值类型与移动语义
Value Category and Move Semantics

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Part 2**
- **Value Category**
  - **decltype**

- **Reference Qualifier**
  - **Deducing this**

- **Copy Elision**
  - **Return Value Optimization**

- **Analyzing Performance of Move Semantics**

Value Category and Move Semantics

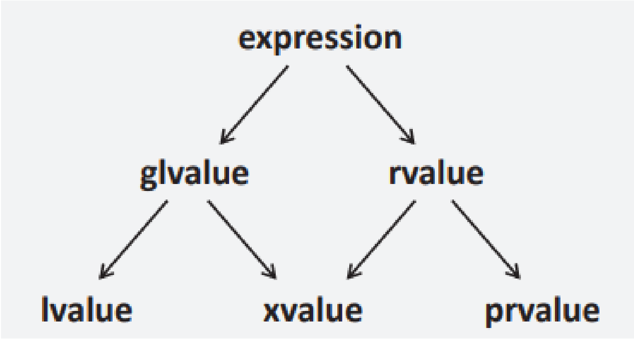
Value Category

Value Category

- Value category is classification of expressions.
- The history:
  - Classic category in K&R C:
    - lvalue: left-hand value, the expression that can appear at the left hand side of =.
    - rvalue: right-hand value, the expression that can only appear at the right hand side of =.
  - In C++, `const` is added.
    - Flaw: `const` cannot appear at the left hand side of =, so why not just use address to distinguish them?
  - Category in ANSI-C (C89) and C++98:
    - lvalue: locator value whose address can be taken by &.
    - rvalue: read-only value.

Value Category

- Since C++11, move semantics is introduced.
  - We need a category that can refer to both `std::move(lvalue)`, and temporaries (classic rvalue)!
    - Since both of them represent values whose resource can be stolen.
  - We then call such category **rvalue**, while “classic rvalue” is **prvalue** (pure rvalue).
    - Those who are rvalue but not prvalue are called **xvalue** (eXpiring value).
  - Notice that xvalue shows some properties similar to lvalue (e.g. comes from lvalue by `std::move`), so we may call them **glvalue** (generalized lvalue).



Value Category

- To be specific:
  - prvalue includes:

prvalue

The following expressions are *prvalue expressions*:

- a literal (except for string literal), such as `42`, `true` or `nullptr`;
- a function call or an overloaded operator expression, whose return type is non-reference, such as `str.substr(1, 2)`, `str1 + str2`, or `it++`;

Same as overloaded operators that commonly return value type instead of reference.

- `a++` and `a--`, the built-in post-increment and post-decrement expressions;
- `a + b`, `a % b`, `a & b`, `a << b`, and all other built-in arithmetic expressions;
- `a && b`, `a || b`, `!a`, the built-in logical expressions;
- `a < b`, `a == b`, `a >= b`, and all other built-in comparison expressions;
- `&a`, the built-in address-of expression;

```
int a = 1;
(a++) = 2;
```

✗

i.e. the member function; their categories are not very useful.

- `a.m`, the member of object expression, where `m` is a member enumerator or a non-static member function[2];
- `p->m`, the built-in member of pointer expression, where `m` is a member enumerator or a non-static member function[2];
- `a.*mp`, the pointer to member of object expression, where `mp` is a pointer to member function[2];
- `p->*mp`, the built-in pointer to member of pointer expression, where `mp` is a pointer to member function[2];

Comma is always same as the last expression, no matter result or category.

- `a, b`, the built-in comma expression, where `b` is an prvalue;
- `a ? b : c`, the ternary conditional expression for certain `b` and `c` (see definition for detail);

Conversion creates new object

- a cast expression to non-reference type, such as `static_cast<double>(x)`, `std::string{}`, or `(int)42`;
- the `this` pointer;

Similar to literals

- an enumerator;

NTTP will be covered in the next lecture.

- a non-type template parameter of a scalar type;

Value Category

- For `? a : b`, it's the category of a & b if they're of the same type and the same category; otherwise it creates a new temporary and thus prvalue.

```
int a = 1, b = 2;
double c = 1.0;

expr ? a : b; // lvalue
expr ? a : c; // prvalue, type is different
expr ? a : 2; // prvalue, category is different
```

- To conclude, in most cases, prvalue is **exactly** temporaries!
  - Literals, including enumerators;
  - Result of function call that returns value type (so it returns temporaries);
  - Operators & conversions that create temporaries;
- There are only few surprising cases, but they're not important.
  - i.e. member function and `this`.

Value Category

- xvalue includes:

xvalue

The following expressions are *xvalue expressions*:

Data members of rvalue, e.g. `std::move(a).b`, `A{}.b`

- `a.m`, the member of object expression, where `a` is an rvalue and `m` is a non-static data member of an object type;
- `a.*mp`, the pointer to member of object expression, where `a` is an rvalue and `mp` is a pointer to data member;
- `a, b`, the built-in comma expression, where `b` is an xvalue;

When b and c are both xvalue

- `a ? b : c`, the ternary conditional expression for certain `b` and `c` (see definition for detail);

(since C++11)

- a function call or an overloaded operator expression, whose return type is rvalue reference to object, such as `std::move(x)`;
- `a[n]`, the built-in subscript expression, where one operand is an array rvalue;
- a cast expression to rvalue reference to object type, such as `static_cast<char&&>(x)`;

Covered later in copy elision

- any expression that designates a temporary object, after temporary materialization; (since C++17)

Covered later in return value optimization.

- a move-eligible expression. (since C++23)

Value Category

- To conclude, xvalue is:
  - Data members of rvalue;
  - Expressions that creates rvalue reference, like function call and conversion.
  - And some special cases including `?:`, `[]` and `,`.
- `std::move` creates xvalue!
  - Yes, it's a function that creates rvalue reference to the original object;
  - But how is it implemented?
    - Hint: • a cast expression to rvalue reference to object type, such as `static_cast<char&&>(x)`;
  - Yes, `std::move(x)` is exactly same as `static_cast<Type&&>(x)`!
    - It's just short for that long expression.
  - Notice that for const object, it generates `const Type&&` (and thus cannot be stolen) since dropping `const` is dangerous.

Value Category

- lvalue includes:

Named variables

Same as overloaded operators that commonly return reference type.

Data members of lvalue, and static data members. (`p->m` is equivalent to `(*p).m` and `*p` is always lvalue.)

rvalue reference to function (but not important).

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 (but see Move-eligible expressions);
- a function call or an overloaded operator expression, whose return type is lvalue reference, such as `std::getline(std::cin, str)`, `std::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 object 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 certain `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)` or `static_cast<void(&)(int)>(x)`;
- a non-type template parameter of an lvalue reference type;
- 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)

Value Category

- To conclude, lvalue is basically “long-living” data.
  - Named variables;
  - Data members of lvalue, and static data members of any category;
  - Result of function call that returns lvalue reference type;
  - Operators & conversions that are equivalent to creating the lvalue reference to the original;
  - Particularly, string literals.
    - You can understand that informally — they’re just stored in read-only segment of the program so they’re long-living (while other literals are just temporaries).
- And some unimportant cases, like rvalue reference to function.

Value Category

- Wait, one more thing…
  - cppreference leaves something out…

7 If E2 is declared to have type "reference to T", then E1.E2 is an lvalue of type T. If E2 is a static data member, E1.E2 designates the object or function to which the reference is bound, otherwise E1.E2 designates the object or function to which the corresponding reference member of E1 is bound. Otherwise, one of the following rules applies.

  - Reason: for static and reference members, the object doesn't in fact own their resources, so regulating them as xvalue may cause unexpected behaviors.
  - This makes a difference for `std::move(a.b)` and `std::move(a).b`.
    - For `struct A { string b; }`, `c = std::move(a.b)` ⇔ `c = std::move(a).b`.
      - They're both xvalue and thus call move assignment of `string`.
    - For `struct A { string& b; }`, `c = std::move(a.b)` ⇎ `c = std::move(a).b`.
      - The former explicitly means "move away `a.b`", while the latter means "get `b` from moved `a`". Then owning or not matters!

This may need to be considered when writing generic code.

Value Category

- Now we can formally distinguish different references.
  - lvalue reference (`Type&`): can only refer to (non-const) lvalue.
  - const lvalue reference (`const Type&`): to be consistent with C++98, it can refer to any value category but it's **read-only**.
  - rvalue reference (`Type&&`): can only refer to (non-const) rvalue, i.e. xvalue & prvalue; So its resource may be stolen.
  - const rvalue reference (`const Type&&`): useless.

- As parameters, the overload resolution rule is:
  - Non-const lvalue will first try to match `&`, and secondly `const&`.
  - const lvalue will only try to match `const&.`
  - rvalue will first try to match `&&`, (and then `const&&`), and secondly `const&`.

Value Category

- Exercise: Alice learns that `const` is beneficial to optimization, so she writes a function like:

```cpp
class A
{
    std::size_t size_;
public:
    A(std::size_t size) : size_{ size } {}
    const std::vector<int> Test(const int ele) const
    {
        return std::vector<int>(size_, ele);
    }
};
```

```cpp
A a{ 1 };
std::vector<int> result = a.Test(2);
```

  - Explain all `const` above and try to find the performance pitfall.

Value Category

```cpp
class A
{
    std::size_t size_;
public:
    A(std::size_t size) : size_{ size } {}
    const std::vector<int> Test(const int ele) const
    {
        return std::vector<int>(size_, ele);
    }
};
```

```cpp
A a{ 1 };
std::vector<int> result = a.Test(2);
```

The temporary is read-only.

`ele` is read-only.

The method cannot modify any data members (except for `mutable`); `const A` can call this function.

- Problem: read-only temporary is completely useless.
  - You can still use a non-const variable to accept that.
  - But it creates a const rvalue, which cannot be bound on `A&&`!
    - It can only be bound on `const A&`, so it calls copy ctor instead of move ctor!
- Conclusion: return value of const value type is almost always useless. (Reference type may be useful, e.g. `operator[]`).

decltype

- So is there a way to judge the value category of the expression?
- Yes, decltype!
  - Abbreviation of **decl**ared **type**.
  - /ˈdaɪkl/ or /ˈdiːkwəl/

- It is a keyword to deduce type from a variable name (including member access) **or** an expression.
  - They have different rules!

Notice that some expressions may be classified wrongly due to wrong compiler implementation, e.g. [msvc].

decltype

- For deducing the type of variable name & member access, just same as the declared type.
  - E.g. `a, a.b, ptr->b`
- Example:

```cpp
#include <string>
#include <iostream>

void test(std::string&& str1, std::string& str2, std::string str3)
{
    std::cout << std::boolalpha;
    std::cout << std::is_same<decltype(str1), std::string>::value      // false
        << " " << std::is_same<decltype(str1), std::string&>::value     // false
        << " " << std::is_same<decltype(str1), std::string&&>::value;   // true

    std::cout <<"\n"<< std::is_same<decltype(str2), std::string>::value // false
        << " " << std::is_same<decltype(str2), std::string&>::value     // true
        << " " << std::is_same<decltype(str2), std::string&&>::value;   // false

    std::cout <<"\n"<< std::is_same<decltype(str3), std::string>::value // true
        << " " << std::is_same<decltype(str3), std::string&>::value     // false
        << " " << std::is_same<decltype(str3), std::string&&>::value;   // false
}

int main()
{
    std::string a;
    test("", a, "");
    return 0;
}
```

You can use `std::remove_reference_t<decltype(str)>` to always get the value type.

decltype

- For deducing the type of expression:
  - `decltype(prvalue)` → value type.
  - `decltype(lvalue)` → lvalue reference.
  - `decltype(xvalue)` → rvalue reference.
- You can judge what value category an expression belongs to in this way.
- E.g. `T1 == int, T2 == int&, T3 == int&&`

```cpp
int a = 0;
using T1 = decltype(1 + 1);
using T2 = decltype(++a);
using T3 = decltype(std::move(a));
```

decltype

- By adding an additional pair of parentheses, a variable name is then an expression.
  - And we know that variable name as expression is just lvalue, so it always gets lvalue reference.
- Example:

```cpp
void test2(std::string&& str1, std::string& str2, std::string str3)
{
    std::cout << std::boolalpha;
    std::cout << std::is_same<decltype((str1)), std::string>::value       // false
        << " " << std::is_same<decltype((str1)), std::string&>::value      // true
        << " " << std::is_same<decltype((str1)), std::string&&>::value;    // false

    std::cout <<"\n"<< std::is_same<decltype((str2)), std::string>::value // false
        << " " << std::is_same<decltype((str2)), std::string&>::value      // true
        << " " << std::is_same<decltype((str2)), std::string&&>::value;    // false

    std::cout <<"\n"<< std::is_same<decltype((str3)), std::string>::value // false
        << " " << std::is_same<decltype((str3)), std::string&>::value      // true
        << " " << std::is_same<decltype((str3)), std::string&&>::value;    // false
}
```

`decltype(auto)`

- Sometimes you may need `decltype(Statement) var = Statement;`
  - While you cannot use `auto`, since it only deduces decayed type.
  - But it's too long to write such statement…
- C++ provides `decltype(auto)`!
  - You can directly write `decltype(auto) var = Statement`.
  - Similar to `auto`, you can also use it in function return type, e.g. `decltype(auto) Func() { return 1; }`.
- Exercise: for `int a = 1;`
  - `decltype(auto) b = a;`
  - `decltype(auto) d = (a);`
  - `decltype(auto) e = std::move(a);`
  - `decltype(auto) c = 1;`

Value Category and Move Semantics

Reference Qualifier

Reference Qualifier

- If seems that some illegal operations for fundamental types become legal for class with operator overloads.
  - Why?

```cpp
int a = 1;
(a + 1) += 1;
```

(局部变量) int a
联机搜索
表达式必须是可修改的左值

Compile error ❌

```cpp
Integer b = 1;
(b + 1) += 1;
```

Compile Okay?!

```cpp
class Integer
{
    int num;
public:
    Integer(int n) : num{ n } {}
    friend Integer operator+(const Integer&, const Integer&);
    Integer& operator+=(const Integer& another) {
        num += another.num;
        return *this;
    }

};

Integer operator+(const Integer& a, const Integer& b)
{
    return { a.num + b.num };
}
```

Reference Qualifier

- Overloaded operators are essentially function call*, so it's equivalent to `operator+(b, 1).operator+=(1)`.
  - `b.operator+` generates an `Integer` rvalue.
  - And `Integer` rvalue can do the function call of course…
- So if we want to make it illegal, we need to prohibit rvalue from calling it.
  - That's what reference qualifier does!

```
Integer& operator+=(const Integer& another) & {
    num += another.num;
    return *this;
}
```

```
Integer b = 1;
(b + 1) += 1;
```

没有与这些操作数匹配的 "+=" 运算符

*But the evaluation order is same as built-in operators since C++17, as we've said in Lecture 1.

Reference Qualifier

- & will bind lvalue only and && will bind rvalue.
  - It can also be combined with cv-qualifiers, so `&` means to bind non-const lvalue while `const&` means to capture all values (equivalent to `Integer&` and `const Integer&`).
  - Unlike cv-qualifiers, once you use ref-qualifiers, overloading without ref-qualifier is illegal.
    - E.g.

Reference Qualifier

- Lots of astonishing utilities come from restricting value category.
    - Case 1 (before C++23): Is there any bug in this piece of code?

```cpp
#include <string>

class Person
{
private:
    std::string name_ = "test";
public:
    const std::string& GetName() const {
        return name_;
    }
};

Person RecruitNewPerson()
{
    return Person{};
}

int main()
{
    for (char ch : RecruitNewPerson().GetName())
        // ...
    return 0;
}
```

- Hint: the essence of range-based for loop is.

The range-based `for` statement

`for (` *init-statement*$_{opt}$ *for-range-declaration* `:` *for-range-initializer* `)` *statement*

is equivalent to

```
{
    init-statement_opt
    auto &&range = for-range-initializer ;
    auto begin = begin-expr ;
    auto end = end-expr ;
    for ( ; begin != end; ++begin ) {
        for-range-declaration = * begin ;
        statement
    }
}
```

Universal reference; you may just see it as a `const&` to the initializer **here (only here and currently)**.

Reference Qualifier

```cpp
#include <string>

class Person
{
private:
    std::string name_ = "test";
public:
    const std::string& GetName() const {
        return name_;
    }
};
```

```cpp
Person RecruitNewPerson()
{
    return Person{};
}

int main()
{
    for (char ch : RecruitNewPerson().GetName())
        // ...
    return 0;
}
```

- So our program is like:

```cpp
auto&& range = RecruitNewPerson().GetName();
for (auto pos = range.begin(); pos != range.end(); ++pos)
    // ...
```

  - `RecruitNewPerson` returns a temporary `Person`…
  - And `GetName` returns reference to its member!
- Once the first statement ends, the `Person` temporary will be destroyed and thus the reference is dangling!
  - So our for-loop is iterating over freed memory…
- Wait, you may remember what we taught in *Lifetime* section:

BTW, && can also extend.

Also, the lifetime of **returned temporaries** can be extended by some references, e.g. we've learnt `const&`.

- NOTE AGAIN: this requires "returned temporaries"; returned reference or pointer to local variable is still **wrong**.

```cpp
struct A{ };
A bar() { return A{}; };
const A& a = bar();
```

Reference Qualifier

- Yes, but what it references is `const std::string&` rather than the `Person` temporary itself.
  - So it won't extend its lifetime…
- Solution 1: let `GetName` return `std::string`.
  - Then we can extend the lifetime by reference.
  - But it may be inefficient for `GetName` of lvalue since the function call will always create a new `std::string`.
    - i.e. `const auto& str = person.GetName()` will create `std::string` unnecessarily.

Reference Qualifier

- Solution 2: use range-based for *init-statement* since C++20.

```cpp
for (auto person = RecruitPerson(); auto ch : person.GetName())
{
    // ...
}
```

  - But this needs users to take care; could we prevent dangling from the scratch?
- Solution 3: use reference qualifier!
  - For lvalue return reference;
  - For rvalue return value type!
    - `std::move` is because rvalue basically means the value can be stolen, so moving away the member is reasonable and efficient.

```cpp
#include <string>

class Person
{
private:
    std::string name_ = "test";
public:
    const std::string& GetName() const& {
        return name_;
    }

    std::string GetName()&& {
        return std::move(name_);
    }
};
```

Reference Qualifier

- Note 1: besides preventing bug, it could be utilized to boost performance.
  - Example:

```cpp
std::vector<std::string> names;
names.push_back(std::move(person).GetName());
```

  - This is equivalent to `std::move(person.name_)`, but exposed by a Getter.
- Note 2: since C++23, lifetime of most temporaries generated by expressions in *range-initializer* will be extended automatically.
  - Since this bug is too common…
  - Unless you're deliberate, lifetime won't be a problem anymore here.

```cpp
using T = std::list<int>;
const T& f1(const T& t) { return t; }
const T& f2(T t)        { return t; } // always returns a dangling reference
T g();

void foo()
{
    for (auto e : f1(g())) {} // OK: lifetime of return value of g() extended
    for (auto e : f2(g())) {} // UB: lifetime of f2's value parameter ends early
}
```

[7] The fourth context is when a temporary object is created in the *for-range-initializer* of a range-based `for` statement. If such a temporary object would otherwise be destroyed at the end of the *for-range-initializer* full-expression, the object persists for the lifetime of the reference initialized by the *for-range-initializer*.

Reference Qualifier

std::optional<T>::and_then

```
template< class F >
constexpr auto and_then( F&& f ) &;          (1) (since C++23)
template< class F >
constexpr auto and_then( F&& f ) const&;     (2) (since C++23)
template< class F >
constexpr auto and_then( F&& f ) &&;         (3) (since C++23)
template< class F >
constexpr auto and_then( F&& f ) const&&;    (4) (since C++23)
```

- Case 2: `std::optional/expected` optimization.
  - Example:

```
std::optional opt{ Object{} };
auto opt2 = opt.or_else([]() -> decltype(opt) { return std::nullopt; });
```

```
Construct at 0x7ffe0ddafeee
Move at 0x7ffe0ddafeec
Destruct at 0x7ffe0ddafeee
Const Copy at 0x7ffe0ddafeea
Destruct at 0x7ffe0ddafeea
Destruct at 0x7ffe0ddafeec
```

```
std::optional opt{ Object{} };
auto opt2 = std::move(opt).or_else([]() -> decltype(opt) { return std::nullopt;
```

```
Construct at 0x7ffdb51c5cbe
Move at 0x7ffdb51c5cbc
Destruct at 0x7ffdb51c5cbe
Move at 0x7ffdb51c5cba
Destruct at 0x7ffdb51c5cba
Destruct at 0x7ffdb51c5cbc
```

Notice that the `Object` in `opt` is moved away.
It still `.has_value()`, but the value is in moved-from states.

- So if you're using a lvalue, the first `or_else` in chain will copy; you need `std::move(xx).or_else()` to make it a move.

Deducing this

- Since C++23, you can also use **explicit object member function** (informally named as **deducing this**).
  - If the first parameter is decorated with `this`, and the decayed type is the class itself, then the first parameter is the object itself.
    - i.e. here `this == &self`.
  - Wow, that's somehow like Python!
  - You can also do something brand new…

```cpp
class Person
{
    std::string name_;
public:
    const std::string& GetName(this const Person& self)
    {
        return self.name_;
    }

    std::string GetName(this Person&& self)
    {
        return std::move(self.name_);
    }
};
```

Deducing this

- It can make the explicit object be of value type!
  - For example, if some object is quite small, we've said it's better to use the value type instead of the reference type (e.g. reducing alias).
  - So if you don't need to modify the original object, you could code like:

```
struct just_a_little_guy {
    int how_smol;
    int uwu(this just_a_little_guy);
};
```

  - Assembly change:

```
sub     rsp, 40
lea     rcx, QWORD PTR tiny_tim$[rsp]
mov     DWORD PTR tiny_tim$[rsp], 42
call    int just_a_little_guy::uwu(void)
add     rsp, 40
ret     0
```

Credit: [C++23's Deducing this: what it is, why it is, how to use it - C++ Team Blog]

```
mov     ecx, 42
jmp     static int just_a_little_guy::uwu(this just_a_little_guy)
```

Deducing this

- Note1: all members should be accessed by the first parameter; `name_`, `this` and `this->name_` are all illegal.
- Note2: it completely replaces the original function;
  - You cannot add any qualifier at the end of the function declarator;
  - You cannot define non-explicit object member function of the same utility.

```
void p(this C) const;      // Error: “const” not allowed here
static void q(this C);     // Error: “static” not allowed here
    void foo(this X const& self, int i); // same as void foo(int i) const &;
//  void foo(int i) const &; // Error: already declared
```

- Note3: you can define recursive lambda in this way.
  - Question: what does this `auto` mean?

Or `auto&` if the lambda is big.

```
auto fib = [](this auto self, int n) {
    if (n < 2) return n;
    return self(n-1) + self(n-2);
};
```

Equivalent to

```
auto fib = []<typename T>(this T self, int n) {
    if (n < 2) return n;
    return self(n-1) + self(n-2);
};
```

Value Category and Move Semantics

Copy Elision

Value Category and Move Semantics

- Copy Elision
  - prvalue copy elision
  - Return value optimization

Copy Elision

- Observe: `auto a = std::string{"Hello, world"};`
  - i.e. `std::string a = std::string{"Hello, world"};`
  - So what functions should be called here?

  - Logical process: `“Hello, world”` is used to construct a `std::string` temporary, and move ctor is called to construct actual variable.
  - Fact: `“Hello, world”` doesn’t call move ctor.
- Such reasonable optimization is called “copy elision”, since it doesn’t need any copy (and move).
  - This actually has some side-effects if move ctor has e.g. output, so it’s compiler’s duty to check it.
  - But since C++17, such elision is obligated in the standard, and it’ll always happen even if there are side effects.

Copy Elision

- We know that prvalue are generally short-living temporaries, either are discarded or are used to generate objects.
  - So copy elision happens for prvalue, collapses the intermediate constructions of temporaries.
  - That is, the time when prvalue is used to construct a real object is delayed **as much as it can**.
    - When the object is finally constructed, it's said the prvalue is **materialized**.

- E.g. `auto a = std::string{"Hello, world"};`
  - `std::string{"Hello, world"}` is a prvalue, so its construction is delayed as much as it can until it finds that a **result object** (i.e. `a`) must be generated.

Copy Elision

- So what’s the definition of “delay as much as it can”?
  - Or, except for a result object, when does a prvalue have to materialize?

  - Binding on some references (e.g. `T&&`, `const T&`);
    - e.g. `void Func(const A&); Func(A{});`
  - For class object, accessing its non-static data members / calling non-static member functions;
    - e.g. `A{}.a;`
  - [Rarely used] As an array, being subscripted or converted to pointers;
  - Used for `std::initializer_list`;
    - e.g. `std::vector{ A{}, A{} }`;
  - Or finally discarded (because logically it should create a new object).

  - If the result object is not lvalue, then it’s materialized as xvalue.

Copy Elision

- Example 1:

```cpp
class C {
public:
    C() {};
    C(const C&) = delete; // this class is neither copyable ...
    C(C&&) = delete; // ... nor movable
};
C createC() {
    return C{ };
}
void takeC(C val) {
    return;
}

int main()
{
    auto n = createC(); // OK since C++17 (error prior to C++17)
    takeC(createC()); // OK since C++17 (error prior to C++17)
    return 0;
}
```

Copy Elision

- Example 2: how many time(s) ctor & dtor are called? `C{ C{ createC() } };`
  - Notice that there is no intermediate procedures that force the returned prvalue `create()` to materialize.
  - Finally, no result object is really generated so the discarded prvalue will be used to materialize a xvalue.
  - Ctor and Dtor is only called **once**.
  - Notice: This won’t compile in msvc 19.29 (The final version of VS2019); this bug is fixed in msvc 19.30 (VS2022).
- Example 3: value category of `Test{}.obj`:
  - `Test{}` is a prvalue and try to delay its materialization.
  - However, accessing its member forces it to materialize to a xvalue.
  - The non-static member of xvalue is still a xvalue.

Value Category and Move Semantics

- Copy Elision
  - prvalue copy elision
  - Return value optimization
    - Implicit move

RVO

- Remember our example?
  - Logical process:
    - strVec creates a temporary;
    - strVec is destructed;
    - The temporary is assigned to v.
  - Fact: we've said that due to NRVO, the process is simplified to:
    - strVec (seen as the temporary) is assigned to v.

```cpp
#include <vector>
#include <string>

std::vector<std::string> CreateAndInsert()
{
    std::vector<std::string> strVec;
    strVec.reserve(3); // prohibit reallocation to reduce complexity.
    std::string s = "data";

    strVec.push_back(s);
    strVec.push_back(s + s);
    strVec.push_back(s);

    return strVec;
}

int main()
{
    std::vector<std::string> v;
    v = CreateAndInsert();
    return 0;
}
```

RVO

- A more obvious example…

```cpp
#include <iostream>
class Object
{
public:
    Object() { std::cout << "Construct at " << this << "\n"; };
    ~Object() { std::cout << "Destruct at " << this << "\n"; };
    Object(const Object&) {
        std::cout << "Const Copy at " << this << "\n";
    };
    Object(Object&&) { std::cout << "Move at " << this << "\n"; };
    Object& operator=(const Object&) {
        std::cout << "Const Copy Assignment at " << this << "\n";
        return *this;
    };
    Object& operator=(Object&&) {
        std::cout << "Move Assignment at " << this << "\n";
        return *this;
    };
};

Object GetObject_RVO()
{
    return Object();
}

Object GetObject_NRVO()
{
    Object obj;
    return obj;
}

int main()
{
    std::cout << std::hex;

    std::cout << "RVO\n";
    Object obj1;
    obj1 = GetObject_RVO();

    std::cout << "\nNRVO\n";
    Object obj2;
    obj2 = GetObject_NRVO();

    std::cout << "\nDone.\n";
    return 0;
}
```

RVO

- Output:

In Windows/VS2019(msvc19.29)/Release:

RVO
Construct at 000000AC4FBCFE72
Construct at 000000AC4FBCFE70
Move Assignment at 000000AC4FBCFE72
Destruct at 000000AC4FBCFE70

NRVO
Construct at 000000AC4FBCFE71
Construct at 000000AC4FBCFE70
Move Assignment at 000000AC4FBCFE71
Destruct at 000000AC4FBCFE70

Done.
Destruct at 000000AC4FBCFE71
Destruct at 000000AC4FBCFE72

In Linux/g++-11/no option:

RVO
Construct at 0x7fff3c079d85
Construct at 0x7fff3c079d87
Move Assignment at 0x7fff3c079d85
Destruct at 0x7fff3c079d87

NRVO
Construct at 0x7fff3c079d86
Construct at 0x7fff3c079d87
Move Assignment at 0x7fff3c079d86
Destruct at 0x7fff3c079d87

Done.
Destruct at 0x7fff3c079d86
Destruct at 0x7fff3c079d85

RVO

- Basically equivalent, so we might as well choose Linux.
- There is an option to disable this optimization in gcc, and we try again:

In Linux/g++-11/-std=c++11 -fno-elide-constructors:

RVO
Construct at 0x7ffc5e3d4bc5
Construct at 0x7ffc5e3d4b97
Move at 0x7ffc5e3d4bc7
Destruct at 0x7ffc5e3d4b97
Move Assignment at 0x7ffc5e3d4bc5
Destruct at 0x7ffc5e3d4bc7

NRVO
Construct at 0x7ffc5e3d4bc6
Construct at 0x7ffc5e3d4b97
Move at 0x7ffc5e3d4bc7
Destruct at 0x7ffc5e3d4b97
Move Assignment at 0x7ffc5e3d4bc6
Destruct at 0x7ffc5e3d4bc7

Done.
Destruct at 0x7ffc5e3d4bc6
Destruct at 0x7ffc5e3d4bc5

RVO

- Comparison:

```
Object GetObject_RVO()
{
    Object obj;
    return obj;
}
int main()
{
    Object obj1;
    obj1 = GetObject_RVO();
}
```

NRVO
Construct at 0x7fff3c079d85
Construct at 0x7fff3c079d87
Move Assignment at 0x7fff3c079d85
Destruct at 0x7fff3c079d87

NRVO
Construct at 0x7ffc5e3d4bc5
Construct at 0x7ffc5e3d4b97
Move at 0x7ffc5e3d4bc7
Destruct at 0x7ffc5e3d4b97
Move Assignment at 0x7ffc5e3d4bc5
Destruct at 0x7ffc5e3d4bc7

- NRVO elides the move from the temporary to return value & the destruction of the temporary.

RVO

- So there are two kinds of RVO.
  - If the returned object is prvalue (e.g. `Object{}`), then since C++17 it will of course elide the temporary. It’s also called RVO directly.
    - So not discussed anymore.
  - Otherwise, if the returned object is lvalue (as our example before), then it’s called NRVO (Named RVO).
- NRVO has many restrictions to apply:
  - Must be a name, no other things (i.e. `return x;`).
  - Must be local variable, with the type **same** as function return type;
    - global ones won’t be destructed after exiting the scope so it cannot be moved.
  - Must **NOT** be the parameter.
  - Must be the only returned variable in all return statements.

- Examples:
  - `return std::move(obj);` ❌ Not a name
  - `return static_cast<Object>(obj);` ❌ Not a name
  - `Object obj; Object Func() { return obj; }` ❌ Not a local variable
  - `Object Func(Object obj) { return obj; }` ❌ Is a parameter
  - ```
    Object Func() {
        Object obj1, obj2;
        if (condition) return obj1;
        else return obj2;
    }
    ```
    ❌ Not all `return` have the same returned variable.
  - ```
    Object Func() {
        Object obj1;
        if (condition) return obj1;
        // ……
        return obj1;
    }
    ```
    ✅
  - ```
    Object Func() {
        int m = 1;
        return m;
    }
    ```
    ❌ type of m (i.e. `int`) isn't same as return type (i.e. `Object`)
    ```

NRVO

- And that's why we say `return std::move(x)` may decrease performance…

```cpp
Object Test()
{
    Object obj1;
    return obj1;
}
```

NRVO, so no intermediate temporary.

```cpp
Object Test()
{
    Object obj1;
    return std::move(obj1);
}
```

No NRVO, so create an intermediate temporary.

An additional move ctor call.

  - Some compiler may have options to check that, e.g. gcc -Wpessimizing-move and –Wredudant-move

NRVO

- Explain the code before:
  - Why is it proper to have `std::move` here?
- Reason: `name_` is not local variable!
  - So `return name_;` doesn't trigger NRVO, and will cause a copy to the temporary.

```cpp
#include <string>

class Person
{
private:
    std::string name_ = "test";
public:
    const std::string& GetName() const& {
        return name_;
    }

    std::string GetName()&& {
        return std::move(name_);
    }
};
```

Implicit Move

- Sometimes NRVO is impossible, but copy is also unnecessary…
  - E.g.

```
Object Func() {
    Object obj1, obj2;
    if (condition) return obj1;
    else return obj2;
}
```

  - Though we cannot trigger NRVO, `obj1` & `obj2` don’t need to copy to a temporary; they can “move” to the temporary.
    - It’s same as writing `std::move(obj1)` & `std::move(obj2)`, but just saves your trouble to write `std::move` over and over again.
  - Such optimization is called “implicit move”.

Implicit Move

- The history of implicit move:
  - From C++11 to C++20, "two overload resolution" will be performed.
    - That is, it will try to see the result as rvalue; if it fails, then lvalue.
  - C++11 requires return type to be exactly same.
  - C++14 loosens it, but still requires the return type accepts explicitly rvalue reference of the type of return value.
  - C++20 loosens it again, to allow it to accept value type.

  - However, the rules are ambiguous (which leads to different behaviors in different compilers) and hard to remember, so C++23 brings a final solution.
    - That is, all non-volatile variables with automatic storage duration (including parameters) or their rvalue references **will be seen as xvalues** in `return`.
      - It's allowed to have additional pairs of parentheses.

Notice that msvc hasn't implemented it as of 2024/11.

Simpler implicit move	P2266R3	13	13	

Implicit Move

- Breaking change: some dangling reference will make compile error.
  - Reason: xvalue cannot be bound on lvalue reference.
  - But dangling `const&` and `&&` are still legal to compile.

```cpp
Object& Test()
{
    Object obj1;
    return obj1;
}
```

- Consider: what's the returned type of `Test()`?
  - Wrong answer: `Object&&` since C++23 and `Object&` before, since it's xvalue / lvalue respectively.
    - Reason: here it's equivalent to `decltype(obj1)`, i.e. deduce the type of variable. **It just deduces its type**, not related to value category.
  - Correct answer: always `Object`.

```cpp
decltype(auto) Test()
{
    Object obj1;
    return obj1;
}
```

Implicit Move

```cpp
decltype(auto) Test()
{
    Object obj1;
    return (obj1);
}
```

- What about:
  - Now it's not a variable, but an expression.
  - So, `decltype` will consider the value category.
  - Before C++23, it's just lvalue, which deduces `Object&`.
  - And since C++23,
    - That is, all non-volatile variables with automatic storage duration (including parameters) or their rvalue references **will be seen as xvalues** in `return`.
      - It's allowed to have additional pairs of parentheses.
    - So then it's xvalue, which deduces `Object&&`.
  - Here both deductions are dangerous since references are dangling; but if `obj1` is parameter `Object& obj1`, then it's Okay.

Implicit Move

- Exercise:

```
Object Test()
{
    return A{}.obj;
}
```

  - Does this trigger NRVO? ❌ Not a name
  - Does this trigger implicit move? ❌ Not a name
  - Does this trigger copy?
    - Still no, because `A{}` is prvalue, and it's materialized to xvalue, and accessing its member will still get xvalue.
    - So, this is just a move, not caused by implicit move.

Copy Elision

- To conclude, for a return value:
  1. Check whether it’s prvalue so it could trigger RVO;
  2. Check whether it’s a local variable that satisfies those restrictions so it could trigger NRVO;
  - And if not, then you could `std::move` every return manually, or you can elide them in these cases:
  1. Check whether the returned one is just a local variable, so implicit move;
  2. Check whether the expression is rvalue.

  - If all of them are false, and you still want to move the returned stuff (e.g. `return name_;` in `Person` before), then you must add `std::move` explicitly.
    - Otherwise copy will happen.

Value Category and Move Semantics

Analyzing Performance of Move Semantics

Analyzing Performance

- We'll use function parameter choices to have an exercise on analyzing move semantics.
  - We've said this ctor implementation is naïve.
    - Here you can add `std::move`.
- In fact, there are also three other choices for parameters (omit salary):

```cpp
#include <string>

class Person
{
public:
    std::string name;
    int salary;

    Person(std::string init_name, int init_salary) :
        name{ init_name }, salary{ init_salary } {}
```

  - `Person(std::string& init_name) : name{init_name} {};`
  - `Person(const std::string& init_name) : name{init_name} {};`
  - `Person(std::string&& init_name) : name{std::move(init_name)} {};`
  - Which one is the best?
    - Try it yourself; analyze by passing in parameters of different value categories!

Scenario analysis

  - First, since prvalue is basically temporary, it will always call ctor, either to construct the parameter, or to construct a temporary to bind on the parameter, so we'll omit it.
- `std::string`
  - For lvalue, it'll be copied to parameter and then the parameter will be moved to the member.
    - 1 copy ctor + 1 move ctor + 1 empty-state dtor;
  - For xvalue, it'll be moved to parameter and then the parameter will be moved to the member.
    - 2 move ctor + 1 empty-state dtor;
  - For prvalue, the parameter will be constructed directly and then the it will be moved to the member.
    - 1 move ctor + 1 empty-state dtor.

Scenario analysis

- `std::string&`
  - For lvalue, nothing happens when passing parameter and the parameter will be copied to the member.
    - 1 copy ctor.
  - Cannot accept rvalue.
- `std::string&&`
  - Cannot accept lvalue.
  - For xvalue, nothing happens when passing parameter and the parameter will be moved to the member.
    - 1 move ctor
  - For prvalue, a temporary will be constructed and bound on the reference; then it will be moved to the member, and finally destructed.
    - 1 move ctor + 1 empty-state dtor

Scenario analysis

- `const std::string&`
  - For lvalue, nothing happens when passing parameter and the parameter will be copied to the member.
    - 1 copy ctor
  - For xvalue, nothing happens when passing parameter and the parameter will be copied to the member.
    - 1 copy ctor
  - For prvalue, a temporary will be constructed and bound on the reference; then it will be copied to the member, and finally destructed.
    - 1 copy ctor + 1 dtor

Parameter choice for normal ctors

- Summary in one table:

	lvalue	xvalue	prvalue
Value type	1 copy + 1 move + 1 empty dtor	2 move + 1 empty dtor	1 move + 1 empty dtor
&	1 copy	no	no
const &	1 copy	1 copy	1 copy + 1 dtor
&&	no	1 move	1 move + 1 empty dtor

- And we cannot have both value & reference overloading (since it's ambiguous), so we have two choices:
  - Value type;
  - Overloading by `const&` + `&&`
    - Due to matching priority, rvalue will call `&&` first, so:

	lvalue	xvalue	prvalue
const & + &&	1 copy	1 move	1 move + 1 empty dtor

Parameter choice for normal ctors

- So for a ctor parameter that:
  - Has the same type as the member;
  - Is moveable;
  - And is used to initialize that member directly;
- Then it seems evident that we need to choose `const&` + `&&`.
- However…
  - Key Observation 1: The performance gap between these two choices are just 1 move ctor + 1 empty-state dtor.
  - Key Observation 2: There are usually many data members in a class, so if every member is initialized with two overloaded types, there should be 2^N overloaded ctors.

Parameter choice for normal ctors

- Conclusion:
  - If the move ctor & empty-state dtor for the parameter is cheap enough, using value type is totally acceptable.
  - Otherwise, it's better to use overloaded references.
- Example: `std::vector<std::string>` matches the former, and `std::array<std::string, 1000>` matches the latter.
  - `std::vector` holds only 3 pointers and move ctor is just copying 3 pointers;
  - While `std::array` holds a total array itself, so it has to move every element to another object in move ctor. So move ctor is a relatively huge cost!
    - Move semantics usually reduces the cost to "shallow copy", but if shallow copy is costly enough, then it's still inefficient.

Analyzing Performance

- The second example is Setter.
  - Getter has been solved by reference qualifier (or deducing this).

```cpp
class Person
{
    std::string name_;
public:
    const std::string& GetName() const& { return name_; }
    std::string GetName() && { return std::move(name_); }
};
```

  - For setter, you can similarly define:
    - By value type;
    - By `const&` + `&&` overloading.
  - It seems that the only difference is 1 move ctor -> 1 move assignment.

Analyzing Performance

- However, move assignment will discard the original resource…
  - Sometimes it's improper, e.g. for `std::string`, the original memory may be larger.
  - E.g. if you append something to the string afterwards, larger space will be more unlikely to reallocate.
- To conclude:
  - For constructing a completely new object, like in ctor, then value type / `const&` + `&&`;
  - For assigning to an object, like in setter, then it's your duty to think about the efficiency of future operations.
    - If reserving the current status is better, `const&` and copy may be enough;
    - Otherwise same as above.

Analyzing Performance

- Particularly, we're here assuming that the class is both copyable and moveable.
  - For a move-only class, the parameter can only be either the value type, or `&&`; and the latter always has lower cost.
  - Only one case to use value type: always discard the caller's ownership.
- These cases are all to assign the parameter directly to something; there are also other parameter cases:
  - Read-only: `const&`, or value type (if it's small enough);
  - Writable: `&`;
  - Create object and emit it out: by return value.
    - In C++98 it's usually `&`, but NRVO and implicit move make return value cheap and convenient.

Summary

- Value Category
  - lvalue, xvalue, prvalue
  - `decltype`, `decltype(auto)`
- Reference qualifier
  - Deducing this
- Copy Elision
  - prvalue copy elision
  - return value optimization
  - Implicit move
- Analyzing performance

Next lecture…

- Template Basics!
- About specialization, concepts, type deduction, etc.
- And back to move semantics, for universal reference.