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

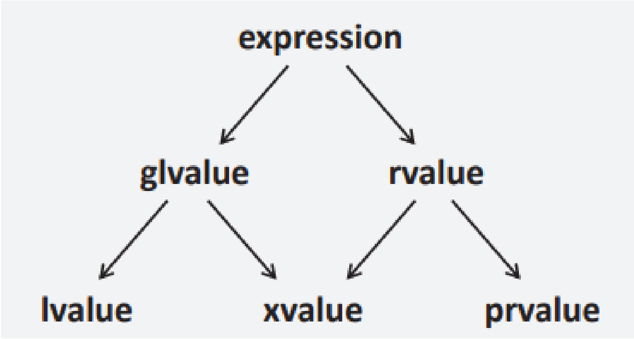
Value Category

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 prvalue but not rvalue 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++`;
- `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;
- `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];
- `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);
- a cast expression to non-reference type, such as `static_cast<double>(x)`, `std::string{}`, or `(int)42`;
- the `this` pointer;
- an enumerator;
- a non-type template parameter of a scalar type;

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

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

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

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

Conversion creates new object

Similar to literals

NTTP will be covered in the next lecture.

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);
- 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)`;

(since C++11)

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:

lvalue

The following expressions are *lvalue expressions*:

Named variables

- 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`;

Same as overloaded operators that commonly return reference type.

- `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);

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

- `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;

rvalue reference to function (but not important).

- 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)` $\Leftrightarrow$ `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)` $\not\Leftrightarrow$ `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&&`

```
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:

```
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?

```
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 ❌

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

Compile Okay?!

```
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!

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

```cpp
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:

```cpp
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
```

```cpp
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);
};
```