9 Big Three

1. 当我们 call 一个 ctor/dtor 时，也会自动 call 它 base class 的 ctor/dtor. 注意 ctor 是先 call base class 再 call derived class，dtor 是先 call derived class 再 call base class.
2. 当函数 pass by value 时，如果有一个参数是一个 class obj，那么我们相当于在这个函数的 stack frame 中使用 copy ctor 初始化了 local variable. **如果我们 `SomeClass a = b` 来初始化一个 class obj，call 的并不是 operator= 而是 copy ctor**

自定义 assignment operator

```
template <typename T>
ordered_set<T> & ordered_set<T>::operator=(const ordered_set<T> & rhs) {
  if (this != &rhs) {   // 1. Check for self-assignment
    delete[] elts;  // 2. delete old dynamic array
    elts = new T[rhs.current_capacity]; // 3. make a new dynamic array
    current_capacity = rhs.current_capacity;
    for (int i = 0; i < size; i++) {ele[i] = rhs.ele[i];}
    //... 4. copy regular attributes
    return *this; // 5. return *this
  }
}
```

The Big Three 是指：dtor, copy ctor, operator=；如果要使用 dynamic memory，就要 deep copy；The rule of the Big Three: 如果你需要为 any of them 提供一个 custom implementation，那么你就必须要为 all of them 提供 custom implementations.

10 Linked List and Iterator

```
template <typename T>
class List {
public:
      class Iterator {
      friend class List;
      public:
        Iterator();
        T & operator*() const;       // overload *
        Iterator & operator++();     // overload ++
        bool operator==(Iterator rhs) const; //...
      private:
        Node *node_ptr;
        Iterator(Node *np);
      };
Iterator begin(); // at beginning
Iterator end(); // "past the end"
  //...
};
template <typename T>
List<T> & List<T>::operator=(const List<T> &rhs) {
  if (this == &rhs) return *this;
  pop_all();       // free resources
  push_all(rhs);  // deep copy
  return *this;
}
template <typename T>
T & List<T>::Iterator::operator*() const {
  return node_ptr->datum;     // 返回 Iterator 的 node_ptr 指向的 Node 值
}
template <typename T>
typename List<T>::Iterator & List<T>::Iterator::operator++() {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
template <typename T>
typename List<T>::Iterator List<T>::Iterator::operator++(int) {
  assert(node_ptr);
  Node *copy = node_ptr;
  node_ptr = node_ptr->next;
  return *copy;
}

template <typename T>
void func() {
  List<int>::Iterator it2; // does not depend on T
  typename List<T>::Iterator it3; // depend on T
}
```

11 Functors

1. function ptr:

注意 1：c++ declaration 是 inside out 的。[], () 比起 prefix operators 如 &, * 等有更高的优先级；注意 2: 我们给 function pointer 赋值不 dereference，只有把需要指向的 function object 的 address 赋给 ptr 这种方法；我们也可以直接写 function pointer = 某个 function，compiler 会自动推测意思，补全为把该 function 的 address 赋给这个 function pointer。

```
int (*fn)(int, int) = nullptr;
*fn = std::max;   // error, 不支持这种写法
int (*fn2)(int, int) = nullptr;
fn2 = &std::max;  // ok, 把 std::max 的 address 赋给 p
int (*fn3)(int, int) = nullptr;
fn3 = std::max;   // ok, 这里 compiler 自动理解意思, 把 std::max 的 address 赋给 p
```

2. functor

```
class GreaterN {
private:
  int threshold;
public:
  GreaterN(int threshold_in): threshold(threshold_in) {}
  bool operator()(int n) const {return n > threshold;}
}
int main() {
  GreaterN g212(212);
  cout << g212(189);    // false
}
```

3. 使用 template class 作为 Predicate 参数以适用任何 Functor

```
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end, Predicate pred) {
      for (Iter_type it = bein; it != end; ++it) {
    if (pred(*it)) {return true;}
  }
  return false;
}
```

12 Recursion

Linked List:

```
int sum(Node* n) {
  if (n == nullptr) return 0;
  else return n->datum + sum(n->next);
}
int last(Node *n) {
      if (n->next == nullptr) return n->datum;
  else return last(n->next);
}
int count(Node* n, int val) {
      return (n->datum==val) + count(n->next, val);
}
int max(Node* n) {
      if (n->next == nullptr) return n->datum;
  else {
    int maxInRest = max(n->next);
    if (n->datum > maxInRest) return n->datum;
    else return maxInRest;
```

Tree:

```
bool contains(Node *r, int val) {
  if (r == nullptr) return false;
  else return (r->datum == val) || contains(r->left, val) || contains(r->right, val);
}
int numLeaves(Node* r) {
  if (r == nullptr) return 0;
  else if ((r->left == nullptr) && (r->right = nullptr)) return 1;
  else return numLeaveds(r->left) + numLeaves(r->right);
}
int height(Node *r) {
  if (r == nullptr) return 0;
  else return 1 + max(height(r->left), height(r->right))
}
int sum(Node* r) {
  if (r==nullptr) return 0;
  else return r->datum + sum(r->left) + sum(r->right)
}
void inOrder(Node* root) {
  if (root != nullptr) {
    inOrder(root->left);      // pre-order: then put process here
    // inorder here: process root
    inOrder(root->right); // post-order: then put process here
  }
}
```

13 BST and map

```
  static Node * find_impl(Node *node, const T &query, Compare less) {
if(!node)
  return nullptr;
else if(less(query, node->datum))
  return find_impl(node->left, query, less);
else if(less(node->datum, query))
  return find_impl(node->right, query, less);
else
  return node;
  }
static Node * insert_impl(Node *node, const T &item, Compare less) {
if(!node)
  return new Node(item, nullptr, nullptr);
else if(less(item, node->datum))
  node->left = insert_impl(node->left, item, less);
else
  node->right = insert_impl(node->right, item, less);
return node;
}

  static Node * min_greater_than_impl(Node *node,
                                      const T &val, Compare less) {
if (!node)
  return node;
else if (less(val, node->datum)) {
  Node* temp = min_greater_than_impl(node->left, val, less);
  if (temp) return temp;
  else return node;
}
else return min_greater_than_impl(node->right, val, less);
}
```

2. const keyword

1. **const pointer**: 形如 `int * const ptr = &x;`，值在 initialization 之后就不能变了. 也就是**它装的 address 不能变**，但是它装的 address 下的变量是可以变的，也就是可以解引用它来更改它指向的 object 的值

2. **pointer to const**: 形如 `const int * ptr = &x;` 或 `int const * ptr = &x`，值可以变，也就是它可以重新指向别的地址，但是这个指向的行为是 `const` 的

   x 不是 `const` 变量，还是可以改变 x 的值，但是**不能 dereference `ptr` 来改变 x 的值.**

3. **如果要指向一个 const variable，那么必须使用 ptr-to-const**

4. 不允许把一个 pointer-to-const 或者 const pointer 的值传给一个 普通的 pointer。（把 const int 的值传给一个普通 int 是可以的）

5. ```
   void func1(const int *ptr) {
       // ptr-to-const, 不能改变指向对象值
   }
   void func2(int * const ptr) {
       // const ptr, 指向的地址不能改变, 可以改变指向对象值
   }
   /* 1. 将普通指针传给要求 pointer-to-const 的函数,
   函数内部无法通过这个指针修改数据. 但是原指针可以; 2. 将普通指针传递给要求
   const pointer 的函数, 函数内部不能改变指针的值, 但是原指针可以; 3. 把 const
   variable 传给参数不是 const 的函数: error */
   ```

6. reference to const: 形如 `int const &ref = x;`，不能用它来 change object 值，但仍可以用这个 object 的非 `const` variable 改 object 值.

7. `const int func()` return 一个 `const`，而 member function `int func() const` 表示这个 member function 不能改变 member 的值.

5. class (C++-style-ADT) and derived class

1. `this` 不仅可以 `->` member variable 还可以 `->` member function. `this -> scale()` 就是 `*(this).scale()`. 可以不写 `this` 让 compiler 自动默认。

2. const 的 object 不能使用非 const 的 member function

3. **只要是同一个 class，那么这个 class 其他 object 的 member 也可以由这个 object 的 member function access.**

4. 一旦自己写了其他 ctor，compiler 自动添加的 default ctors 就失效

5. 一旦自己写了其他 ctor，compiler 自动添加的 default ctors 就失效; 如果不使用 initializer list，当 class 的成员中包括了另一个 class，compiler 就会自动调用这个另一个 class 的 default constructor，而它有可能已经没有 default constructor 了，那么就会 error. 使用 initializer list 可以避免这个错误.

6. derived class: 形如 `class Bicycle: public Vehicle {};` Derived class 自动继承 base class 的所有 member variables 以及 functions，但 **Derived Class 无法 assess Base Class 中的 Private members**. 必须通过 base class 的 public function 来 access. `protected` variables 表示子类 access 的 private member.

7. Container ADT

3. binary search: $O(log\ n)$

```
int binarySearch(int arr[], int size, int target) {
  int left = 0;
  int right = size - 1;
  while (left <= right) {
    int mid = (left + right) / 2; // binary split!
    if (arr[left] == target) {return mid;}  // found!
    if (arr[mid] < target) {left = mid + 1;}  // not found: move side
    else {right = mid - 1;}
  }
  return -1; // if not found: not in the array
}
```

其他 operator overload

```
bool operator==(const Person &p1, const Person &p2) {
  return p1.name == p2.name;
}
```

```
template <typename T>
ordered_set<T> & ordered_set<T>::operator=(const ordered_set<T> & rhs)
{
  if (this != &rhs) {   // 判断是否是给自己赋值, 这非常重要!
    delete[] elts;  // delete old dynamic array this is pointing to.
    elts = new T[rhs.current_capacity]; // make a new dynamic array

    // copy all attributes
    current_capacity = rhs.current_capacity;
    size = rhs.size;
    for (int i = 0; i < size; i++) {
      ele[i] = rhs.ele[i];
    }
    return *this;
  }
}
```

	UnsortedSet	SortedSet	BSTSet
insert	O(n) contains? O(n) + insert at end O(1), maybe grow O(n)	O(n) contains? O(log n) + insert/shift right O(n), maybe grow O(n)	O(log n)
remove	O(n) find O(n) + replace with last O(1)	O(n) find O(log n) + remove/shift left O(n)	O(log n)
contains	O(n) linear search	O(log n) binary search	O(log n)
size	O(1)	O(1)	O(1) Must track this with separate variable

4. Stream 和 C-string

1. ```
   char str1[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
   char str2[6] = "hello"; // 等价
   const char *ptr = "hello";    // 等价, 但注意必须要 const, 因为 array 的首元素地址是 const 的
   ```

   如果从 "" 形式 initialize， `\0` 会被自动加上，但**如果从 array 形式 initialize，我们必须在最后一个元素后手动加上 '\0'**

2. cout 由 char array 名字 decay 形成的 char* 会 cout 整个 char array 而不是首元素地址, 这是一个 standard overload.

3. 不要 out 并非指向一个 C string 的 char* 型变量. 它也会由于 overload 一直输出到找到 null 为止

4. `x += 1` 和 `++x` 是等价，先 +1 再 evaluate expression；和 `x++` 不等价的.

5. 把string 转化成 int/double：`stoi()` 和 `stod()` for c++-string 以及 `atoi()`, `atof()` for c-string.

6. main 格式: `int main(int argc, char *argv[]) {}`. `cout << argv[1]`的时候，`argv[1]` 是一个地址，但是我们 cout 的是一整个 argument, 由于 c-stirng 的 cout overload.

7. 把 C++ string 转成 C string `const char *cstr = str.c_str();`，把 C string 转成 C++ string: `string str = string(cstr);`

8. strcpy 的作用是把第二个 cstr 的值传给第一个.

   ```
   void strcpy(char *dst, const char *src) {
         while (*dst++ = *src++) {}
   }
   ```

9. Concatenate: `strcat(cstr1, cstr2)` for c-string 以及 `str1 += str2` for c++-stirng.

10. compare: `strcmp(A,B)` return 第一个不一样的字母的 ascii 码的大小差距. (A 比 B 小就返回负值)

11. 比较 c-string:

    ```
    if (argv[1] == "--debug") { ... }
    // BAD, compares addresses
    if (argv[1] == argv[2]) { ... }
    // BAD, compares addresses
    if (std::string(argv[1]) == "--debug") { ... }
    // OK, wrap one in a std::string
    if (argv[1] == "--debug"s) { ... }
    // OK, ""s suffix creates a std::string
    ```

6. Polymorphism

1. **只有 1. pointer 类型以及 2. 作为引用的 variables 才会有 dynamic type！！**

   ```
   void fp(Vehicle *vp) {cout << vp -> get_insurance_amount() << endl;}
   //vp 的 static type 是 Vehicle *, 因为我们这样 declare
   int main() {Car mc; fp(&mc);}  //这里 parameter vp 的 dynamic type 是 Car *, 因为 compile 时传进去的类型是个 Car *
   ```

2. **一旦 base class 中一个 member function 是 `virtual` 的，那么它的 derived class 中所有这个函数的 override 函数自动变成 `virtual` 的**

3. 一个 **pure virtual function** 指的是一个以 `=0` 为结尾 declare 的 function。表示这个 function 是没有定义的. 一个 **abstract class** 指包含至少一个 pure virtual function 的 class，不能 instantiate，但如果有 data member，则仍然需要一个 constructor；**pure abstract class**，指**所有函数都是 abstract functions，并且没有 data members 的 class**，不需要写 constructor；且 compiler 会自动为 pure abstract class 提供 default constructor。

4. ```
   Bicycle b;
   Vehicle* vp = &b; //upcasting
   Bicycle* bp = vp; //不能这样 donwcast, error
   Bicycle* bp = dynamic_cast<Bicycle*>(vp); //downcast
   ```

8 Managing Dynamic Memory

Memory leaks/errors

1. **memory leak**: object 无法 delete
2. **orphaned memory**: 失去 address 的 hold 使得 leak 不可避免
3. **double delete**: 尝试 `delete` 同一个 heap object 多次
4. **Non-heap delete**: 对一个指向 non-heap object 的 ptr 使用 `delete`
5. **wrong delete**: 错用 `delete` 和 `delete[]`
6. **use a dead object**: dereference 了一个已经 dead 的 heap object.

dynamic array 的 grow:

```
template <typename T>
void ordered_set<T>::grow() {
  // new a dynamic array into a temporary ptr var
  int *tmp = new int[current_capacity + 1];
  // copy the original dynamic array into the new one
  for (int i = 0; i < size; i++) {
    tmp[i] = ele[i];
  }

  delete[] elts;  // delete the original dynamic array
  elts = tmp; // reassign the value of tmp to elts
  current_capacity++; // update capacity
}
```

```
template <typename T>
class ordered_set {
private:
  static const int DEFAULT_CAPACITY = 100; // for default ctor
  T *ele;   // dynamic array
  int size;
  int current_capacity; // dynamic capacity
public:
      ordered_set();
  ordered_set(int capacity);
}
```
```