVE SEMANTICS IN C++

Created by Łukasz Ziobroń and Kamil Szatkowski

MOVE SEMANTICS

Rationale: better optimization by avoiding redundant copies, improved safety by keeping only one instance.

New syntax elements:

- `auto && value` - r-value reference
- `Class(Class &&)` - move constructor
- `Class& operator=(Class&&)` - move assignment operator
- `std::move` auxilary function
- `std::forward` auxilary function

R-VALUE AND L-VALUE REFERENCES

```
struct A {
    int a, b;
};

A foo() {
    return {1, 2};
}

A a;                    // l-value
foo();                  // r-value
A & ra = a;             // l-value reference to l-value, OK
A & rb = foo();         // l-value reference to r-value, ERROR
A const& rc = foo();    // const l-value reference to r-value, OK (exceptic

A && rra = a;           // r-value reference to l-value, ERROR
A && rrb = foo();       // r-value reference to r-value, OK

A const ca{20, 40};
A const&& rrc = ca;     // const r-value reference to const l-value, ERROR
```

- l-value object has a name and address
- l-value object is persistent, in the next line it can be accessed by name
- r-value object does not have a name (usually) or address
- r-value object is temporary, in the next line it will not be accessible

Value categories

Each C++ expression (an operator with its operands, a literal, a variable name, etc.) is characterized by two independent properties: a *type* and a *value category*. Each expression has some non-reference type, and each expression belongs to exactly one of the three primary value categories: *prvalue*, *xvalue*, and *lvalue*.

- a glvalue ("generalized" lvalue) is an expression whose evaluation determines the identity of an object, bit-field, or function;
- a prvalue ("pure" rvalue) is an expression whose evaluation either
  - computes the value of the operand of an operator (such prvalue has no *result object*), or
  - initializes an object or a bit-field (such prvalue is said to have a *result object*). With the exception of decltype, all class and array prvalues have a result object even if it is discarded. The result object may be a variable, an object created by new-expression, a temporary created by temporary materialization, or a member thereof; (since C++17)
- an xvalue (an "eXpiring" value) is a glvalue that denotes an object or bit-field whose resources can be reused;
- an lvalue (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) is a glvalue that is not an xvalue;
- an rvalue (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is a prvalue or an xvalue.

Note: this taxonomy went through significant changes with past C++ standard revisions, see History below for details.

VALUE CATEGORIES IN C++

- lvalue
- prvalue
- xvalue
- glvalue = lvalue | xvalue
- rvalue = prvalue | xvalue

Full list at cppreference.com

Primary categories

lvalue

The following expressions are *lvalue expressions*:

- the name of a variable, a function, a template parameter object (since C++20), or a data member, regardless of type, such as `std::cin` or `std::endl`. Even if the variable's type is rvalue reference, the expression consisting of its name is an lvalue expression;
- a function call or an overloaded operator expression, whose return type is lvalue reference, such as `std::g`… `::cout << 1`, `str1 = str2`, or `++it`;
- `a = b`, `a += b`, `a %= b`, and all other built-in assignment and compound assignment expressions;
- `++a` and `--a`, the built-in pre-increment and pre-decrement expressions;
- `*p`, the built-in indirection expression;
- `a[n]` and `p[n]`, the built-in subscript expressions, where one operand in `a[n]` is an array lvalue (since C++11);
- `a.m`, the member of object expression, except where `m` is a member enumerator or a non-static member function, or where a is an rvalue and `m` is a non-static data member of non-reference type;
- `p->m`, the built-in member of pointer expression, except where `m` is a member enumerator or a non-static member function;
- `a.*mp`, the pointer to member of object expression, where a is an lvalue and `mp` is a pointer to data member;
- `p->*mp`, the built-in pointer to member of pointer expression, where `mp` is a pointer to data member;
- `a, b`, the built-in comma expression, where b is an lvalue;
- `a ? b : c`, the ternary conditional expression for some b and c (e.g., when both are lvalues of the same type, but see definition for detail);
- a string literal, such as `"Hello, world!"`;
- a cast expression to lvalue reference type, such as `static_cast<int&>(x)`;

- a function call or an overloaded operator expression, whose return type is rvalue reference to function;
- a cast expression to rvalue reference to function type, such as `static_cast<void (&&)(int)>(x)`. (since C++11)

Properties:

- Same as glvalue (below).
- Address of an lvalue may be taken: `&++i` [1] and `&std::endl` are valid expressions.
- A modifiable lvalue may be used as the left-hand operand of the built-in assignment and compound assignment operators.
- An lvalue may be used to initialize an lvalue reference; this associates a new name with the object identified by the expression.

USAGE OF MOVE SEMANTICS

```
template <typename T>
class Container {
public:
    void insert(const T& item);  // inserts a copy of item
    void insert(T&& item);       // moves item into container
};

Container<std::string> c;
std::string str = "text";

c.insert(str);              // lvalue -> insert(const std::string&)
                            // inserts a copy of str, str is used later
c.insert(str + str);        // rvalue -> insert(string&&)
                            // moves temporary into container
c.insert("text");           // rvalue -> insert(string&&)
                            // moves temporary into container
c.insert(std::move(str));   // rvalue -> insert(string&&)
                            // moves str into container, str is no longer u
```

PROPERTIES OF MOVE SEMANTICS

- Move operation transfer all data from the source to the target.
- It leaves the source object in an unknown, but safe to delete state.
- "moved-from" objects should never be used. They can only be safely destroyed or, if possible, new resource can be assigned to them (`reset()`).

```
std::unique_ptr<int> pointer1{new int{5}};
std::unique_ptr<int> pointer2 = std::move(pointer1);
*pointer1 = 4;  // Undefined behavior, pointer1 is in moved-from state
pointer1.reset(new int{20});    // OK
```

IMPLEMENTATION OF MOVE SEMANTIC

```
class X : public Base {
    Member m_;

    X(X&& x) : Base(std::move(x)), m_(std::move(x.m_)) {
        x.set_to_resourceless_state();
    }

    X& operator=(X&& x) {
        Base::operator=(std::move(x));
        m_ = std::move(x.m_);
        x.set_to_resourceless_state();
        return *this;
    }
};
```

IMPLEMENTATION OF MOVE SEMANTICS

TASK

Write your own implementation of unique_ptr

Hints:

- Template class
- RAII
- Copy operations not allowed
- Move operations allowed
- Interface functions - at least:
  - `T* get()`
  - `T& operator*()`
  - `T* operator->()`

RULE OF 3

If you define at least one of: destructor, copy constructor, copy assignemnt operator it means that you are manually managing resources and you should implement them all. It will ensure correctness in every context.

RULE OF 5

Rule of 3 can be further optimized by adding move operations - move constructor and move assignment operator. From C++11 use Rule of 5.

RULE OF 0

If you use RAII handlers, all the copy and move operations will be generated automatically. Don't implement any of special methods (destructor, move constructor, move assignment operator, copy constructor, copy assignment operator).

Eg. when you have unique_ptr as your class member, copy operations of your class will be automatically blocked, but move operations will be supported.

IMPLEMENTATION OF `std::move()` AND "UNIVERSAL REFERENCE"

```
template <typename T>
typename std::remove_reference<T>::type&& move(T&& obj) noexcept {
    using ReturnType = std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(obj);
}
```

- `T&&` as a template function parameter is not only r-value reference
- `T&&` is a "forwarding reference" or "universal reference" (name proposed by Scott Meyers)
- `T&&` in templates can bind to l-values and r-values
- `std::move()` takes any kind of reference and cast it to r-value reference

REFERENCE COLLAPSING

When a template is being instantiated reference collapsing may occur.

```
template <typename T>
void f(T& item) {
    // ...
}

void f(int& & item);   // passing int& as a param -> f(int&)
void f(int&& & item);  // passing int&& as a param -> f(int&)
```

Reference collapsing rules:

- `T& &` -> `T&`
- `T& &&` -> `T&`
- `T&& &` -> `T&`
- `T&& &&` -> `T&&`

INTERFACE BLOAT

Trying to optimize for every possible use case may lead to interface bloat.

```cpp
class Gadget;

void f(const Gadget&);
void f(Gadget&);
void f(Gadget&&);

void use(const Gadget& g) { f(g); }            // calls f(const Gadget&)
void use(Gadget& g)       { f(g); }            // calls f(Gadget&)
void use(Gadget&& g)      { f(std::move(g)); } // calls f(Gadget&&)

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);       // calls use(const Gadget&) then calls f(const Gadget&)
    use(g);        // calls use(Gadget&) then calls f(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls f(Gadget&&)
}
```

PERFECT FORWARDING

Forwarding reference `T&&` + `std::forward()` is a solution to interface bloat.

```
class Gadget;

void f(const Gadget&);
void f(Gadget&);
void f(Gadget&&);

template <typename Gadget>
void use(Gadget&& g) {
    f(std::forward<Gadget>(g)); // forwards original type to f()
}

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);         // calls use(const Gadget&) then calls f(const Gadget&)
    use(g);          // calls use(Gadget&) then calls f(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls f(Gadget&&)
}
```

std::forward

Forwarding reference (even bind to r-value) is treated as l-value inside template function.

```
template <typename T>
void use(T&& t) {             // pass t as l-value unconditionally
    f(t);
}

template <typename T>
void use(T&& t) {             // pass t as r-value unconditionally
    f(std::move(t));
}

template <typename T>
void use(T&& t) {             // pass t as r-value if it is r-value,
    f(std::forward(t));       // pass as l-value otherwise
}
```

TEMPLATE TYPE DEDUCTION

KNOWLEDGE CHECK :)

```
template <typename T>
void copy(T arg) {}

template <typename T>
void reference(T& arg) {}

template <typename T>
void universal_reference(T&& arg) {}

int main() {
    int number = 4;
    copy(number);         // int
    copy(5);              // int
    reference(number);    // int&
    reference(5);         // candidate function [with T = int] not viable: ex
    universal_reference(number);              // int&
    universal_reference(std::move(number));  // int&&
    universal_reference(5);                   // int&&
}
```

`decltype((expr))` WITH PARENTHESES

```
int x;
auto y = x;                  // int y = x;
decltype(x) z = x;           // int z = x;

int a;
auto b = (a);                // int b = a;
decltype((a)) c = a;         // int& c = a;

int d();
auto e = d();                // int e = d();
decltype(d()) f = d();       // int&& f = d();

decltype(auto) f1() {
    int x = 0;
    return x;                // returns int
}
decltype(auto) f2() {
    int x = 0;
    return (x);              // returns int&
}
```

- `decltype(x)` and `decltype((x))` are different things, two different keywords could be proposed instead of one.
- `decltype(x)` gives the declared type of the identifier `x`
- If you pass something that is not an identifier, it determines the type, then appends `&` for lvalues, `&&` for xvalues, and nothing for prvalues.

TYPE DEDUCTION

KNOWLEDGE CHECK :)

```
Gadget g;
auto a = g;                 // a : Gadget
decltype(auto) b = g;       // b : Gadget
decltype(g) c = g;          // c : Gadget
decltype((g)) d = g;        // d : Gadget&
```

```
Gadget g;
const Gadget& cg = g;
auto e = cg;                // e : Gadget
decltype(auto) f = cg;      // f : const Gadget&
decltype(cg) g = cg;        // g : const Gadget&
decltype((cg)) h = cg;      // h : const Gadget&
```

MOVE SEMANTICS

- r-value and l-value references
- Move constructor and move assignment operator
- Rule of 0, 3, 5
- `std::move()` and `std::forward()`
- Forwarding reference
- Reference collapsing
- Perfect forwarding
- `decltype((expr))`