Class Design and Const Correctness

review of class design

- **interface**: specifies the operations that can be performed on instances of the class.
- **implementation**: specifying how those operations are to be performed.

class design terminology

public members are basically methods, vars are excluded in public members

- **public members:** methods that are part of the interface, can be called by a client (and inside the class).
- **private members**: helper methods and variables that can only be accessed inside the class.

const correctness

Recap

- For the most part, always anything that does not get modified should be marked const
- Pass by const reference is better than pass by value
  - Not true for primitives (`bool`, `int`, etc)
- Member functions should have both const and non const iterators
- Read right to left to understand pointers
- Please don’t make a method to blow up earth

Final Notes

const用在什么东西上面?
1. object（primitive type）
2. function
3. 特殊：const ptr和iterator

`const` on objects:

> Guarantees that the object won't change by allowing you to call only const functions and treating all public members as if they were const. This helps the programmer write safe code, and also gives the compiler more information to use to optimize.

`const` on functions:

> Guarantees that the function won't call anything but const functions, and won't modify any non-static, non-mutable members.

不会改变members

A Const Pointer

- Using pointers with const is a little tricky
  - When in doubt, read right to left

```
//constant pointer to a non-constant int
int * const p;

//non-constant pointer to a constant int
const int* p;
int const* p;

//constant pointer to a constant int
const int* const p;
int const* const p;
```

Operators

How does C++ know how to apply operators to user-defined classes?

```
vector<string> v{“Hello”, “World”};
cout << v[0];
v[1] += “!”;
```

C++ tries to call these functions.

note: there are 2 ways of calling operator in total;
the first one: declare operator as member function: cout.operator(...)
the second one: declare operator as non-member function: operator<<(lhs, rhs)...

```
vector<string> v{“Hello”, “World”};
cout.operator<<(v.operator[](0));
v.operator[](1).operator+=(“!”);
```

实际上cout这行是错的，因为<<operator不是以member function形式定义的，是一个non-member function，实际上是operator<<(v.operator[](0)); operator[]是一个v的member function，所以用.访问

Or these ones.

```
vector<string> v{“Hello”, “World”};
operator<<(cout, v.operator[](0));
operator+=(operator[](v, 1), “!”);
```

Key Takeaways

1. Respect the semantics of the operator. If it normally returns a reference to *this, make sure you do so!
2. When overloading operators as a member function, the left hand argument is the implicit *this.

chain together: a += (a += 3); cout << a << endl

Key Takeaways

1. The arithmetic operators return copies but doesn’t change the objects themselves. The compound ones do change the object.

General rule of thumb: member vs. non-member

1. Some operators must be implemented as members (eg. [], (), ->, =) due to C++ semantics.
2. Some must be implemented as non-members (eg. <<, if you are writing class for rhs, not lhs).
3. If unary operator (eg. ++), implement as member.

2. cout << vec; lhs: cout(stream); rhs:vec(vector<string>); writing class for rhs(vector<string>); << operator belongs to STD, not class vector<string>: cout.operator << (vec); the operator<< belongs to cout streams.
binary operator: write out the expression involving the operator first. consider lhs and rhs, the operator belongs to the lhs; l and r equality;

General rule of thumb: member vs. non-member

4. If binary operator and treats both operands equally (eg. both unchanged) implement as non-member (maybe friend). Examples: +, <.
5. If binary operator and not both equally (changes lhs), implement as member (allows easy access to lhs private members). Examples: +=

return value, do we need a reference or a value? when someone calls this function, does he need a value or reference
<< operator: chain together. change the original value or give a new value? e.g cout << … change the original cout stream

The subscript operator is one that must be implemented as a member.

```
string& vector<string>::operator[](size_t index) {
    return _elems[index];
}

const string& vector<string>::operator[](size_t index) const {
    return _elems[index];
}
```

变量型vecotr用前者；常量型vector用后者

Concept check

1. Why are we returning a reference?
2. Why are there two versions, one that is a const member, and one that is a non-const member?
3. Why are we not performing error-checking?

The client could call the subscript for both a const and non-const vector.

```
vector<string> v1{“Green”, “Black”, “Oo-long”};

const vector<string> v2{“16.9”, “fluid”, “ounces”};

v1[1] = 0; // calls non-const version, v1[1] is reference
int a = v2[1]; // calls const version, this works

v2[1] = 0; // does not work, v2[1] is const
```

if there is no const version, v2[1] = 0 works, but this is a const vector, nothing can be changed. Unexpected behavior occurs.

Declare non-member functions as friends of a class to give them access to private members.

在interface里面说一句就可以了

```
class Fraction {
public:
     Fraction(int hour, int minute);
     ~Fraction();
     // other methods

private:
     int hour, minute;
     friend ostream& operator<<(ostream& out, const Fraction& t);

}
```

friend

Principle of Least Astonishment (POLA)

C. 160

Design operators primarily to mimic conventional usage.

Principle of Least Astonishment (POLA)

C. 161

Use nonmember functions for symmetric operators.

Principle of Least Astonishment (POLA)

Always provide all out of a set of related operators.

配套

when designing operator

Summary of POLA

Notes for function design in a class:
1. const: return value; parameter, function, does it work for const object?
2. reference
3. private/public
4. POLA(operator)

Operator semantics are very important!

- Should this be a member or a non-member (friend or not?)
- Should the parameters be const or not?
- Should the function be const or not?
- Should the return value be a reference or a const reference?
- What is the convention for overloading that operator?

Special member functions are (usually) automatically generated by the compiler.

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.

create new;

- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.

Member Initialization List

const correctness; delare and assign simultaneously

Prefer to use member initialization list, which constructs each member with given value.

- Faster. Why construct, then reassign?
- Can’t reassign references, so must construct references directly with what they refer to.

```
template <typename T>
MyVector<T>::MyVector() :
                    _logicalSize(0), _allocatedSize(kInitialSize),
          _elems(new T[kInitialSize]) { }
```

std::initializer_list

```
template <typename T>
MyVector<T>::MyVector(std::initializer_list<T> init) :
        _logicalSize(init.size()),
         _allocatedSize(max(kDefaultSize, 2 * _logicalSize)),
        _elems(new T[_allocatedSize])
{
        std::copy(init.begin(), init.end(), _elems);
}
```

The copy operations must perform the following tasks.

assignment比copy多两个东西：1. 打扫干净屋子再请客；2. 防self– assignment

Copy Constructor

- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.

Copy Assignment

- Clean up any resources in the existing object about to be overwritten.
- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.

rule of three

When do you need to write your own special member functions?

When the default one generated by the compiler does not work.

Most common reason: ownership issues
A member is a handle on a resource outside of the class.
(eg: pointers, mutexes, filestreams.)

Rule of Three

If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three.

The fact that you defined one of these means one of your members has ownership issues that need to be resolved.

Rule of Zero

If the default operations work, then don’t define your own custom ones.

Move Semantics

CS 106L Winter 2020 – Avery Wang and Anna Zeng

Where we are going!

对lvalue只能copy；对rvalue尽量用move（偷）

- How do we distinguish between when we CAN and CANNOT move? = l vs. r-values

steal：偷：1. transfer拿过来；2. 把别人的扫荡干净，set other to defalut value

- How do we actually move? = implementation
- Can we force a move to occur? = std::move
- How do we apply move? = swap and insert
- Can we apply this to templates? = perfect forwarding

r-value reference（int&& a = rvalue）: 1. r:告诉是alias to an r-value

2. reference：可以pass by reference

r-value reference binds to an r-value, but itself is a l-value

a = b + 7编译器仍然会默认调用copy，一定要用std::move(b+7)强制转换，明摆着告诉编译器，这时候就会用move；但要注意当std::move(l-value)之后，里面的l-value不能够再使用。

Which ones cause compiler errors?

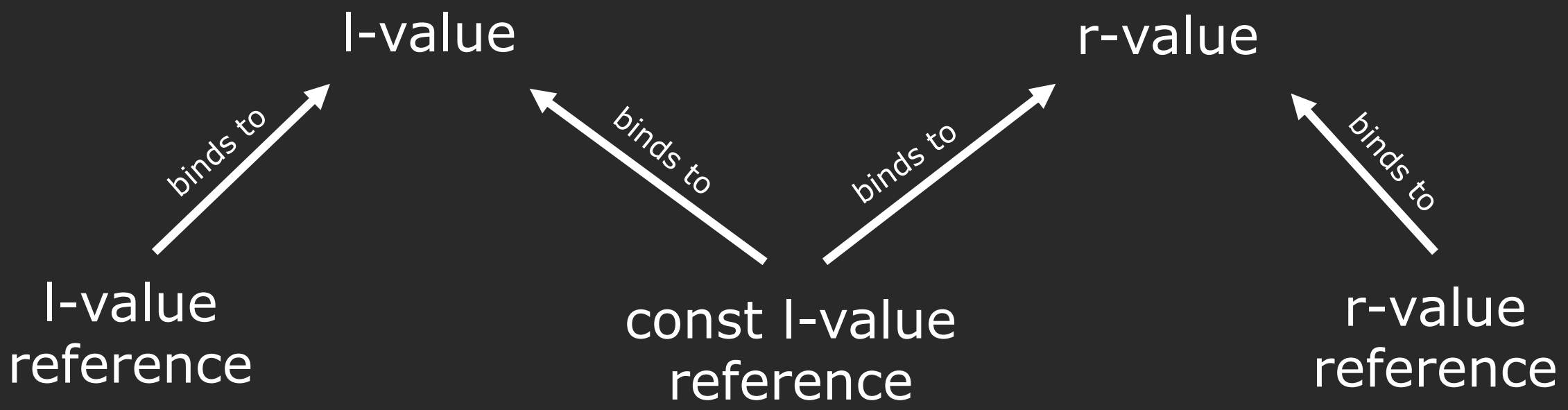
```
void lref(vector<int>& v);
void clref(const vector<int>& v);
void rref(vector<int>&& v);
// BTW: no one uses crref
```

r-reference作用：1. r部分：声明处理的对象是一个r-value（区分于l- value）；2. reference部分：可以用于pass by reference

```
vector<int> v1 = v2 + v3;
lref(v1);                   // l-ref binds to l-v
rref(v1);                   // r-ref no bind to l-v
lref(v2 + v3);              // l-ref no bind to r-v
clref(v2 + v3);             // cl-ref binds to r-v
rref(v2 + v3);              // r-ref binds to r-v
```

non-const l-reference
不能配r-value，要改r-value的值要明说，用r- value reference；但const l-reference因为不会改变r-value的值

Everything in one picture.



Key steps for a move constructor

一个字：偷steal：1. 先other的东西转移transfer过来；2. 再把other的东西扫荡干净（才算偷：我有了，你没有了）

- Transfer the contents of other to this.
  - Move instead of copy whenever possible!
- Leave other in an undetermined but valid state
  - Normally: set it to the default value of class

Rule of Five

If you explicitly define (or delete)
a copy constructor, copy assignment,
move constructor, move assignment, or destructor,
you should define (or delete) all five.

The fact that you defined one of these means
one of your members has ownership issues
that need to be resolved.

Rule of Zero

If the default operations work, then don’t define your own custom ones.

Rule of Zero

If the default operations work, then don’t define your own custom ones.

You are more likely to make a mistake if you define it when you don’t need to.

Design choices for iterators

The iterator should only support operators that it can perform in constant time!

(example: technically you could have + overloaded for list, but the runtime would not be constant!)

Step 1: Determine your private instance (member) variables.

```
MyVector<T>* _pointee;
size_type _index;
```

Point of concession: We want to minimize the dependencies our classes have with each other, and have the iterator store a pointer to the container is not ideal. For StringVector in fact, we really just need a pointer. For GapBuffer, we only need the pointee to a place in the array, index, gap location and size, and capacity. For a linked list in fact, just the pointee to the node! However, this makes implementing iterators very algorithmically challenging. In the interest of keeping things simple, we will break encapsulation and coupling rules.

Step 1: Determine your private instance (member) variables.

```
MyVector<T>* _pointee;
size_type _index;
```

Point of concession: We want to minimize the dependencies our classes have with each other, and have the iterator store a pointer to the container is not ideal. For StringVector in fact, we really just need a pointer. For GapBuffer, we only need the pointee to a place in the array, index, gap location and size, and capacity. For a linked list in fact, just the pointee to the node! However, this makes implementing iterators very algorithmically challenging. In the interest of keeping things simple, we will break encapsulation and coupling rules.

Step 2: Determine the public behaviors of your class.

- Our class will be a **nested class** (alternative: pimpl)
- Its official name is MyVector::iterator.
- The MyVector and iterator class are mutual friends.
- The iterator class has access to all the member types and template-y stuff.
- The iterator class inherits from an iterator tag.
- A ton of operators.

Step 3: How do we construct an instance of iterator?

- Let's create a constructor that initializes all members.
- We don't want the constructor to be accessible by anyone other than the class itself.

- Idea: declare the iterator constructor private.
- MyVector is a friend of iterator, so it can still call the private constructor.

Step 4: implement a ton of operators.

- ++, -- (pre+posfix)
- ==, !=, <, >, <=, >=
- * (deref)
- +=, -=
- +, - (iter+n, n+iter, iter-n, iter-iter)
- = (assignment – next time!)

Step 5: add iterator creators in MyVector

```
iterator begin() {
    return iterator(this, 0);
}

iterator end() {
    return iterator(this, _logical_size);
}
```

Step 2: Determine the public behaviors of your class.

- Our class will be a **nested class** (alternative: pimpl)
- Its official name is MyVector::iterator.
- The MyVector and iterator class are mutual friends.
- The iterator class has access to all the member types and template-y stuff.
- The iterator class inherits from an iterator tag.
- A ton of operators.

Step 3: How do we construct an instance of iterator?

- Let’s create a constructor that initializes all members.
- We don’t want the constructor to be accessible by anyone other than the class itself.

- Idea: declare the iterator constructor private.
- MyVector is a friend of iterator, so it can still call the private constructor.

Step 4: implement a ton of operators.

- ++, -- (pre+posfix)
- ==, !=, <, >, <=, >=
- * (deref)
- +=, -=
- +, - (iter+n, n+iter, iter-n, iter-iter)
- = (assignment – next time!)

Step 5: add iterator creators in MyVector

```
iterator begin() {
    return iterator(this, 0);
}

iterator end() {
    return iterator(this, _logical_size);
}
```