20 Deep Copies and the Big Three
- 20.1 Copy Constructor
  - 20.1.1 如何 declare 一个 copy ctor
  - 20.1.2 default copy ctor 用以复制 dynamic set: shallow copy, 导致各种 memory error
  - 20.1.3 写一个 custom version of copy ctor 以实现 deep copy
- 20.2 Assignment operator `=`
  - 20.2.1 两种不同的 Copy: 使用 copy ctor 的 initialization 和 使用 operator `=` 的 assignment
  - 20.2.2 class default overloaded `operator=`
  - 20.2.3 写一个 deep copy 的 customed overloaded `operator=` for a class
- 20.3 The Rule of The Big Three
  - 20.3.1 the Big Three 的特点:
  - 20.3.2 什么时候需要 custom Big Three
  - 20.3.3 Custom the Big Three 的注意事项

20 Deep Copies and the Big Three

复习:

```
class A {
public:
  A() {
    std::cout << "A ctor" << std::endl;
    pa = new int(5);
  }
  // non-virtual dtor: static binding!!!
  ~A() {
    std::cout << "A dtor" << std::endl;
    delete pa;
  }
private:
  int* pa;
};
```

```
class D : public A {
public:
  D() {
    std::cout << "D ctor" << std::endl;
    pd = new int(6);
  }
  ~D() {
    std::cout << "D dtor" << std::endl;
    delete pd;
  }
private:
```

```
  int* pd;
};
```

```
int main() {
  A *x = new D();
  // 1. 在 heap 中 new 了一个对象, 类型是 D.
  // 2. 生成了一个指针 x: static type 为 A *, 而 dynamic type: D *
  // 3. 将 x 指向这个对象
  std::cout << "In main" << std::endl;
  delete x;
}

/*
result:
A ctor
D ctor
In main
A dtor
*.
```

principle(1)：由于 D 是 A 的 subtype，**当我们 call 一个 D 类型的 constructor 时，compiler 也会先自动 call A 类型的 constructor，然后再 call 这个 D 的 constructor.**

principle(2)：同样，**当我们 call 一个 D 类型的 destructor 时，compiler 也会在之后自动 call A 类型的 destructor.** 注意 **ctor 是先 call base class 再 call derived class，dtor 是先 call derived class 再 call base class！！**

所以，这个地方由于我们在 heap 中 new 了一个 D 类型的 object，我们先后自动 call 了 A 和 D 的 default constructor.

```
A ctor
D ctor
In main
```

同时，我们在 main 的 stack frame 中生成了一个指针 x: static type 为 A *, 而 dynamic type: D *.

我们注意到，`~A` **并不是一个 virtual function，因而是 static binding 的！** 所以当 `main()` 运行结束后，`delete` 只会 call `x` 的 static type 也就是 `A *` 对应的 `A` 的 dtor，而不会 call `D` 的 dtor.

```
A dtor
```

但是如果我们把 `~A` 改成 virtual function，那么它获取到的就是 `x` 的 dynamic type. 因而根据 principle (2)，我们就会自动在 call `A` 的 destructor 之后也 call `D` 的 destructor.

```
  virtual ~A() {
    std::cout << "A dtor" << std::endl;
    delete pa;
  }
/*
result:
A ctor
D ctor
In main
D dtor
A dtor
*.
```

20.1 Copy Constructor

Copy ctor，顾名思义，就是从现有的 object 复制出另一个 object 的 ctor.

形式如:

```
Example::Example(const Example &someotherExs)
  : /*member initializer list...*/ {
  // body...
}
```

我们即便不写 copy constructor，c++ 也会自动提供一个 implicit copy ctor:

```
Cowboy(const Cowboy& other)
  : name(other.name) /*contains all member of other*/ {}
```

实际上，当我们在给函数传递参数，并且是 pass by value 的时候，如果有一个参数是一个 class object，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，因而是 implicitly call 了 copy ctor.

但是：copy ctor 中一定是 pass by reference 的。

用法如：

```
int main() {
  Cowboy c2("Frankie");
  Cowboy c3(c2);  // call copy ctor
  byVal(c2);  //implicitly call copy ctor
  byRef(c2);  //None ctor called(since ref)
}
```

20.1.1 如何 declare 一个 copy ctor

```
class Example {
public:
  Example(const Example &other);  //all right!
}
```

记得 ctor 中我们要 copy 的对象一定是 pass by reference 的.

当我们**在别的地方 call copy ctor 的时候，我们是 pass by value 的**，因为 **pass 一个 class object by value 就触发了 copy ctor**，这就是 implicitly call copy ctor 的方式，因为 pass class object by value 就是 call copy ctor。

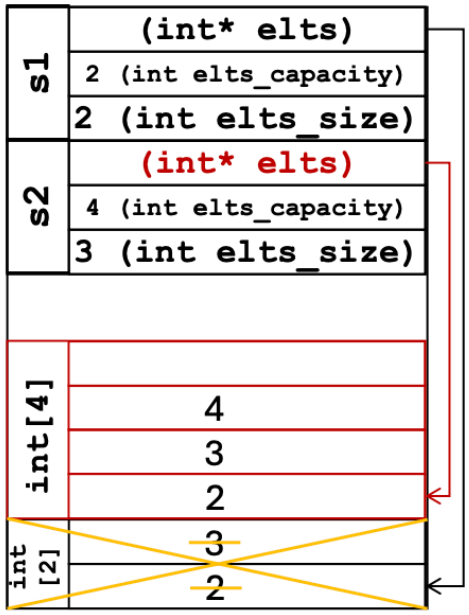
正因此我们**写 copy ctor 就必须把要 copy 的 object pass by reference**，否则这就成为了一个 inifite recursion: copy ctor call copy ctor，recurse inifinitely.

```
class Example {
public:
  Example(const Example other); //error, recurse inifinitely
}
```

20.1.2 default copy ctor 用以复制 dynamic set: shallow copy, 导致各种 memory error

```
template <typename T>
class Unordered_set {
private:
  T *elts;
  int elts_capacity;
  int elts_size;
public:
  //...
}

// implicit copy ctor: 会copy: elts(other.elts);

int main() {
  Unordered_set<int> s1(2);
  s1.insert(2);
  s1.insert(3);

  Unordered_set<int> s2 = s1; // 现在 s1.elts 和 s2.elts 指向同一个 dynamic array!

  s2.insert(4);
  // s2 和 s1 指向的 dynamic array 因为 capacity 满了 并且 s2 又要 insert 所以而 call grow()了. 这意味着原本的 dynamic array 被删除了, 但是因为 s2 call 的 insert, 只有 s2 会获得 new 的 dynamic array. s1.elts 已经 dead 了
```

```
  std::cout << s1 << std::endl; // s1 指向的 dynamic array 已经被 s2 因为 grow() 删除了, 所以这里是一个 undefined behavior, 尝试 cout 一个 dead object.
}
```



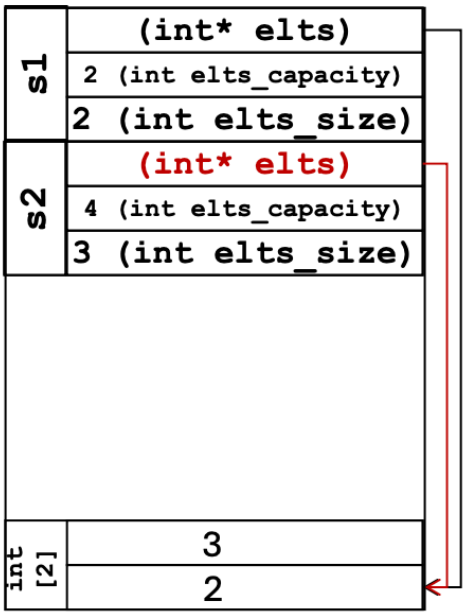
我们通过 implicit copy ctor，`s2 = s1` copy 的是 `s1` 的每个 member，因而 `s1.elts` 和 `s2.elts` 指向的是 heap 中的同一个 dynamic array.

因而任何对 `s1` 的变化都会同时影响 `s2`.

这种影响会带来不可避免的错误，比如上面的代码：使用了一个 dead object.

再比如:

```
int main() {
  Unordered_set<int> s1(2);
  s1.insert(2);
  s1.insert(3);
  Unordered_set<int> s2 = s1;
}
// 在 program 结束后 error, 因为这个时候 s1 和 s2 都自动 call dtor, 但是它们的 elts 指向的 dynamic error 是同一个, 因而 double delete 了.
```



20.1.3 写一个 custom version of copy ctor 以实现 deep copy

```
template <typename T>
ordered_set<T>::ordered_set(const ordered_set &other)
  : elts(new T[other.current_capacity]),
    size(other.size),
    current_capacity(other.current_capacity) {
  for (int i = 0; i < elts_size; ++i) {
    elts[i] = other.elts[i];
  }
}
```

deep copy: 创建一个 following pointer，也就是 `new` 一个新的 dynamic array 并把原来的 dynamic array 的元素全部 copy 过来，而不是 copy 原来的 dynamic array 的指针。这样就阻止了我们在 20.1.2 中的这些诸多 memory errors 以及 undefined behaviors.

20.2 Assignment operator `=`

20.2.1 两种不同的 Copy: 使用 copy ctor 的 initialization 和 使用 operator `=` 的 assignment

c++ 中 class object 的 `=` 在 initializaiton 和 assignment 的 context 下使用的函数是不同的。

```
int main() {
  Example a1(2, 3.5);
  Example a2(a1); // copy ctor
  Example a3 = a1; // init a3 as copy of a1: copy ctor
  func(a1); // init parameter a as copy of a1: copy ctor
}
```

第一次 initalize 一个 class object 等于另一个 object：使用的是该 class 的 copy ctor 函数. （包括 function 的 pass by value 也是相同的.）

```
int main() {
  Example a1(2, 3.5);
  Example a2(4, 6.7);
  a1 = a2;  // assign 而不是 initialize: operator=
}
```

并非第一次 initialize，而是重新给一个 class object assign 另一个 object 的值：使用的是该 class 的 overloaded `operator=`.

我们即便不写一个 class 的 overloaded `operator=`，这个 class 也会自动获得一个 default overloaded `operator=`，就像是自动获得一个 default ctor 和 default copy ctor 一样。

20.2.2 class default overloaded `operator=`

一个 class 自动获得的 defalut overloaded `operator=` 是长这样的：

```
Example & Example::operator=(const Example &rhs) {
  x = rhs.x;
  y = rhs.y;
  return *this;
}
```

这个 **defalut overloaded `operator=` 的作用实际上和 default copy ctor 做的事情基本一样：都是 member by member copy，也就是 shallow copy. 因而同样会出现相同的 memory errors/undefined behaviors！**

```
a = b;
// 也就是 a.operator=(b), this 是一个指向 a 的 ptr, 返回 *this, 也就是 lhs object.
```

这里值得一提的是，其实 `operator=` 完全可以是个 `void` 的函数，但是 implementation 却是返回原 class object 的引用，这种设计是为了允许链式赋值。

```
a = b = c;
```

冷门知识之这是可行的。

20.2.3 写一个 deep copy 的 customed overloaded `operator=` for a class

```
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;
  }
}
```

和写一个 deep copy customed copy ctor 基本一样，逻辑就是 `new` 一个 dynamic array 并把 rhs 这个 object 的 dynamic array 的元素以及其他 members 都 copy 过来。

原则就是：对于 dynamic 的 members，`new` 一个新的并传递 rhs 的值；对于 static 的 members，直接传递。

非常值得注意的一个点是：`if(this != &rhs)` 这一句子赋值判断是很重要的，虽然这种自己给自己 assign 的情况不太常见，但是如果不加这一句子，那么在这种情况下 `delete[] elts` 就会删除自己的 dynamic array，从而这个 set 就没了。

20.3 The Rule of The Big Three

The Big Three 是指：

1. **Destructor**
2. **Copy Constructor**
3. **Assignment Operator (`operator=`)**

The rule of the Big Three 指的是：

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.

20.3.1 the Big Three 的特点：

1. 所有 objects 都有 the big three！！
2. 如果不写 custom implementation， the big three 都是有 Implicit 的。并且它们都有问题：implicit destructor 没有 spectial cleanup，implicit copy ctor 和 implicit assignment operator 都是 shallow copy 而不是 deep copy.

20.3.2 什么时候需要 custom Big Three

当我们需要 deep copy 的时候，通常是当 object owns and manages any resource(即 dynamic memory) 的时候。

20.3.3 Custom the Big Three 的注意事项

Destructor

1. **Free resources (e.g., dynamic memory)**

Copy Constructor

1. Copy regular members from other
2. Deep copy resources from other

Assignment Operator

1. **Check for self-assignment**
2. **Free old resources**
3. Copy regular members from rhs
4. Deep copy resources from rhs
5. **return *this**