Rvalueness, Move Semantics and Perfect Forwarding

Concepts

Rvalueness is introduced to solve two problems :

- move semantics and
- perfect forwarding for template only

Rvalueness and rvalue reference are two different concepts :

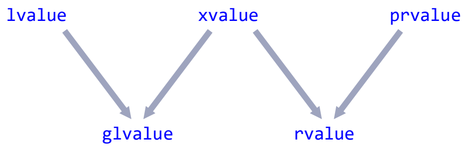
- rvalueness is an attribute of an expression denoting whether it is temporary
- rvalue reference is a variable that binds to a certain set of expressions

Reference

- Rvalue References, Move Semantics, Universal References, by Masaryk University
- C++ Rvalue References Explained, by Thomas Becker
- Universal References in C++11, by Scott Meyers
- New Value Terminology, by Bjarne Stroustrup
- What are Move Semantics, by Fred, Stack Overflow

A1. Motivation

The story begins with a class `T` which requires considerable effort to construct, copy and destruct. Suppose now we have an existing but temporary instantiation of it `obj0`, how can we make a clone `obj1` with minimum effort? We thus introduce the move semantics which basically permits `obj1` to steal the ownership of `obj0` resources without deep copying. However there is one inviolate rule for moving, that is ***"move only when it is unquestionably safe to do so"***. There are two possible cases, *(1)* when compiler knows that it is safe to move, which means that the source object is surely temporary or *(2)* when programmer tells compiler to do so, because he is gonna to give up ownership of the source object `obj0` for the sake of an efficient clone of the destination object `obj1`. The definitely temporary expression is `prvalue`, the programmer-told temporary expression is `xvalue`, and expression that belongs to neither cases is `lvalue`. `prvalue` and `xvalue` expressions are safe to move from, while `lvalue` expressions are not-safe-to-move-from, rvalueness is a classification according to the safe-to-move-from attribute of an expression.

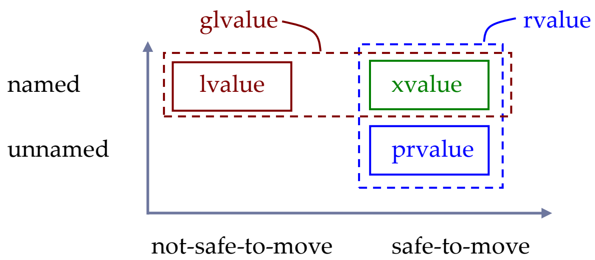


`prvalue = pure rvalue`

`glvalue = generalized lvalue`

According to Bjarne Stroustrup, the *5* classes can be considered as a classification according to :

- `lvalue` named and not-safe-to-move-from
- `xvalue` named and (claimed-to-be) safe-to-move-from
- `prvalue` unnamed and safe-to-move-from
- `rvalue` safe-to-move-from
- `glvalue` named



The three sets are mutually-exclusive and complementary. On top, we define supersets `rvalue` and `glvalue` as shown above. All C++ expressions are either `lvalue` or `rvalue`. Rvalueness is ***NOT* a type**, it is ***an attribute*** of an expression, it can either be :

- `lvalue` expression is not-safe-to-move-from (never ever try to move from it)
- `rvalue` expression is safe-to-move-from (move semantics for sake of speed)

A2. Rvalue reference

Given a function taking `movable_class` as argument, we may offer two overloads (allow users to pick their preferred one) :

```
void fct(const movable_class&  arg);          // copy semantic version, which keeps arg unchanged
void fct(      movable_class     );           // move semantic version, which moves arg content
```

Here comes the question, what kind of adornments should we put inside the red boxes? Firstly, we move resources from `arg`, hence there should be no `const`, secondly we need to pass by reference, not pass by value, yet we have to distinguish between reference to `lvalue` expression and reference to `rvalue` expression, hence C++ committee decided to use `T&` for binding to `lvalue` expressions, and use `T&&` for binding to `rvalue` expressions. The former is thus called `lvalue` reference, while the latter is called `rvalue` reference.

Thus in C++, we have 9 different types for class `T` by adding various adornments :

- `T*`
- `T* const`
- `const T*`
- `const T* const`
- `T[]`

pointer

- `const T&`
- `T&`
- `T&&`
- `T`

lvalue and rvalue reference

A3. Four rules that govern rvalueness

Now let us define `lvalue`, `xvalue` and `prvalue`. First, named expression is addressable implying that it may be referenced by pointers or iterators somewhere waiting to access its content, thus moving from named expression will result in dangling pointers, hence all named expressions are `lvalue`. Secondly, literals are `prvalue`, address of named variable is also temporary number in nature, thus it is considered as a literal and `pvalue`. The only exception is `"This is a string."`, which is surprisingly addressable, hence it is `lvalue`. The third rule is `static_cast` with which we can tell compiler that a particular named expression is in fact safe-to-move-from, hence it can turn `lvalue` into `xvalue`. Why not `prvalue`? Because it is not surely temporary, it is claimed-to-be temporary. The forth rule is a function that generates different various rvaluesness based on its (unnamed) return type.

```
class T
{
    const X&  factoryA() { return x;   }                // returns const lvalue
    X&        factoryB() { return x;   }                // returns       lvalue
    X&&       factoryC() { return static_cast<X&&>(x); }   // returns       xvalue (claimed-to-be temporary)
    X         factoryD() { return X(); }                // returns      prvalue (purely temporary)
    X x;
};
```

rvalueness	*rule 1-3*	*rule 4*
`lvalue` expression	all named variables, together with `"This is a string."`	`T&  f()`
`xvalue` expression	static cast `static_cast<T&&>(x)` and `std::move(x)`	`T&& f()`
`prvalue` expression	literals such as `123, 3.14, &object` which is `0x00001234`	`T   f()`

A4. Unnamed rvalue reference and named rvalue reference

There are 3 occasions in which `rvalue` reference of class `T` is used :

`T&& factory();`	declared as function return	this is unnamed `rvalue` reference (i.e. `factoryC` in rule 4)
`T&& x = ...;`	declared as object	this is named `rvalue` reference
`void fct(T&& x)`	declared as function argument	this is named `rvalue` reference

Unnamed return value from `factory` gives a `xvalue T` object, it is an approach to create `rvalue` expression, like `factoryC` in rule 4. On the contrary, expression `x` in the other two cases is ***NOT*** for creating `rvalue` expression, instead, it is used as a placeholder to bind to `rvalue` expressions for further manipulation (recall that binding is the original purpose of introducting `rvalue` reference to C++, read part A2). However as expression `x` is named, it is addressable, it is not-safe-to-move-from, in other word named `rvalue` reference is `lvalue` by itself and expression `x` effectively extends lifetime of that temporary object it binds. If we want to claim `rvalue` nature of `x` for further manipulation, we have to cast it by `std::move(x)`, i.e. using rule 2.

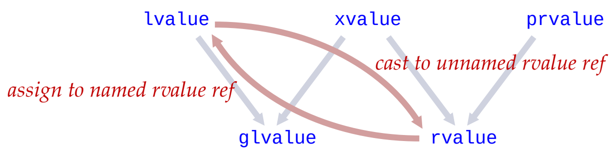
- 4.1 unnamed rvalue reference is used to *create rvalue expression*, see part A3
- 4.2 named rvalue reference is a placeholder that *binds to rvalue expression*, see part A5, however it is `lvalue` by itself

To summarise, we have :

- rvalueness of an expression is an attribute, denoting temporary and ready-to-be-moved,
- unnamed `rvalue` reference generates `rvalue` expression,
- named `rvalue` reference acts as a binding specification, specifying that it can bind to `rvalue` expression.

We have a mechanism to perform inter-conversion between `lvalue` and `rvalue` :

- extend lifetime of a `rvalue` expression by assigning it to a <u>named `rvalue` reference</u>
- move from a `lvalue` expression by static casting it to a <u>unnamed `rvalue` reference</u> using `std::move`



A5. Unnamed rvalue reference factory – avoid dangling

`Rvalue` reference is similar to `lvalue` reference, in that both are auto-dereferencing pointers, no explicit dereferencing by `*` is needed, both should be initialized on instantiation. Ensure that they aren't dangling when dereferenced, hence `factory0` and `factory3` crash.

```
// Please read B3 remark 3 for explanation of all factories.

T&  factory0()     { T x; return x; }            // [outcome 2] crash, return reference to temp obj, dangling
T&  factory1(T& x) {      return x; }            // [outcome 3] no crash, do nothing
T&& factory2()     { T x; return x; }            // [outcome 1] compile error, cannot bind T&& to lvalue variable
T&& factory3()     { T x; return std::move(x); } // [outcome 2] crash, return reference to temp obj, dangling
T&& factory4(T& x) {      return std::move(x); } // [outcome 3] no crash, do nothing
T   factory5()     { T x; return std::move(x); } // [outcome 4] ok, non-optimal, involves move
T   factory6()     { T x; return x; }            // [outcome 5] ok, optimal, named return value optimization kicks in
```

A6. Named rvalue reference binding

Binding of rvalue reference is different from that for lvalue reference (binding is the constraint on expression an identifier denotes) :

```
X global_x;

const X&  factoryA() { return global_x; }
X&        factoryB() { return global_x; }
X&&       factoryC() { return static_cast<X&&>(global_x); }
X         factoryD() { return X(); }

void fct(const X& x)   { std::cout << "\noverload1" << std::flush; } // binds to const-lvalue, lvalue and rvalue
void fct(X&  x)        { std::cout << "\noverload2" << std::flush; } // binds to              lvalue
void fct(X&& x)        { std::cout << "\noverload3" << std::flush; } // binds to                         rvalue
void fct(X   x)        { std::cout << "\noverload4" << std::flush; } // binds to const-lvalue, lvalue and rvalue
```

Existence of `overload4` usually results in ambiguity compile error. By removing it, we can resolve ambiguity.

	const lvalue exp	**lvalue expression**	**xvalue expression**	**prvalue expression**
rule1-3	const named obj	named obj	`static_cast<X&&>`	literal
rule4	`const X& factoryA()`	`X& factoryB()`	`X&& factoryC()`	`X factoryD()`
binds to `fct(const X& x)`	yes	yes	yes	yes
binds to `fct(X&  x)`	no	yes	no	no
binds to `fct(X&& x)`	no	no	yes	yes
binds to `fct(X   x)`	yes	yes	yes	yes

named rvalue ref   unnamed rvalue ref

A7. Rvalueness, rvalue reference, move semantics and movability

Do not confuse these 4 concepts :

- rvalue expressions are temporary objects, willing to give up ownership
- rvalue references are handles that can binds to temporary objects, telling others that the ownership may be taken away soon
- move semantics is triggered when we apply assignment on temporary objects
  move semantics is NOT triggered when we do member access
- movability describes whether move semantics is implemented for a class

```
void fct(const X&)
{
    process(x.mem0);            // ok no matter whether X is movable or non-movable
    process(x.mem1);            // ok no matter whether X is movable or non-movable
    X temp = x;                 // invoke X::operator=(const X&) for movable or non-movable X, compile error if it doesn't exist
}

void fct(X&&)
{
    process(x.mem0);            // ok no matter whether X is movable or non-movable
    process(x.mem1);            // ok no matter whether X is movable or non-movable
    X temp = std::move(x);      // invoke X::operator=(X&&) for movable X or X::operator=(const X&) for non-movable X
}
```

If function `fct` is not going to take away the ownership, please declare it as `void fct(const X&)`.

Some examples

```
X object;
X& lv_ref = object;
X&& rv_ref = X();
```

	rvalueness	*reasons*
`"This is a string."`	`lvalue`	addressable
`object`	`lvalue`	it is named
`lv_ref`	`lvalue`	it is named
`rv_ref`	`lvalue`	it is named
`X()`	`prvalue`	surely temporary
`123`	`prvalue`	literal
`'c'`	`prvalue`	literal
`&object`	`prvalue`	literal, since address = `0x0000ABCD`
`this`	`prvalue`	literal, since `this` is in fact a pointer
`static_cast<X&&>(object)`	`xvalue`	said-to-be temporary
`std::move(object)`	`xvalue`	said-to-be temporary (implementation is static_cast)

Reusing the factories in previous examples.

	rvalueness of RHS	*which binds to or invokes*
`X&  y0 = X();`	`prvalue`	compile error : can't assign rvalue obj to lvalue ref
`X&& y1 = X();`	`prvalue`	binding done, no constructor invoked
`X&  y2 = std::move(object);`	`xvalue`	compile error : can't assign rvalue obj to lvalue ref
`X&& y3 = object;`	`lvalue`	compile error : can't assign lvalue obj to rvalue ref
`X   z1 = factoryB();`	`lvalue`	invoke copy constructor `X::X(X& rhs)`
`X&  z2 = factoryB();`	`lvalue`	binding done, no constructor invoked
`X&& z3 = factoryB();`	`lvalue`	compile error : can't assign lvalue obj to rvalue ref
`X   z4 = factoryC();`	`xvalue`	invoke move constructor `X::X(X&& rhs)`
`X&  z5 = factoryC();`	`xvalue`	compile error : can't assign rvalue obj to lvalue ref
`X&& z6 = factoryC();`	`xvalue`	binding done, no constructor invoked
`X   z7 = factoryD();`	`prvalue`	invoke move constructor `X::X(X&& rhs)`
`X&  z8 = factoryD();`	`prvalue`	compile error : can't assign rvalue obj to lvalue ref
`X&& z9 = factoryD();`	`prvalue`	binding done, no constructor invoked

If we define multiplication operator as `X operator*(const X&, const X&) { return X(); }`

`X     a, b;`	-	-
`X&    c0 = a*b;`	`prvalue`	compile error : can't assign rvalue obj to lvalue ref
`X&&   c1 = a*b;`	`prvalue`	invoke `X operator*(const X&, const X&)`

If we define multiplication operator as `X& operator*(X& lhs, X& rhs) { return lhs; }`

`X     a, b;`	-	-
`X&    c0 = a*b;`	`lvalue`	invoke `X& operator*(X&, X&)`
`X&&   c1 = a*b;`	`lvalue`	compile error : can't assign lvalue obj to rvalue ref

B1. Methods for move semantics

How to implement a movable class? Six basic methods will be defined implicitly by compiler if we define nothing :

`array<T>::array();`	known as default constructor or `DC`
`array<T>::array(const array<T>&);`	known as copy constructor or `CC`
`array<T>::operator=(const array<T>&);`	known as copy assignment or `CA`
`array<T>::array(array<T>&&);`	known as move constructor or `MC`
`array<T>::operator=(array<T>&&);`	known as move assignment or `MA`
`array<T>::~array();`	known as default destructor or `DD`

We define the 6 methods explicitly according to rule of 3 and rule of 5 :

- rule of 3
  - define copy constructor / copy assignment / destructor all or none of them
  - do it if the class contains non-trivially deallocated resources *AND* if it is non-movable, don't touch them otherwise
- rule of 5
  - define copy constructor / copy assignment / move constructor / move assignment / destructor all or none of them
  - do it if the class contains non-trivially deallocated resources *AND* if it is movable, don't touch them otherwise

Compiler defines the 6 methods implicitly according to :

- default constructor is *NOT* implicitly define when
  - the class contains non-default-constructible components, or
  - the class contains user-defined constructor (i.e. `T::T()=default` automatically becomes `T::T()=delete`)
- copy constructor is *NOT* implicitly define when
  - the class contains non-copy-constructible components, or
  - the class contains user-defined move constructor / move assignment
- copy assignment is *NOT* implicitly define when
  - the class contains non-copy-assignable components, or
  - the class contains user-defined move constructor / move assignment
- move constructor is *NOT* implicitly define when
  - the class contains non-move-constructible components, or
  - the class contains user-defined copy constructor / copy assignment / move assignment / destructor
- move assignment is *NOT* implicitly define when
  - the class contains non-move-assignable components, or
  - the class contains user-defined copy constructor / copy assignment / move constructor / destructor

All 6 basic functions are summarised as :

```
                        DC   CC   CA   MC   MA   DD
No implicit DC if       x
No implicit CC if            x         x    x
No implicit CA if                 x    x    x
No implicit MC if            x    x    x    x    x
No implicit MA if            x    x    x    x    x
No implicit DD if
```

```
                T with ptr                   mov-array<T>        unique-ptr<T>       shared-ptr<T>
                                     DC =        NEW                 NEW                 NEW
CA(nonsafe)     DEL NEW CPY RET      CC =        NEW CPY                                 LHS ++n
CA(safe)        NEW SWP DEL RET      CA =    DEL NEW CPY RET                         --n LHS ++n RET
                                     MC =        LHS RHS             LHS RHS             LHS RHS
                                     MA =    DEL LHS RHS RET     DEL LHS RHS RET     --n LHS RHS RET
                                     DD =    DEL                 DEL                 --n
                                     CA+MA = CPY SWP     RET
                C++03.doc                    rvalue.doc          rvalue.doc          C++11.doc
```

```
template<typename T> class mov_array<T> // a glimpse into implementation
{
    DC   {              ptr = new T[];                                    }
    CC   {              ptr = new T[];    copy(rhs.ptr, ptr);             }
    CA   { del ptr;     ptr = new T[];    copy(rhs.ptr, ptr);    return *this; }
    MC   {              ptr = rhs.ptr;    rhs.ptr = nullptr;              }
    MA   { del ptr;     ptr = rhs.ptr;    rhs.ptr = nullptr;     return *this; }
    DD   { del ptr;                                               }
};
```

Constant member in class will disable implicit move semantics, since move semantics takes a `non-const rvalue` reference to `rhs` object as argument, stating that ownership will be taken away from `rhs`. Therefore if a member is declared `const`, implicit move is disabled (even if we declare move semantics as `=default`), unless we define move constructor and move assignment explicitly.

```
struct non_implicit_move { non_implicit_move& operator=(non_implicit_move&&) = default; const int mem; };
non_implicit_move x,y; x = std::move(y); // compile error
```

B2. Movable class `array` implementation

Please verify the following implementation with `g++-10`.

```
template<typename T> struct array
{
    // (1) size = max num of element T, not num of bytes
    // (2) use malloc (in heap) instead of new (in free-store), as no constructor is called
    array(int reserve = 1024) : capacity(reserve), size(0), ptr((char*) malloc(sizeof(T)*capacity))
    {
        // no need to call any constructor, avoid double-construction
    }

    array(const array<T>& rhs)
    {
        capacity = rhs.capacity;
        size     = rhs.size;
        ptr      = (char*) malloc(sizeof(T)*capacity);
        ::memcpy(ptr, rhs.ptr, sizeof(T)*size); // destination, source, num of bytes
    }

    array<T>& operator=(const array<T>& rhs)
    {
        if (ptr != nullptr) delete[] ptr;
        capacity = rhs.capacity;
        size     = rhs.size;
        ptr      = (char*) malloc(sizeof(T)*capacity);
        ::memcpy(ptr, rhs.ptr, sizeof(T)*size); // destination, source, num of bytes
        return *this;
    }

    array(array<T>&& rhs)
    {
        capacity = rhs.capacity;
        size     = rhs.size;
        ptr      = rhs.ptr;
        rhs.capacity = 0;
        rhs.size     = 0;
        rhs.ptr      = nullptr;
    }

    array<T>& operator=(array<T>&& rhs)
    {
        if (ptr != nullptr) delete[] ptr;
        capacity = rhs.capacity;
        size     = rhs.size;
        ptr      = rhs.ptr;
        rhs.capacity = 0;
        rhs.size     = 0;
        rhs.ptr      = nullptr;
        return *this;
    }

    // Alternative swap implementation that leads to non-deterministic destruction
    array<T>& operator=(array<T>&& rhs)
    {
        std::swap(capacity, rhs.capacity);
        std::swap(size,     rhs.size);
        std::swap(ptr,      rhs.ptr);
        return *this;
    }

    ~array() { if (ptr != nullptr) delete[] ptr; }

    // Implement later
    void push_back(const T& x);
    void push_back(T&& x);
    template<typename... ARGS> void emplace_back(ARGS&&... args);

    // Other functions
          T& operator[](int n)       { return (reinterpret_cast<T*>ptr)[n % capacity]; }
    const T& operator[](int n) const { return (reinterpret_cast<T*>ptr)[n % capacity]; }

    void print() const
    {
        if (ptr == nullptr) { std::cout << "nullptr"; return; }
        for(int n=0; n!=size; ++n) std::cout << ptr[n] << " ";
    }

    int capacity = 0;   // unit = element T
    int size = 0;       // unit = element T
    char* ptr = nullptr;
};

array<int> x0,x1,x2,x3;
for(int n=0; n!=1024; ++n) x0[n] = 100+n;   array<int> y0(x0);
for(int n=0; n!=1024; ++n) x1[n] = 200+n;   array<int> y1(std::move(x1));
for(int n=0; n!=1024; ++n) x2[n] = 300+n;   array<int> y2; y2 = x2;
for(int n=0; n!=1024; ++n) x3[n] = 400+n;   array<int> y3; y3 = std::move(x3);
```

B3. Movable class `array` remarks

Remark 1 : copy-and-swap idiom

We can merge copy assignment and move assignment into one assigment that binds to both `lvalue` and `rvalue` expression known as *copy and swap idiom* as the following :

```
Array<T>& array<T>::operator=(array<T> rhs) // Dont confuse with alternative implementation of array<T>& operator=(array<T>&& rhs)
{
    std::swap(capacity, rhs.capacity);
    std::swap(size,     rhs.size);
    std::swap(ptr,      rhs.ptr);
    return *this;
}
x0.print(); // 100 101 102 103 ...    y0.print(); // 100 101 102 103 ...
x1.print(); // nullptr                y1.print(); // 200 201 202 203 ...        move constructor sets rhs to nullptr
x2.print(); // 300 301 302 303 ...    y2.print(); // 300 301 302 303 ...
x3.print(); // nullptr                y3.print(); // 400 401 402 403 ...        move assignment sets rhs to nullptr too
```

When this assignment is invoked with `lvalue` argument, it triggers :

- a copy construction of `rhs` from `x2` (see the code in previous page) followed by
- a swap between `rhs.ptr` and `this->ptr` (which is `y2.ptr`)
- finally `rhs` goes out of scope and is destructed

When this assignment is invoked with `rvalue` argument, it triggers :

- a move construction of `rhs` from `x3`, which swaps between `rhs.ptr` and `x3.ptr`, `x3.ptr` now becomes `nullptr`
- a swap between `rhs.ptr` and `this->ptr` (which is `y3.ptr`)
- finally `rhs` goes out of scope and is destructed

Remark 2 : non-deterministic destruction

Suppose we implement move assigment using `swap-implementation` for `array<T>`, and if we perform the following :

```
array<T> a,b,c,d;
a = std::move(b);                    // line 1 : original resource in a is now moved to b
b = std::move(c);                    // line 2 : original resource in a is now moved to c
c = std::move(d);                    // line 3 : original resource in a is now moved to d
```

When `a` is assigned with `b`, the original resource owned by `a` is *NOT* deallocated in *line 1*, instead it is moved to `b`, and destruction is delayed until `b` goes out of scope. If this process is repeated, then the destruction will be delayed until `c` or `d` goes out of scope. This is called delayed destruction or non-deterministic destruction, which happens when we implement assignment of movable class by *copy-and-swap idiom*. Delayed destruction can be a problem sometimes, in particular for RAII, such as mutex.

Remark 3 : Factory and return value optimization

Lets design a factory for movable class `array`, if we implement it in various ways (exactly the same as those in A5) :

```
array<T>   global_x;
array<T>&  factory0(){ array<T> x; return x;                  } // useless, not a factory at all
array<T>&  factory1(){             return global_x;           } // useless, not a factory at all
array<T>&& factory2(){ array<T> x; return x;                  } // compile error, cannot bind array<T>&& to lvalue x
array<T>&& factory3(){ array<T> x; return std::move(x);       } // crash, return dangling pointer
array<T>&& factory4(){             return std::move(global_x); } // casting global_x as rvalue
array<T>   factory5(){ array<T> x; return std::move(x);       } // construct as temporary, then moved to callee
array<T>   factory6(){ array<T> x; return x;                  } // construct directly on callee

array<T> x2 = factory2(); x2.print();  // compile error
array<T> x3 = factory3(); x3.print();  // compile OK, but crash
array<T> x4 = factory4(); x4.print();  // stealing ownership of global_x may not be what we want
array<T> x5 = factory5(); x5.print();  // construct on x, then moved to x5
array<T> x6 = factory6(); x6.print();  // construct on x6 directly (return value optimization)
```

- declared return type must be able to bind to return statement, otherwise there will be compile error, such as `factory2`
- return reference to temporary variable will end up with dangling pointer and crash, such as `factory0` and `factory3`
- return reference to non-temporary variable does not crash, such as `factory1` and `factory4`
- return by value binding to `rvalue std::move(x)` results in a move, however this is not the optimal choice
- return by value binding to `lvalue x` results in direct construction on callee's variable `x6`, called **return value optimization**

Remark 4 : Forcing moving semantics using `std::move`

Now, lets add two `push_back` to class `array<T>`, one `lvalue` version as well as one `rvalue` version :

```
template<typename T> void array<T>::push_back(const T& x)
{
    (*this)[size % capacity] = x;
    ++size;
}

template<typename T> void array<T>::push_back(T&& x)
{
    (*this)[size % capacity] = std::move(x); // line1
    ++size;
}

array<T> my_array; T x;
my_array.push_back(x);                  // invoke push_back(const T&) for lvalue input
my_array.push_back(std::move(x));       // invoke push_back(T&&) for xvalue input
my_array.push_back(T{});                // invoke push_back(T&&) for prvalue input
```

Wrapper function `std::move` is necessary for the implementation of `rvalue` version in `line1` as we need to static-cast `lvalue` variable `x` into `rvalue`, so as to invoke move semantics instead of copy semantics, this is called **forcing move semantics**. Furthermore, remark 5 shows another forcing move semantics.

Remark 5 : Forcing moving semantics using `std::move_if_noexcept` for exception safety

However the move constructor `T::T(T&&)` invoked in `line1` may throw. If an object `x` is partially moved when `T::T(T&&)` throws, then `array<T>` fails to guarantee strong exception safety, as both `array<T>` and `x` cannot restore original states. If we want to make `array<T>` strong exception safe, like `std::vector`, we have to replace *unconditional cast* `std::move` by *conditional cast* `std::move_if_noexcept`.

```
template<typename T> void array<T>::push_back(T&& x)
{
    (*this)[size % capacity] = std::move_if_noexcept(x);
    ++size;
}
```

Conditional cast `std::move_if_noexcept` casts an expression as `rvalue` only when move constructor of `T` is declared `noexcept` or when `T` is non-copy-constructible, otherwise the expression is casted as `lvalue` and invokes copy semantics.

Problems for this implementation

- double construction of element `T`, requires `T` to be default constructible
- no emplace construction of element `T`
- separated implementation for copy and move `push_back`

C1. Motivation - *Why do we need perfect forwarding?* 24,3x3x3

Let's define our problem. We want to implement a template algorithm function `fct`, which delegates to various `impl` overloads. We wish to invoke `lvalue` implementation for `lvalue` input, invoke `rvalue` implementation for `rvalue` input respectively. This problem is called perfect forwarding, which happens in template programming. Which implementation `fct0-fct3` should we adopt?

```
template<typename T> int fct0(const T& x) { return impl(x);            } // fails as it always invoke lvalue implementation
template<typename T> int fct1(const T& x) { return impl(std::move(x)); } // fails as it always invoke rvalue implementation
template<typename T> int fct2(      T  x) { return impl(x);            } // fails as it always invoke lvalue implementation
template<typename T> int fct3(      T  x) { return impl(std::move(x)); } // fails as it always invoke rvalue implementation

int impl(const array<int>&  x) { std::cout << "lvalue implementation for array"; }
int impl(      array<int>&& x) { std::cout << "rvalue implementation for array"; }
```

1.1 The challenges are that `fct` needs to handle :

- bind to both `lvalue` and `rvalue` argument
- forward `x` to `impl` appropriately, however
- - `static_cast<T&>(x)` is always forwarded as `lvalue`
- - `static_cast<T&&>(x)` is always forwarded as `rvalue`

1.2 Solution is to make use of :

→ universal reference `T&&` (it is not `rvalue` reference) and
→ conditional cast `std::forward<T>`

```
    template<typename T> int fct(T&& x) { T& t0; T t1[100]; return impl(std::forward<T>(x)); } // solution to perfect forwarding
//  template<typename T> int fct(T&& x) { T& t0; T t1[100]; return impl(std::forward   (x)); } // compile error : deduction fails

    std::vector<my_class> x;
    fct(x);              // output = lvalue implementation
    fct(std::move(x)); // output = rvalue implementation
```

Explanation

1. First of all, understand the terminology, there are 3 different types :
- expression type is concrete type of input expression on template invocation
- template type is `T` in template declaration
- argument types are types `[T*, const T&, T&, T&&, T]` of various variables in template prototype / definition body

2. On invocation of template function `fct(x)` with :
- `lvalue` expression `x`, compiler resolves template type `T` ⇒ `X&` or equivalently resolves argument type `T&&` ⇒ `X&`
- `xvalue` expression `x`, compiler resolves template type `T` ⇒ `X` (NOT `T` ⇒ `X&&`) or equivalently resolves argument type `T&&` ⇒ `X&&`
- `prvalue` expression `x`, compiler resolves template type `T` ⇒ `X` also or equivalently resolves argument type `T&&` ⇒ `X&&`

= please refer to <u>*type deduction of universal reference*</u> for details

3. `std::forward<T>(x)` is a conditional cast such that :
- when `T` is resolved as lvalue `X&`, then `std::forward<T>(x)` is equivalent to `static_cast<X&>(x)`
- when `T` is resolved as xvalue `X&&`, then `std::forward<T>(x)` is equivalent to `static_cast<X&&>(x)`
- when `T` is resolved as prvalue `X`, then `std::forward<T>(x)` is also equivalent to `static_cast<X&&>(x)`
- as static cast depends on `T`, it must be supplied explicitly to `std::forward<T>(x)`

4. To understand this solution to perfect forwarding, we have to go through 3 steps :
- identifying universal reference
- type deduction of universal reference
- appropriate use of `std::forward<T>(x)`

Summary

expression	template	argument
E = X&	T = X&	T&& = X&
E = X&&	T = X	T&& = X&&
E = X	T = X	T&& = X&&

expression	template	casting output
E = X&	-	static_cast<X&&>(expression)
E = X&&	-	static_cast<X&&>(expression)
E = X	-	static_cast<X&&>(expression)

expression	template	casting output
E = X&	T = X&	static_cast<X&> (expression)
E = X&	T = X&&	static_cast<X&&>(expression)
E = X&	T = X	static_cast<X&&>(expression)

C2. Identifying universal reference

Both `rvalue` reference and universal reference are denoted as double ampersand, `&&` means universal reference when :

1. type deduction is involved *AND*
2. taking the form `T&&` or `auto&&` (even `const T&&` makes it a `rvalue` reference).

→ otherwise double ampersand means `rvalue` reference

```
std::vector<int> vec{1,2,3,4,5};
int    n  = 123;                    //    rvalue reference, as no type deduction
auto&& n0 = 123;                    // universal reference, as there is type deduction
auto&& n1 = n;                      // universal reference, as there is type deduction
auto&& n2 = std::move(n);           // universal reference, as there is type deduction

// template function
template<typename T> void fct0(T&& arg);
template<typename T> void fct1(const T&& arg);
template<typename T> void fct2(std::vector<T>&& arg);

fct0(123);                          // universal reference, as there is type deduction
fct0(n);                            // universal reference, as there is type deduction
fct0(std::move(n));                 // universal reference, as there is type deduction
fct1(vec);                          //    rvalue reference, as type deduction is not in form of T&&
fct2(vec);                          //    rvalue reference, as type deduction is not in form of T&&

// template class
template<typename T> struct classA
{
    void add(T&& data);
    void cmp(classA<T>&& rhs);
    template<typename ARG> void fct(ARG&& arg);
};

classA x,y;
x.add(data);                        //    rvalue reference, as no type deduction [T is known in obj declaration]
x.cmp(y);                           //    rvalue reference, as no type deduction [T is known in obj declaration]
x.fct(123);                         // universal reference, as there is type deduction [T is independent of ARG]
```

C3. Type deduction of universal reference

Steps involved in type deduction for universal reference :

1. ***reference stripping*** of expression type (equivalent to calling `remove_reference<...>::type`), suppose the result is `X`
2. resolve template type `T` to `X&` for `lvalue` invocation argument
   resolve template type `T` to `X` for `rvalue` invocation argument (including both `xvalue` and `prvalue`)
3. substitute previous result into every argument type, perform ***reference collapsing***, there may be multi-instances inside function

```
if (is_lvalue(expression_type))      template_type T = reference_stripping(expression_type)&
else                                 template_type T = reference_stripping(expression_type)

argument_type = reference_collapsing(T plus adornments) // there are multi-instances with different adornments
```

Reference collapsing rules are :

```
X   + &       is interpreted as X&
X   + &&      is interpreted as X&&
X&  + &       is interpreted as X&
X&& + &       is interpreted as X&
X&  + &&      is interpreted as X&
X&& + &&      is interpreted as X&&
```

Example

```
template<typename T> void fct(T&& arg);
int    a  = 1;
int&   b  = 2;
int&&  c  = 3;

auto&& u0 = a;                      //  lvalue expression, auto = strip(int)&   = int&, ARG = auto&& = int& && = int&
auto&& u1 = b;                      //  lvalue expression, auto = strip(int&)&  = int&, ARG = auto&& = int& && = int&
auto&& u2 = c;                      //  lvalue expression, auto = strip(int&&)& = int&, ARG = auto&& = int& && = int&
auto&& u3 = std::move(a);           //  xvalue expression, auto = strip(int&&)  = int,  ARG = auto&& = int  && = int&&
auto&& u4 = 123;                    // prvalue expression, auto = strip(int)    = int,  ARG = auto&& = int  && = int&&

fct(a);                             //  lvalue expression, T = strip(int)&   = int&, ARG = T&& = int& && = int&
fct(b);                             //  lvalue expression, T = strip(int&)&  = int&, ARG = T&& = int& && = int&
fct(c);                             //  lvalue expression, T = strip(int&&)& = int&, ARG = T&& = int& && = int&
fct(std::move(a));                  //  xvalue expression, T = strip(int&&)  = int,  ARG = T&& = int  && = int&&
fct(123);                           // prvalue expression, T = strip(int)    = int,  ARG = T&& = int  && = int&&
```

C4. Appropriate use of forward (Implementation of move and forward)

- `std::move` is an unconditional static cast, that casts universal reference to named expression into `rvalue` :
- if universal reference binds to `lvalue` expression, it casts the `lvalue` expression to `rvalue` (`xvalue` to be precise)
- if universal reference binds to `rvalue` expression, it casts the `rvalue` expression to `rvalue` (`xvalue` to be precise)
- `std::move_if_noexcept` is similar, except that it is conditional on `noexcept` guarantee, read B3 remark 5
- `std::forward` is a conditional static cast, which is used inside **another template function** with same template parameter :
- it casts `lvalue` expression into `lvalue` on giving `T=X&`
- it casts `lvalue` expression into `rvalue` (`xvalue` to be precise) on giving `T=X&&`
- it casts `lvalue` expression into `rvalue` (`xvalue` to be precise) on giving `T=X`

```
// Implementations in Thomas Becker's blog
template<typename T> typename remove_reference<T>::type&& std::move(T&& x) noexcept
{
    return static_cast<typename remove_reference<T>::type&&>(x);
}

template<typename T> T&& std::forward(typename remove_reference<T>::type& x) noexcept
{
    return static_cast<T&&>(x);
}
```

Verification of move

When `std::move` is invoked with `lvalue` expression of type `X`

```
// with step 1 and step 2, compiler generates :
typename remove_reference<X&>::type&& std::move(X& && x) noexcept
{
    return static_cast<typename remove_reference<X&>::type&&>(x);
}
// with step 3, compiler generates :
X&& std::move(X& x) noexcept
{
    return static_cast<X&&>(x);
}
```

When `std::move` is invoked with `rvalue` expression of type `X`

```
// with step 1 and step 2, compiler generates :
typename remove_reference<X>::type&& std::move(X&& x) noexcept
{
    return static_cast<typename remove_reference<X>::type&&>(x);
}
// with step 3, compiler generates :
X&& std::move(X&& x) noexcept
{
    return static_cast<X&&>(x);
}
```

Verification of forward

When template function `f` containing `std::forward` is invoked with `lvalue` expression of type `X`

```
// with step 1 and step 2, compiler generates :
X& && std::forward(typename remove_reference<X&>::type& x) noexcept
{
    return static_cast<X& &&>(x);
}
// with step 3, compiler generates :
X& std::forward(X& x) noexcept
{
    return static_cast<X&>(x);
}
```

When template function `f` containing `std::forward` is invoked with `rvalue` expression of type `X`

```
// with step 1 and step 2, compiler generates :
X&& std::forward(typename remove_reference<X>::type& x) noexcept
{
    return static_cast<X&&>(x);
}
// with step 3, compiler generates :
X&& std::forward(X& x) noexcept
{
    return static_cast<X&&>(x);
}
```

C5. Naïve implementation of emplace

Lets continue with our class `array<T>` to support emplace.

```
template<typename T> class array
{
    template<typename X, typename Y, typename Z> void emplace_back(X&& x, Y&& y, Z&& z) // universal reference
    {
        new (ptr + sizeof(T)*size) T{std::forward<X>(x), std::forward<Y>(y), std::forward<Z>(z)};
        ++size;
    }
    template<typename... ARGS> void emplace_back(ARGS&&... args) // universal reference
    {
        new (ptr + sizeof(T)*size) T{std::forward<ARGS>(args)...};
        ++size;
    }
}
```

1. Emplace means insertion inplace without copy semantics nor move semantics (involves ***delayed construction***)

2. Emplace implementation usually involves 3 techniques :
- placement `new` to decouple allocation from construction
- variadic template as the number of arguments varies case by case
- universal reference of input arguments, forwarded by `std::forward` as they may be temporary

3. Movability of container `array<T>`, element `T` and its raw material `XYZ` are independent. In our example :
- `array<T>` is movable, but not critical here
- `T` movability is don't care
- `XYZ` movability is considered with universal reference

C6. Naïve implementation of unique pointer

Unique pointer is a smart pointer that implements move semantics while forbids copy semantics. It triggers transfer of ownership.

```
template <typename T> class unique_ptr
{
    explicit unique_ptr(T* p = nullptr) { ptr = p; }
    ~unique_ptr() { if (ptr!=nullptr) delete ptr; }

    unique_ptr(unique_ptr&& rhs)
    {
        ptr = rhs.ptr;
        rhs.ptr = null_ptr;
    }

    unique_ptr& operator=(unique_ptr&& rhs)
    {
        if (ptr!=nullptr) delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = null_ptr;
        return *this;
    }

    T& operator*()  const { return *ptr; }
    T* operator->() const { return  ptr; }
    T* ptr = nullptr;
}
```

They are the same, also same as the one in `array<T>`

1. Constructing unqiue pointer with factory (using univeral reference and `std::forward`)

```
template<typename T, typename X, typename Y, typename Z> unique_ptr<T> make_unique(X&& x, Y&& y, Z&& z)
{
    return unique_ptr<T>(new T{ std::forward<X>(x), std::forward<Y>(y), std::forward<Z>(z) });
}
template<typename T, typename... ARGS> unique_ptr<T> make_unique(ARGS&&... args)
{
    return unique_ptr<T>(new T{ std::forward<ARGS>(args)... });
}
```

2. Constructing unique pointer with `lvalue` input results in compile error :

```
unique_ptr<SHAPE> shape0;
unique_ptr<SHAPE> shape1(shape0);             //  lvalue, compile error
unique_ptr<SHAPE> shape2(std::move(shape1));  //  xvalue, OK
```

3. Passing unique pointers to functions

- There are two types of functions taking unique pointer as input :
- function that does move the ownership, such as `f0` (correct approach) and `f1`
- function that does not move the ownership, such as `g0` and `g1` (correct approach)

```
void f0(std::unique_ptr<std::string> up) // accept rvalue input only
{
    auto tmp = std::move(up); // compile error without std::move
    std::cout << "\nf0 = " << *tmp;
}
void f1(std::unique_ptr<std::string>& up) // accept lvalue input only
{
    auto tmp = std::move(up); // compile error without std::move
    std::cout << "\nf1 = " << *tmp << " [taking ownership from lvalue is dangerous]";
}
void g0(std::unique_ptr<std::string> up) // accept rvalue input only
{
    std::cout << "\ng0 = " << *up << " [no one takes ownership, resource is destructed]";
}
void g1(std::unique_ptr<std::string>& up) // accept lvalue input only
{
    std::cout << "\ng1 = " << *up;
}

auto up0 = std::make_unique<std::string>("This is test0.");
auto up1 = std::make_unique<std::string>("This is test1.");
auto up2 = std::make_unique<std::string>("This is test2.");
auto up3 = std::make_unique<std::string>("This is test3.");
auto up4 = std::make_unique<std::string>("This is test4.");
auto up5 = std::make_unique<std::string>("This is test5.");

    f0(std::move(up0));
//  f0(up0);            // compile error
//  f1(std::move(up1)); // compile error
    f1(up1);
    g0(std::move(up2));
//  g0(up2);            // compile error
//  g1(std::move(up3)); // compile error
    g1(up3);
```

C7. Copyability, movability and `=delete =default`

- When we declare copy constructor and copy assignment `=delete`, the class becomes non-copyable and non-movable.
- When we declare move constructor and move assignement `=default` on top of above, the class becomes movable.

Why? Since implicit move constructor is disabled when copy constructor is declared.

```
struct A
{
    A() = default;
    A(const A&) = delete;
    A& operator=(const A&) = delete;
};

struct B
{
    B() = default;
    B(const B&) = delete;
    B& operator=(const B&) = delete;
    B(B&&) = default;
    B& operator=(B&&) = default;
};

std::cout << std::is_copy_constructible<A>::value << std::is_copy_constructible<B>::value; // False False
std::cout << std::is_copy_assignable   <A>::value << std::is_copy_assignable   <B>::value; // False False
std::cout << std::is_move_constructible<A>::value << std::is_move_constructible<B>::value; // False True
std::cout << std::is_move_assignable   <A>::value << std::is_move_assignable   <B>::value; // False True
```

- If we declare non-copyable member in a class, the class auto becomes non-copyable (even declare copy `cstr` as `=default`).
- If we declare non-movable member in a class, the class auto becomes non-movable (even declare move `cstr` as `=default`).
- If we declare reference member in a class, default `cstr` and two assignments are auto deleted (copy/move `cstr` are retained).
- Refer to `C++01.doc`, `std::vector<T>` requires `T` to be copyable, thus `T` cannot contain non-copyable members like `mutex` or `atomic`.

Some useful traits for rvalue reference (please check the relation between traits and C++ concepts) :

```
std::is_same<T,U>::value
std::is_assignable<T>::type
std::is_copy_assignable<T>::value
std::is_move_assignable<T>::value
std::add_lvalue_reference<T>::type
std::add_rvalue_reference<T>::type
std::remove_reference<T>::type
```

Appendix 1 : Number of constructor invocations in push-back / emplace-back

This test has been done in MSVS2019. Given a movable structure for testing :

```
struct X
{
    X()                      : a(1), b(2), c(3)                { std::cout << "constructor0 "; }
    X(int a, int b, int c) : a(a), b(b), c(c)                  { std::cout << "constructor1 "; }
    X(const X&  rhs)         : a(rhs.a), b(rhs.b), c(rhs.c)    { std::cout << "constructor2 "; }
    X(      X&& rhs)         : a(rhs.a), b(rhs.b), c(rhs.c)    { std::cout << "constructor3 "; }
    ~X()                                                       { std::cout << " destructor_ "   << a << b << c; }
    const X& operator=(const X&  rhs) { a=rhs.a; b=rhs.b; c=rhs.c; std::cout << "copy-assignment "; return *this; }
    const X& operator=(      X&& rhs) { a=rhs.a; b=rhs.b; c=rhs.c; std::cout << "move-assignment "; return *this; }

    int a; int b; int c;
};
```

[Step 1 - STL vector]

Now we create a vector. Let's count the number of constructor invocations in each case. The final `emplace_back(6,7,8)` doesn't involve any copy nor move, while other `push_back` and `emplace_back` involve construction of a temporary object, followed by a copy or a move.

```
std::vector<X> vecA;
vecA.reserve(10);                                                  // doesn't invoke constructor 0
std::cout << "trial 0 : "; X x0(2,3,4); vecA.push_back   (x0);       // invokes constructor 1,2
std::cout << "trial 1 : ";              vecA.push_back   (X{3,4,5}); // invokes constructor 1,3, destructor
std::cout << "trial 2 : "; X x1(4,5,6); vecA.emplace_back(x1);       // invokes constructor 1,2
std::cout << "trial 3 : ";              vecA.emplace_back(X{5,6,7}); // invokes constructor 1,3, destructor
std::cout << "trial 4 : ";              vecA.emplace_back  (6,7,8);  // invokes constructor 1 (no copy nor move)
                                                                   // invokes destructor  7 times (x0, x1 and 5 in vector)
```

[Step 2 - Incorrect implementation of vector]

The following naive vector implementation does not correctly implement `emplace_back`, as :

- its `T impl[N]` requires `T` to be ***default constructible*** (unnecessary), it invokes many default constructer at the beginning (slow)
- its `emplace_back` (in `approach 1`) invokes copy or move assignment of `T`, a real emplace must not involve copy or move
- its `emplace_back` (in `approach 2`) invokes double construction of `T`, which is a dangerous thing
- though `approach 2` is slightly better, as it decouples stack memory allocation and explicit construction by placement new

```
template<typename T, std::uint32_t N> struct my_vec_fail
{
    void push_back(const T& rhs) // for lvalue
    {
        impl[size] = rhs;
        ++size;
    }

    void push_back(T&& rhs) // for rvalue reference, not universal reference
    {
        impl[size] = std::move(rhs);
        ++size;
    }

    template<typename... ARGS> void emplace_back(ARGS&&... args) // for universal reference
    {
              impl[size] = T(std::forward<ARGS>(args)...); // approach 1 : not a real emplace, as it involves copy or move
    //   new (impl+size)   T(std::forward<ARGS>(args)...); // approach 2 : double construction
        ++size;
    }

    T impl[N];
    std::uint8_t size = 0;
};

my_vec_fail<X> vecB;                                               // invokes constructor 0 for 5 times
std::cout << "trial 0 : "; X x2(2,3,4); vecB.push_back   (x2);       // invokes constructor 1,   copy-assignment
std::cout << "trial 1 : ";              vecB.push_back   (X{3,4,5}); // invokes constructor 1,   move-assignment, destructor
std::cout << "trial 2 : "; X x3(4,5,6); vecB.emplace_back(x3);       // invokes constructor 1,2, move-assignment, destructor
std::cout << "trial 3 : ";              vecB.emplace_back(X{5,6,7}); // invokes constructor 1,3, move-assignment, destructor x 2
std::cout << "trial 4 : ";              vecB.emplace_back(  6,7,8);  // invokes constructor 1,   move-assignment, destructor
                                                                   // invokes destructor  7 times (x0, x1 and 5 in vector)
```

[Step 3 - Correct implementation of vector]

Can we build one which behaves in the same way as `std::vector`? Yes, if we do the following :

- replace `T impl[N]` by `char impl[N*sizeof(T)]` (not to use heap memory `char* impl` for low latency) or
- replace `T impl[N]` by `char impl[N*sizeof(T)+N*sizeof(M)]` if we want to include meta data (like atomic flag in `lockfree_mpmcq`)
- ► which can avoid : (1) unnecessary `std::default_initializable` requirement on `T`
  (2) dangerous double construction
  (3) copy or move that make emplace no longer an emplace

```
template<typename T, typename M, std::uint32_t N> struct my_vec
{
    ~my_vec() // explicit destruction is also needed
    {
        for(std::uint8_t n=0; n!=size; ++n)
        {
            reinterpret_cast<T*>(&impl[size_TM * n])->~T();
        }
    }

    void push_back(const T& rhs)
    {
        new (&impl[size_TM * size]) T(rhs);
        ++size;
    }

    void push_back(T&& rhs) // rvalue reference, not universal reference
    {
        new (&impl[size_TM * size]) T(std::move(rhs));
        ++size;
    }

    template<typename... ARGS> void emplace_back(ARGS&&... args) // correct implementation
    {
        new (&impl[size_TM * size]) T(std::forward<ARGS>(args)...); // an important pattern in Yubo
        ++size;
    }

    static constexpr std::uint32_t size_TM = sizeof(T) + sizeof(M);
    std::array<char, size_TM * N> impl;
    std::uint32_t size = 0;
};

// For iterator, provide :
// one function to get the data
// one function to get the meta-data (such as atomic flag in lockfree_mpmcq)
```

Appendix 2 : Why `std::remove_cvref_t` is needed in universal reference?

Consider a self-made container offering a `lvalue` version and a `rvalue` version `invert` function :

```
template<typename T> class my_container
{
    iterator insert(const T& arg);
    iterator insert(T&& arg);
};
```

Now we merge two versions together using universal reference :

```
template<typename T> class my_container
{
    template<typename U>
    iterator insert(U&& arg) { impl(std::forward<U>(arg)); }
};
```

Now we apply constraint on the universal reference type `U`, so that `T` and `U` are the same type. However the following fails ...

```
template<typename T> class my_container
{
    template<typename U> requires std::same_as<T,U>
    iterator insert(U&&) { impl(std::forward<U>(arg)); }
};

container<my_class> c;
my_class object;
c.insert(object);             // U = const myclass&, U&& = const myclass& &&, compile error for  lvalue, no matched candidate
c.insert(std::move(object)); // U =       myclass,  U&& =       myclass  &&, compile ok    for  xvalue
c.insert(my_class{});        // U =       myclass,  U&& =       myclass  &&, compile ok    for prvalue
```

Thus we need to remove both `const` and reference before doing `std::same_as` comparison. The following works for all 3 insertions.

```
template<typename T> class my_container
{
    template<typename U> requires std::same_as<T,std::remove_cvref_t<U>>
    iterator insert(U&&) { impl(std::forward<U>(arg)); }
};
```