COMP 2012 Final Review

1. 4种 temp object 出现的形式

(1) const ref

```
int a = 1;          // a□ → a[1]      (named → named)
const int & a = 1;  // [1] → a□       (unnamed → named)
```

(2) argument passing

```
void func(const int& a);
func(1);   // similar to previous one
//or
void func(const INT& A);  // INT has a conversion constructor that accepts int
func(1);
```

(3) return by value

```
INT return_INT() { return INT(1); }
INT a = return_INT();
```

(4) evaluation of expressions (a = b+c+d;)

user-defined type

temp + d

copy temp

Caution:

```
INT a(1);
INT b(2);
INT c = (a+1)+2;
```

In this case, the unnamed guy "1", "(a+1)" and "(a+1)+2" will exist untill whole expression finishes (after c's copy constructor is called)

2. rvalue reference

```
int&& a;    // Error! rvalue reference must be initialized
int&& a=b;  // Error! rvalue reference can only bind to unnamed guy
```

```
rbv(int& b);
int&& a = rbv(b);  // Error! rvalue reference can only bind to unnamed guy
```

```
func(const int& a);
int&& a = 5;
func(a);   // No error! Can take almost every arguments
```

Best match:

have both { int&& parameter / const int& parameter / int& parameter }

- 传入 unnamed guy ⇒ call int&& one
- 传入 named guy ⇒ call int& one

rvalue reference v.s. const reference v.s. non-const reference

	int&&	const int&	int&
can bound to named guy?	✗	✓	✓
can bound to unnamed guy?	✓	✓	✗
can modify?	✓	✗	✓
needs to be initialized?	✓	✓	✓

Caution: const int&& can only be bound to rvalue, 跟 int&& 的区别只有 const

3. move semantics

适用条件: when we wish to get the value from unnamed temp object ① but it is to be killed ②

反例: ① if the object is global, it won't be killed after the function terminates ⇒ move constructor won't be called

②
```
INT func(){
    INT a;
    return a;   // → return by value, 虽然是named的, 但C++识别到该local variable将会GG, 于是implicitly改成 return move(a);
}
```
move(a) 可以 return 一个 a 的 unnamed 版本, 这个表述和 a 本身 address 一样, 对它作任何改动 a 也会变

应用场景: ① return by value (外面的东西 steal 到了, 当函数 terminate 的时候 kill unnamed guy)

② INT a=b; 的 implicit conversion

<more>

Caution: ① move constructor 不能 call move assignment operator

因为在 move constructor 里 parameter && 已经变成 named guy了

(可以用 move() 来重新变回 unnamed guy, 再 call move assignment operator)

② move constructor 不能 pass by const ref 因为 cut the link 的时候要 modify 它的

③
```
INT h() { INT a; return 0; }
//main
INT i = h();  // 两次 move, 因为 h() 是 return by value, 要先 move 到 temp object 再到 i
```
```
INT a;
a=b;  // 一次 move, 因为通常 move assignment operator 是 return by reference
```

④ int& ar = *(new int(3)); // 可以用 int& bind unnamed object 的特例

注意 copy constructor call operator= 的 caution, 即要先用 MIL initialize this→ptr 为 nullptr

operator= (正常 or move 的 const 可加可不加, 加了是防止 (a=b)=c

4. Hashing

hash function: h(k) = k mod m (m → size of hash table, should not be 2^n or 10^n)

interpret keys (strings, etc.) to int

- ① deterministic
- ② efficient
- ③ Uniform distribution
- ④ 能map到table里所有位置
- ⑤ 微小key扰动的结果很不一样

Dealing with strings:

① ASCII: 0+1+2 = 3 (a b c) (permutation share same value)

(example: abc) ② ASCII: $256^2 a + 256^1 b + 256^0 c$ (max: $26 \cdot (27^2 + 27^1 + 27^0)$, when m is large, 很多位置hash不到)

take first 3 letters as key ③ ASCII: $256^2 a + 256^1 b + 256^0 c + \cdots$

(2)是确保每位的位值大于前一位max值

3)可能是加上3数字)

Collision Handling

Home position: h(k)

① seperate chaining: one slot, multiple object

Insert: 放到 T[h(k)] → 空: initialize linked list; 不空: 放到 Linked list 第一个

Delete: 在 T[h(k)] 里找 key=k 的删

② Open addressing: reallocate the object!

Way to reallocate

(1) linear probing: $h_i(k) = (h(k) + i) \bmod m$

primary clustering: $T_{insert} = O(\frac{1}{2} \cdot size\_of\_cluster)$ Cluster easy to 滚雪球

(2) quadratic probing: $h_i(k) = (h(k) + i^2) \bmod m$

If table size is prime & half empty ⇒ can always be inserted

secondary clustering: home position 一样的会形成 cluster

(3) double hashing: $h_i(k) = (h(k) + i \cdot f(k)) \bmod m$

(f(k) is a function of key k)

一般来说, f(k) = R − (k mod R)

↳ R<m, R prime, m prime

Intuition: ① f(k) ≠ 0, or hash 变成原地踏步, that's why 要用 R 减

② (f(k), m) = 1, or 又能 hash 到 table 里都的位置 ⇒ make m prime!

让 R<m 是 ensure f(k) 不是 m 的倍数, m 又是 prime

⇒ (f(k), m) = 1

Insert: 找 T[h(k)]: { empty: insert! / active: reallocate! / deleted: insert! }

Delete: 找 T[h(k)]: { empty: stop searching, object not in hash table / active: delete & return / deleted: skip & search for next one }

Re-hashing

Load factor $\alpha = N/m$ ($^{active}/_{all}$)

α → 1, operation become very slow

⇒ rehash the whole table

by doubling table size and

hase every element to new

table using new table size

5. BST

Big O Notation: upper bound

$O(f(N)): \leq \alpha \cdot f(N) + \beta$

Trees: O(log N) insert, search, delete

siblings: 同 parent 的 child

length: number of edges on path

depth of node: length of unique path from root to that node

height of node: max{len(node, leaf)} (all leaves)

height of a tree: height of the root

ancestor & decendant: ancestor & decendant 包括自己, 但 proper ancestor & proper decendant 不包括

Traversal { InOrder / PreOrder / PostOrder } → 给任意一种就能推断出树的结构

不是任意的 int 序列都能成为上述 traversal 的序列

Stack/Queue implementation

① InOrder

pseudo:

```
if root == nullptr, return
cur = root
stack
while(cur != nullptr or stack not empty)
    while(cur != nullptr)
        stack.push(cur)        // 假如 cur 不是 nullptr
        cur = cur.left         // 一直往左走, 把路上的 node 压进 stack 里
    cur = stack.top            // 假如 cur 是 nullptr, 那就 start 处理 stack 顶的 element
    print cur
    stack.pop
    cur = cur.right
```

② PreOrder

pseudo:

```
if root==nullptr, return
stack.push(root)
while stack not empty:
    print top
    pop
    push(top.right)
    push(top.left)
```

③ PostOrder

pseudo:

把 Pre 改成先 push left 再 push right

搞完 reverse

Search & Insert: { search: 找到了v 没找到依据值继续找
insert: 到leave了就插入 没到就依据值做recursion

Delete:

pseudo:

```
if this is null, return
if value < this value
    left.remove
else if value > this value
    right.remove
else if this.left is null & this.right is null
    this value = this right.findmin();
    this right.remove(this value);
else
    temp = this→root
    this→root = (this left != null?) this left : this right
    temp→root→left = null
    temp→root→right = null
    delete temp
```

6. Generic programming

function template

① 用 template <> 来做模版 specialization

template <typename T, int N>
void size(T (&arr)[N]) {...}

② template <typename T, int N> ----------→ 应用在把array传进function里
↳ cannot be modified, const

③ 假如如不实例化 → template, compiler 不会check syntax
又有实例化了比如<int>类, compiler 就会在 int setting 下 check 整个 function/class 的 syntax error

④ 使用多种type的formal parameter的问题

```
template <typename T1, typename T2>
T1& larger(const T1& a, const T2& b)
```

solution: { return by value (type conversion)
don't turn, print

⑤ { function template: compiler 会 deduce template arguments
class template: 必须手动 specify

7. Operator overloading

① { nickname: + only can be used when calling the function
formal name: operator+() can be used anywhere

② { cannot be overloaded: . :: ?: .*
only member function: = [] ()
only global function: << >>

不能被变 { the number of parameters
Associativity
Precedence

也不能define新的

假如同时 overload了 global & member function

它优先用member版本

如果 { member: Vector operator+(const Vector& v) const → 重载掉这个
global: Vector operator+(const Vector& v1, const Vector& v2) ⇒ Ambiguous!

③ operator<< return type 是 ostream&

不能加 const! cout 的时候在一直变

& 是因为这个特殊 object 不会被 kill

不能是 void, 要不然就不能 连续 cout 了

④

	global	member
V2 = V0+V1	✓	✓
V2 = V0+1	✓	✓
V2 = 1+V0	✓	X
V2 = 1+2	✓	✓

↳ 先算出1+2 再 construct object

⑤ member function: +=, = return type 是 const ref

const 是因为 防止 (a+=b)+=c

ref 是 enable a+=b+=c 功能

⑥ operator = (member)

1. check self-assignment
2. do assignment

Caution: 如果 copy constructor 要 call operator=
要先把 pointer 设为 nullptr
来防止 delete 垃圾值的情况出现
str{nullptr}

⑦ operator []

两种: (1) read-only
double operator[](int id) const const object 才能 call
(2) read & write
double& operator[](int id) non-const object 才能 call

⑧ operator ++

两个版本

(1) read-only
double& operator++() 加完 return *this 即可

(2) read & write
double operator++(int)
↳ compiler 用这个 int 标志来识别 post-increment
↳ 要用temp存着 original value, 然后加, 再return temp

8. friend

在 class scope 里任意位置 declare friend class/function

properties: 无相互性, 传递性, 遗传性

9. Lambda expression

```
int hehe = 10;
auto gaga = [hehe] (int a, int b) { return hehe; };
// 不能grab global variable
```

```
int result = [] (int a, int b) { return (a > b)? a: b; } (10, 20);
cout << result <<endl;
// 不用specify return type, c++会automatically帮我decide
auto haha = [] (int a, int b) { return (a > b)? a: b; }; // 用一个function pointer来接住lambda expression
// 让它下次可以直接call haha()来用
cout << haha(10, 20) << endl;

int a = 1, b = 2, c = 3, x = 4, sum = 0;
for(int k = 0; k < 4; ++k)
{
    // 这里的[=]意思是我要grab所有define过的local variable, =的意思是by value
    cout << [=] (int x) { return a * x * x + b * x + c; } (k) <<endl;
    // expression a * x * x + b * x + c 中的x指的是parameter x而不是local variable x
    // 因为x是parameter, considered更近
}
for(int k = 0; k < 4; ++k)
{
    // 这里的[&]的意思是grab by reference
    cout << [&] (int x) { return a * x * x + b * x + c; } (k) <<endl;
}
for(int k = 0; k < 4; ++k)
{
    // 这里的[&sum]的意思是只grab sum这个variable, 而且是by reference
    cout << [&sum] (int x) { return a * x * x + b * x + c; } (k) <<endl;
    // 这里的[a]的意思是只grab sum这个variable, 而且是by value
    cout << [a] (int x) { return a += x; } (k) <<endl;
    // 上面这样的是不行的, 因为a虽然是grab by value, 但是其实是不可以改的
    // 如果一定要改的话要加mutable keyword
    // 但是就算这样你在函数里改了也不能真的改它的值, 因为是by value
    cout << [a] (int x) mutable { return a += x; } (k) <<endl;
}
a = 10, b = 11, c = 12;
for(int k = 0; k < 4; ++k)
{
    // 这里的a, b, c的值还是1, 2, 3, 因为是grab by value的, 所以跟什么时候grab的有关系
    // grab完之后再也不变了
    cout << [=] (int x) { return a * x * x + b * x + c; } (k) <<endl;
    // 但这里的a, b, c就变成10, 11, 12了, 因为是grab by reference
    cout << [&] (int x) { return a * x * x + b * x + c; } (k) <<endl;
    // 下面这样写是不可以的
    cout << [const&] (int x) { return a * x * x + b * x + c; } (k) <<endl;
}
```

这样大概是看不出效果的 得用个function pointer 指着这个lambda expression, 然后改abc的值才ok

```
for(int k = 0; k < 4; ++k)
{
    // 如果想call好几个的话可以这样
    cout << [a, &b, &c] (int x) { return b += x; } (k) <<endl;
    // 也可以这样
    cout << [&, a] (int x) { return b += x; } (k) <<endl;
    // 不能反过来写
    cout << [a, &] (int x) { return b += x; } (k) <<endl;
}
// 假如不想让c++帮我决定return type, 要自定义
// 就在()和{}之间写个->然后写想return的type
auto tata = [] (int a, int b) ->int { return (a > b)? a: b; };
```

function pointer

```
// function pointer
int(*haha) (int, int) = larger;
int result = haha(10, 20);
// or
int(*haha) (int, int) = &larger;
int result = (*haha)(10, 20);
// 对于lambda expression来说
// 可以用function pointer来指着
// 但是当capture list里面有东西的时候不可以这样
int (*hehe) (int, int) = [] (int a, int b) ->int { return (a > b)? a: b; };
```

Final Past papers Error-prone

separate chaining
最后密写 nullptr

destructor 可 dynamic binding
不可 inherit

在 push_back 的时候 注意本来是什么 type 的
假如有 inheritance hierachy 务必 cast 成
对应 type

小心 BST T/F 题！pre 的第一个不一定是 minimum

在 class scope 里不用写 <T>
在 tpp 的 { } 里不用写 <T>，在 { } 外面
(如传入 parameter 或 namespace 符号)
就要写 <T>

没问题的！
注意如 template 就不能 type conversion

原理

operator 还是 override，因为
return type 不一样

operator<< 的 type T1, T2
不能跟 class 的 T1, T2 名字一样

有 dynamic 东西
的 debug 注意
double free

注意 return type

print 函数注意！
print 之前 check nullptr！

别忘了 return 该 return 的东西

倒序 destruct！

const object call 了
non-const func

operator 记得 self-check!!!

注意 global variable：在 main 开头
的时候就 construct，main 最后
destructor，假如 return 它 by
value 就是 copy 一个 temp guy
出去，里面的不删！

传了两个 A ... 注意看 type 也只能这

原来 char[a] 和 char [b]
type mismatch

global function 该 declare friend 的可 declare

注意 return type

no! 可以 access！因为传进来是
Pocket Monster & type
答案反而不对，因为不知道 enemy
是 exactly 哪个 type