Moveing Obejcts

A reason to move rather than copy occurs in classes such as the IO or `unique_ptr` classes. These classes have a resource (such as a pointer or an IO buffer) that may not be shared. Hence, objects of these types cannot be copied but can be moved.

Rvalue References

To support move operations, the new standard introduced a new kind of reference, an ***rvalue reference***.

> An rvalue reference is a reference that must be bound to an rvalue. An rvalue references is obtained by using `&&` rather than `&`.

Rvalue references have the important property that they may be bound to an object that is about to be destroyed. As a result, we are free to "move" resources from an rvalue reference to another object.

> More details will be introduced in Section "**Rvalue and Lvalue**".

One saying to the definition of rvalue and lvalue is that, an lvalue expression refers to an object's identity whereas an rvalue expression refers to an object's value.

> There is more to say in terms of lavalu reference and rvalue reference in Section **Rvalue and Lvalue**.

```
int i=3;
int &lr=i; // ok, lr is an lvalue reference and can be bound to i
int &&rr=i;// error, cannot bind an rvalue reference to an lvalue

int &r2=i*42; // error:i*42 is an rvalue
const int &r3=i*42;// ok, we can bind a reference to const to an rvalue.

int &&rr2=i*42; // ok, bind rr2 to the result of the multiplication
```

Function that return lvalue references, along with the assignment, subscript, dereference, and prefix increment/decrement operators, are all examples of expressions that return lvalues. We can bind an lvalue reference to the result of any of these expressions.

`Example regrading assignment return lvalue reference is shown as below.`

```
int a=b;// copy-initialization, not assignment
int a,b=4;
(a=b)=3;// (a=b) return a rvalue reference bound to a
        // As a result, a will be 3, not 4;
```

`Example: subscript`:

```
int a[]={1,2,3};// vector<Int> a{1,2,3};
a[i]=3;// i=0,1,2 a[i] is a lvalue.
```

`Example: dereference`:

```
int a=3;
int *ra=&a;//
*ra=4;
```

Function that return a nonreference type, along with the arithmetic, relational, bitwise, and postfix increment/decrement opeartors, all yield rvalues. We cannot bind an lvalue reference to these expressions, but we can bind either an lvalue reference to `const` or an rvalue reference to such expressions.

Lvalue Persist; Rvalues Are Ephemeral

Lvalue have persistent state, whereas rvalues are either literals or temporary objects created in the course of evaluating expressions.

Because rvalue references can only be bound to temporaries, we know that,

- the referred-to object is about to be destroyed
- there can be no other users of that object.

> The code that uses an rvalue reference is free to take over resources from the object to which the reference refers.

Variables Are Lvalues

```
int &&rr1=42; // ok, literals are rvalues, rr1 is an lvalue
int &&rr2=rr1;// error, the expression rr1 is an lvalue
```

A variable is an expression with one operand and no operator (although we rarely think about it this way). Like any other expression, a variable expression has lvalue/rvalue property.

> Variable expressions are lvalues. We cannot bind an rvalue reference to a variable defined as an rvalue reference type.

The library `move` Function

Altough we cannot directly bind an rvalue reference to an lvalue, we can explicitly cast an lvalue to its corresponding rvalue reference type. We can also obtain an rvalue reference bound to an lvalue by calling a new library function named `move`, which is defined in the `utility` header.

```
int &&r1=42; // r1 is an lvalue with an rvalue reference type, can be bound to an
rvalue (literals)
int &&rr3=std::move(rr1); // ok
```

Calling `move` tells the compiler that we have an lvalue that we want to treat as if it were an rvalue.

It is essential to realize that the call to `move` promises that we do not intend to use `rr1` again except to assign to it or destroy it. After a call to `move`, we cannot make any assumptions about the value to the moved-from object.

Comment: the moved-from object may been destroyed, we have no idea that what the value to the move-from object is.

> We can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved-from object.

Move Constructor and Move Assignment

To enable move operations for our own types, we can define a move constructor and a move-assignment operator.

> The move constructor and move-assignment operator are similar to the corresponding copy operations, but ther "steal" resources from their given object rather than copy them.

Like the copy constructor, the move constructor has an initial parameter that is a reference to the class type, but is an rvalue reference. As in the copy constructor, any additional parameters must all have default arguments.

In addition to moving resources, the move constructor must ensure that the moved-from object is left in a state such that destroying that object will be harmless. In particular, once its resources are moved, the original object must no longer point to those moved resources-reponsibility for those resources has been assumed by the newly created object.

Example on `StrVec` defined before.

```
// noexcept indicates that move won't throw any exceptions
// member initializers take over the resources in s
StrVec::StrVec(StrVec
&&s)noexcept:elements(s.elements),first_free(s.first_free),cap(s.cap){
    // leave s in a state in which it is safe to run the destructor
    s.elements=s.first_free=s.cap=nullptr;
}
```

Unlike the copy constructor, the move constructor does not allocate any new memory; it takes over the memory in the given `StrVec`. Having taken over the memory from its argument, the constructor body sets the pointers in the given object to `nullptr`. After an object is moved from, the object continues to exist. Eventually, the moved-from object will be destroyed, meaning that the destructor will be run on that object. The `StrVec` destructor calls `deallocate` on `first_free`. If the neglected to change `s.first_free`, then destroying the moved-from object would delete the memory we just moved.

Move Operations, Library Containers, and Exceptions

A move operation ordinarily does not itself allocate any resources, since it executes by "stealing" resources. ***As a result, move operations ordinarily will not throw any exceptions.*** When we write a move operation that cannot throw, we should inform the library of that fact.

Thus, one way inform the library is to sepcify `noexcept` on our constructor. For now what's important to know is that `noexcept` is a way for us to promise that a function does not throw any exceptions.

```
class StrVec{
    public:
        StrVec(StrVec &&)noexcept; // move constructor
        // ...
};
StrVec::StrVec(StrVec &&s) noexcept:/*member initailizers*/
{
    // constructor boday
}
```

Understanding why `noexcept` is needed can help deeped our understanding of how the library interacts with objects of the types we write. We need to indicate that a move operation doesn't throw because of two interrelated facts:

- although move operations usually don't throw exceptions, they are permitted to do so.
- the library containers provide guarantees as to what they do if an exception happens.

Move-assignment Operator

The move-assignment operator does the same work as the destructor and the move constructor. As with the move constructor, if our move-assignment operator won't throw any exceptions, we should make it `noexcept`. Like a copy-assignment operator, a move-assignment operator must guard against self-assignment:

```
StrVec &StrVec::operator=(StrVec &&rhs)noexcept{
    // direct test for self-assignment
    if(this!=&rhs){
        free(); // free existing elements
        elements=rhs.elements; // take over resources from rhs
        first_free=rhs.first_free;
        cap=rhs.cap;
        // leave rhs in a destructible state

        rhs.elements=rhs.first_free=rhs.cap=nullptr;
    }
    return *this;
}
```

In this case, we check directly whether the `this` pointer and the address of `rhs` are the same.

We do the check because that rvalue could be the result of calling `move`. As in any other assignment operator, it is crucial that we not free the left-hand resources before using those (possibly same) resources from the right-hand operand.

A Moved-from Object Must Be Destructible

In addition to leaving the moved-from object in a state that is safe to destroy, move operations must guarantee that the object remains valid. In general, a valid object is one that can safely be given a new value or used in other ways that do not depend on its current value.

On the other hand, move operations have no requirements as to the value that remains in the moved-from object. As a result, our programs should never depend on the value of a moved-from object.

For example, when we move from `string` or other container object, we know that the moved-from object remains valid. As a result, we can run operations such as `empty` or `size` on moved-from objects. However, we have no idea what result we'll get. We might expect a moved-from object to be empty, but that is not guaranteed (as long as the moved-from is in a valid state).

> In a conclusion, after a move operation, the "moved-from" object must remains a valid, destructible object but users may make no asumption about its value.

The Synthesized Move Operations

The condition under which it systhesizes a move operation is different from those in which it systhesizes a copy operation.

Differently from the copy operations, for some classes the compiler does not synthesize the move operations at all. In particular, if a class defines its own copy constructor, copy-assignment operator, or destructor, ***the move constructor and move assignment operator are not synthesized.*** When a class doesn't have a move operation, the corresponding copy operation is used in place of move through function matching.

When?

The compiler will synthesize a move constructor or a move-assignment operator ***only*** if the class doen't define any of its own copy-control members and if every `nonstatic` data member of the class can be moved.

The compiler can move members of built-in type. It can also move members of a class type if the member's class has the corresponding move operation:

```
// the compiler will synthesize the move operation for X and hasX
struct X{
    int i; //built-in type can be moved
    std::string s;// string defines its own move operation
};

struct hasX{
    X mem; // X has systhesized move operation
};

X x,x2=std::move(x); // uses the synthesized move constructor
hasX hx,hx2=std::move(hx);
```

Unlike the copy operations, move operation is never implicitly defined as a deleted function[^1]. However, if we explicitly ask the compiler to generate a move operation by using `=default`, and the compiler is unable to move all the members, then the move operation will be defined as deleted. With one important exception, the rules for when a synthesized move operation is defined as deleted are analogous to those for the copy operations:
[^1]: Copy operation will be defined as a deleted function in some cases.

- the move constructor is defined as deleted if the class has a member that defines its own copy constructor but does not also define a move constructor, or if the class has a member that doesn't define its own copy operations and for which the compiler is unable to synthesize a move constructor. Similar for move-assgnment.
- the move constructor or move-assignment operator is defined as deleted if the class has a member whose own move constructor or move-assignment operator is deleted or inaccessible.
- like the copy constructor, the move constructor is defined as deleted if the destructor is deleted or inaccessible.
- like the copy-assignment operator, the move-assignment operator is defined as deleted if the class has a `const` or reference member.

```
// assume that Y is a class that defines its own copy constructor but not a move
constructor. Also, the compiler will not synthesize its move construct
struct hasY{
    hasY()=default;
    hasY(hasY &&)=default;
    Y mem; // hasY will have a deleted move constructor
};
hasY hy,hy2=std::move(hy); // error:move constructor is deleted
```

The compiler can copy objects of type `Y` but cannot move them. Class `hasY` explicitly requested a move consturctor, which the compiler is unable to generate. Hence, `hasY` will get a deleted move constructor.

There is one final interaction between move operations and the synthesized copy control members: Whether a class defines its own move operations has an impact on how the copy operations are synthesized. If the class defines either a move constructor and/or a move-assignment operator, then the synthesized copy constructor and copy-assignment operator for that class will be defined as deleted.

> Classes that define a move constructor or move-assignment operator must also define their own copy operations. Otherwise, those members are deleted by default.

Rvalues Are Moved, Lvalues Are Copies

When a class has both a move constructor and a copy constructor, the compiler uses ordinary function matching to determine which constructor to use. For example, in our `StrVec` class the copy versions take a reference to `const StrVec`. As a result, they can be used on any type that can be converted to `StrVec`. The move versions take a `StrVec &&` and can be used only when the argument is a (`nonconst`) rvalue.

```
StrVec v1,v2;
v1=v2; // v2 is an lvalue; use the copy assignment
```

```
StrVec getVec(istream &); // getVec returns an rvalue;

v2=getVec(cin);// getVec(cin) is an rvalue; move assignment
```

In the second assignment, we assign from the result of a call to `getVec`. That expression is an rvalue. In this case, both assignment operators are viable-we can bind the result of `getVec` to either operators'parameter. Calling the copy-assignment operator requires a consversion to const, whereas `StrVec&&` is an exact match. Hence, the second assignment uses the move-assignment operator.

But Rvalues Are Copied If There Is No Move Operator

What if a class has a copy constructor but does not define a move constructor? In this case, the compiler will not synthesize the move constructor, which means the class has a copy constructor but no move constructor. If a class has no move constructor, function matching easures that objects of that type are copied, even if we attempt to move them by calling `move`:

```
class Foo{
    public:
        Foo()=default;
        Foo(const Foo &);

};
Foo x;
Foo y(x);// copy constructor; x is an lvalue
Foo z(std::move(x));// copy constructor, because there is no move constructor
```

The call to `move(x)` in the initialization of `z` returns a `Foo &&` bound to `x`. The copy constructor for `Foo` is viable because we can convert a `Foo &&` to a `const Foo &`. Thus, ...

It is worth noting that using the copy constructor in place of a move constructor is almost surely safe (and similarly for the assignment operators). Ordinarily, the copy constructor will meet the requirements of the corresponding move constructor: It will copy the given object and leave that original object in a valid state. Indeed, the copy constructor won't even change the value of the original object.

Copy-and-Swap Assignment Operators and Move

If we add a move constructor to class of `hasPtr`, it will effectively get a move assignment operator as well:

```
class HasPtr{
    public:
        // default constructor
        HasPtr(const string &s=string(),const int &v=0):ps(new string(s)),val(v){}

        // copy constructor
        HasPtr(const HasPtr &s):ps(new string(s.ps)),val(s.val){}

        // copy assignment operator without swap operation
        HasPtr & operator=(const HasPtr &rhs){
            auto newdata=new string(*rhs.ps);
            delete this->ps;
            this->ps=newdata;
            this->val=rhs->val;
            return *this;
        }
        // added move constructor
        HasPtr(HasPtr &&p)noexcept:ps(p.ps),val(p.val){
            p.ps=0;// p.ps=nullptr;
        }
        // assignment operator is both the move-and copy-assignment operator
        HasPtr& operator=(HasPtr rhs){
            swap(*this,rhs);
            return *this;
        }

        ~HasPtr(){delete ps;}
    private:
        string *ps;
        int val;
        //
};
```

`HasPtr & operator=(HasPtr rhs)`. That operator has a nonreference parameter, which means the parameter is copy initialized. *Depending on the type of the argument, copy initialization uses either the copy constructor or the move constructor;* lvalus are copied and rvalues are moved. As a result, this single assignment operator acts as both the copy-assignment and move-assignment.

For example, assuming both `hp` and `hp2` are `HasPtr` objects:

```
hp=hp2; // hp2 is an lvalue; copy constructor usec to copy hp2
hp=std::move(hp2); // move constructor moves hp2
```

In the first assignment, the right-hand operand is an lvalue, so the move constructor is not viable. The copy constructor will be used to initialize `rhs`. The copy constructor will allocate a new `string` and copy the `string` to which `hp2` points.

In the second assignment, we invoke `std::move` to bind an rvalue reference to `hp2`. In this case, both the copy constructor and the move constructor are viable. However, because the argument is an rvalue reference, it is an exact match for the move constructor. The move constructor copies the pointer from `hp2`. It does not allocate any memory.

After the `swap`, `rhs` will hold a pointer to the `string` that had been owned by the left-hand side. That `string` will be destroyed when `rhs` goes out of scope.

> Note: Ordinarily, if a class defines any of these operations, it usualy should define them all. As we've seen, some classes must define the copy constructor, copy-assignment operator, and destructor to work correctly. Such classes typically have a resource that the copy members must copy. Ordinarily, coping a

> resource entails some amount of overhead. Classes that define the move constructor and move-assignment operator can avoid this overhead in those circumstance where a copy isn't necessary.

Move Iterators

The `reallocate` member will be wrote as below.

```
void StrVec::reallocate(){
    // allocate space for twice as many elements as the current size
    // if the StrVec is empty, we allocate room for one element.
    auto newcapacity=size()?2*size():1;

    auto newdata=alloc.allocate(newcapcity);
    auto dest=newdata;
    auto elem=elements;

    for(size_t i=0;i!=size();++i)
        alloc.construct(dest++,std::move(*elem++));
    // std::move(*elem++) is equivalent to std::move(*elem);++elem;

    free();// free the old space once we've moved the elements

    // update
    elements=newdata;
    first_free=dest;
    cap=elements+newcapacity;
}
```

The `reallocate` memebr of `StrVec` used a `for` loop to call `constructor` to copy the elements from the old memory to the new. As an alternative to writing that loop, it would be easier if we could call `uninitialized_cop` to construct the newly allocated space. However, `uninitialized_copy` does what it says: It copies the elements. There is no analogous library function to "move" objects into unconstructed memory.

Instead, the new library defines a **move iterator** adaptor. A move iterator adapts its given iterator by changing the behavior of the iterator's dereference operator. Ordinarily, an iterator dereference operator returns an lvalue reference to the element. Unlike other iterators, the dereference operator of a move iterator yields an rvalue reference.

We transform an ordinary iterator to a move iterator by calling the library `make_move_iterator` funtion. This function takes an iterator and returns a move iterator.

All of the original iterator's other operations work as usual. Because these iterators support normal iterator operations, we can pass a pair of move iterators to an algorithm. In particular, we can pass move iterators to `uninitialized_copy`:

```
void StrVec::reallocate(){
    // allocate space for twice as many elements as the current size
    // if the StrVec is empty, we allocate room for one element.
    auto newcapacity=size()?2*size():1;
    auto first=alloc.allocate(newcapcity);

    auto
last=unitialized_copy(make_move_iterator(begin()),make_move_iterator(end()),first)
;


    free(); // free the old space
    // update
    elements=first;
    first_free=last;
    cap=elements+newcapacity;
}
```

`unitialized_copy` calls `construct` on each element in the input sequence to "copy" that element into the destination. That algorithm uses the iterator dereference operator to fetch elements from the input sequence. Because we passed move iterators, the dereference operator yields an rvalue reference, which means `construct` will use the move constructor to construct the elements.

It is worth noting that standard library makes no guarantees about which algorithms can be used with move iterators and which cannot. Because moving an object can obliterate the source, you should pass move iterators to algorithms only when you are confident that the algorithm does not access an element after it has assigned to that element or passed that element to a use-defined function.

> Note: Because a moved-from object has indeterminate state, calling `std::move` on an object is a dangerous operation. When we call `move`, we must be absolutely certain that there can be no other uses of the moved-from object.

Rvalue References and Member Functions

Member functions other than constructors and assignnment can benefit from providing both copy and move versions. Such move-enabled members typically use the same paramter pattern as the copy/move constructor and the assignment operators-one version takes an lvalue reference to `const`, and the second takes an rvlaue referece to `noncconst`.

```
// the library containers that define push_back provide two versions
void push_back(const &T);// copy:binds to any kind of T
void push_back(T &&); // move:binds only to modifiable (nonconst) rvalues of type
T
```

The second version of `push_back` is an exact match for `nonconst` rvalues and will be run when we pass a modifiable rvalue. This version is free to steal resources from its parameter. Thus, the argument must be a modifiable object.

Ordinarily, there is no need to define versions of the operation that take a `const T &&` or a (plain) `T &`. Usually, we pass an rvalue reference when we want to "steal" from the argument. In order to do son, the argument must not be `const`. Similarly, copying from an object should not change the object being copied. As a result, there is usually no need to define a version that take a (plain) `T &` parameter.

> Overloaded functions that distinguish between moving and copying a parameter typically have one version that takes a `const T &` and one that takes a `T &&`.

As a more concrete example, we'll give our `StrVec` class a second version of `push_back`:

```
class StrVec{
    public:
        void push_back(const std::string &); // copy the element
        void push_back(std::string &&);// move the element
        // other memebrs as before
};

void StrVec::push_back(const std::string &s){
    chk_n_alloc(); // ensure that there is room for another element
    alloc.construct(first_free++,s);
}

void StrVec::push_back(std::string &&s){
    chk_n_alloc(); // reallocates the StrVec if necessary
    alloc.construct(first_free++,std::move(s));
}
```

The rvalue reference version of `push_back` calls `move` to pass its parameter to `construct`. As we've seen, the `construct` function uses the type of its second and subsequent arguments to determine which constructor to use. Because `move` returns an rvalue reference, the type fo the argument to `construct` is `string &&`. Therefore, the `stirng` move constructor will be used to construct a new last element.

When we call `push_back` the type of the argument determines whether the new element is copied or moved into the container:

```
StrVec vec; // empty StrVec
string s="some string or another";
vec.push_back(s); // calls push_back(const string &s);
vec.push_back("done"); // calls push_back(string &&); // literals are rvalues
```

These calls differ as to whether the argument is an lvalue or an rvalue (the temporary `string` created from "done"). The calls are resolved accordingly.

Rvalue and Lvalue Reference Member Functions

Ordinarily, we can call a member function on an object, regardless of whether that object is an lvalue or an rvalue. For example:

```
string s1="a value", s2="another";
auto n=(s1+s2).find('a');
```

Here, we called the `find` member on the `string` rvalue that results from adding two `string`s. Sometimes such usage can be suiprising:

```
s1+s2="wow!"; // ok, since this will use the string assignment operator, not built-in assignment operator
// equivalent statement
(s1+s2).operator=("wow");// s1+s2 is an rvalue, and it can call the member function (i.e, the assignment operator member function)
```

Here we assign to the rvalue result of concatentating these `string`s. Prior to the new standard, these was no way to prevent such usage. In order to maintain backward compatability, the library classes continue to allow assignment to rvalues. However, we might want to prevent such usage in our own classes. In this case, we'd like to force the left-hand operand (i.e., the object to which `this` points) to be an lvalue.

We indicate the lvalue/rvalue property of `this` in the same way that we define `const` member functions; we place a **reference qualifier** after the parameter list:

```
class Foo{
    public:
        Foo &operator=(const Foo &) &; // may assign only to modifiable lvalues
        // other members of Foo
};
Foo & Foo::operator=(const Foo &rhs) &{
    //...
    return *this;
}
```

We may run a function qualified by `&` only on an lvalue and may run a function qualified by `&&` only on an rvalue:

```
Foo &retFoo(); // returns a (an lvalue) reference; a call to retFoo is an lvalue
Foo retVal(); // returns by value (an rvalue); a call to retVal is an rvalue
Foo i,j; // iand j are lvalues;

i=j; // ok, i is an lvalue and can call the assignment operator

retFoo()=j;// ok, retFoo() returns an lvalue;

retVal()=j;// error, retVal() returns an rvalue;

i=retVal();// ok, we can pass an rvalue as the right-hand operand to assignment. The reason is the parameter of assignment operator function is "const Foo & rhs". Also, an const lvalue reference can be bound to an rvalue.
```

A function can be both `const` and reference qualified. In such cases, the reference qualifier mush follow the `const` qualifier. That is,

```
class Foo{
    public:
        Foo someMem() & const ;// error, const qualifier must come first
        Foo anotherMem)()const &; // ok, const qualifier comes first
}
```

Overloading and Reference Functions

Just as we can overload a member function based on whether it is `const`, that is.

```
class Foo{
    public:
        T func(Foo &f);
        // overloaded version
        T func(const Foo &f);

        Foo & function(){
            // do something
            return *this;
        };
        // overloaded version
        const Foo & function()const {
            return *this;
        }

};
```

Similarly, we can overload a function based on its reference qualifier. Moreover, we may overload a function by its reference qualifier and by whether it is a `const` member. As an example, we'give `Foo` a `vector` member and a function named `sorted` that returns a copy of the `Foo` object in which the `vector` is sorted:

```
class Foo{
    public:
        Foo sorted() &&; // may run on modifiable rvalues
        Foo sorted() const &; // may run on any kind of Foo
        // other members of Foo

    private:
        vector<int> data;
};

// this object is an rvalue, so we can sort in place
Foo Foo::sorted()&&{
    sort(data.begin(),data.end());
    return *this;
}

// this object is either const or it is an lvalue; either way we can't sort in
place
Foo Foo::sorted()const &{
    Foo ret(*this);// make a copy
    sort(ret.data.begin(),ret.data.end());// sort the copy
    return ret;// return the copy
}
```

When we run sorted on an rvalue, it is safe to sort the data member directly. The object is an rvalue, which means it has no other users, so we can change the object itself. When we run sorted on a const rvalue or on an lvalue, we can't change this object, so we copy data before sorting it.

Overload resolution uses the lvalue/rvalue property of the object that calls sorted to determine which version is used:

```
retVal().sorted(); // retVal() is an rvalue, calls Foo::sorted() &&
retFoo().sorted(); // retFoo() is an lvalue, calls Foo::sorted() const &
```

Some Remarkable Points

When we define const member functions, we can define two versions that differ only in that one is const qualified and the other is not. There is no similar default for reference qualified functions. When we difine two or more members that have the same name and the same parameter list, we must provide a reference qualifier (i.e., && or &) on all or none of those functions:

```
class Foo{
    public:
        Foo sorted() &&;
        Foo sorted() const ;// error:must have reference qulifier
        //Foo sorted() const &;// ok, may run on lvalues or const lvalues or
rvalues, i.e., any kind of Foo

        using Comp=bool(const int &,const int &);
        Foo sorted(Comp*);
        Foo sorted(Comp*)const; // ok
};
```

Why?

It is easily see that,

```
// ok, these two function can be distinguished by the compiler to call which
function.
```

```
Foo sorted();
Foo sorted() const;// may run on any lvalues (const and nonconst) of type Foo, but objects of type const Foo are the better function matching than Foo Foo::sorted().

Foo a;
const Foo b;
a.sorted();// calls Foo::sorted();
b.sorted();// calls Foo::sorted()const;
```

However, if we define the overloaded function as follows, the compiler cannot distinguish which one will be called.

1.

```
// error
Foo sorted() &;// may run on lvalues, cannot run on rvalues (an lvalue reference cannot be bound to an rvalue)
Foo sorted() const;// may run on lvalues.
```

2.

```
// error
Foo sorted() && ;
Foo sorted() const;
```

3.

```
// error
Foo sorted() &&;
Foo sorted() ;
```