More about move-semantics

The last time, we did a naive implementation of std::move

```
template <typename T>
T&& naive_move(T& value) {
    return static_cast<T&&>(value);
}
```

Today we will see the details of the *T&&* type and also talk about r-value and l-value.

Let's give a format definition to l-value and r-value. https://en.cppreference.com/w/cpp/language/value_category

The first thing to know, is that l-value and r-value are **types of expressions** (more like a syntax construction)

Let's look at an example:

```
struct String {
};

void f(const String&) {
    std::cout << 1;
}

void f(String&&) {
    std::cout << 2;
}

int main() {
    String s;
    f(s);           // << 1;
    f(String());    // << 2;
}
```

So, *T&&* is a special type, that **attracts** r-values.

In the first semester we had a few similar situations:

```
/*
const T& <---- const T
      T& <---- T

Base&    <---- Base
Derived& <---- Derived
*/
```

Now we meet it again in this form:

```
/*
const type& <---- lvalue
      type&& <---- rvalue
*/
```

Let's talk more about the properties of Rvalue references

Rvalue references and their properties

Properties:

1. Rvalue reference may be initialized only from rvalue expression
2. Rvalue reference, being returned from function or cast-expression, is rvalue expression

```
int main() {
    // First
    {
        int x = 0;
        // int&& r = x;                          // CE, x is lvalue
        int&& r = std::move(x);                  // OK
        // int&& rr = r;                         // CE, r is lvalue
        int& l = r;                              // OK

        x = 1;
        std::cout << x << r << l;                // << 111, because everything is still a re
    }

    // Second
    {
        int&& r = 1;                             // OK (xvalue thing)
        r = 2;

        std::string&& s = "sdfskfjskdfjlskf";    // OK
        s.push_back('a');                        // OK
        std::cout << s;                          // << sdfskfjskdfjlskfa
    }
}
```

What if we make a const rvalue reference?

```
void f(const String&&) {
    std::cout << 2;
}
```

Usually there is no reason to do so, though some examples of it's use exist. For example:

```
const std::string s = "dfjskfjdskfjsdkf";
std::string s2 = std::move(s);
```

Here we try to move a const object, **that's why the copy-ctor will be called** and as a result, std::move(s)'s type is **const string&&**.

Of course if we had a c-tor from **const string&&**, than that would be called (though, it is not usually implemented).

Why naive move is incorrect

We have the following problem: in out vector class we want to implement *emplace_back()*. In what way should we accept the arguments?

```
template <typename... Args>
void emplace_back(???... args) {
    if (sz >= cap) {
        reserve(cap > 0 ? cap * 2 : 1);
    }
    new (arr + sz) T(args...);
    ++sz;
}
```

But the actual problem, is that we **would like to pass some arguments as rvalue and other arguments as lvalue**. We need to find a way to pass variable number of args and at the same time save their lvalue/rvalue-ness. Here is our crutch:

```
// If T&& is an argument type of a template function
// where T is a template parameter of this function
// then T&& accepts both rvalue and lvalue

// If T&& accepts lvalue in that case, then T is deduced as lvalue-reference
                                    otherwise T is deduces as non-reference

// Perfect forwarding
template <typename... Args>
void emplace_back(Args&&... args) {
    if (sz >= cap) {
        reserve(cap > 0 ? cap * 2 : 1);
    }
    new (arr + sz) T(std::forward<Args>(args)...);
    ++sz;
}
```

std::forward does exactly that.

```
template <typename T>
void f(T&& x) {
```

```
    T y = x; // int& y = x;
    ++y;
    std::cout << x;
}

/*
    Referencing collapsing rules:

    T& + &   -> &
    T& + &&  -> &
    T&& + &  -> &
    T&& + && -> &&
*/

int main() {
    int x = 0;
    f(x); // T = int&, decltype(x) = int&

    f(5); // T = int, decltype(x) = int&&
}
```

Finally we can understand why naive move is wrong. It can only accept lvalue, but must be able to accept everything! But we can't simply accept T&&, because it will be perfect forwarding, and will add extra references.