19 Managing Dynamic Memory

Review: 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.

19.1 Building dynamic set with dynamic array

我们之前写的 set 的大小实际上是固定的。我们用了一个 const 的 Capacity 作为 array 大小，在此基础上通过限制用户能够接触到的元素来模拟 set 的大小变动。

这个 set 的一个很大的缺点是：set 的元素的数量上限很明确。如果我们把 `Capacity` 设为 100，那就只能装 100 个元素；如果我们把 `Capacity` 设为 10000，那么我们每建立一个 set 就会占据大量的 stack memory，即便我们只需要几个元素的 set.

而现在我们会写 dynamic array 之后，我们可以写一个真正大小动态变化的 set.

```
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);
}
```

然后我们写 ctors:

```
template <typename T>
ordered_set<T>::ordered_set()
  :size(0),
  current_capacity(DEFAULT_CAPACITY) {
    ele = new int[current_capacity];  // 创造 dynamic array
}

// overloaded ctor
template <typename T>
ordered_set<T>::ordered_set(int capacity)
  :size(0),
  current_capacity(capacity) {
    ele = new int[current_capacity];
}
```

这种只需要传一个参数的情况下我们也可以只写一个带有 default argument 的 ctor:

```
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(int capacity = DEFAULT_CAPACITY);
}
```

19.1.1 (题外话) constructor 的 default argument

C++ 的 class ctor 的 default argument 完全不如 python 灵活。

1. function call 的 arguments 顺序必须和 declaration 中的匹配。
2. **只有 declaration 的 parameter list 中的最后一个 parameter 才允许有 default value!**

所以基本没什么用。但是这种只有一个参数的 ctor 中还是比较有用的。

19.1.2 Destructor

我们发现，我们在 constructor 中创建了一个动态数组，但是我们并没有设置方法来删除它，这会导致 memory leak.

尤其是当我们在一些函数中创建 dynamic set 时：

```
void foo() {
  Unordered set is;
}
// memory leak when foo() is called
```

我们自然地想到一个办法，就是写一个 `Unordered_set::delete_array()` 之类的方法。但是：1. 它会破坏 encapsulation(封装)；2. 我们每创建一个 dynamic set 都要记得 call 这个函数.

C++ 提供了一个更加好的办法：destructor，简称 dtor.

Destructor run **automatically** when an object is destroyed. 比如当 `is` 是一个作为 local variable 的 set，它在出 scope 时会自动运行 dtor；当 `is` 是一个 global variable 时，他在 program 结束的时候会自动运行 dtor.

local variable:

```
int foo() {
  ordered_set is; // ctor runs
} // dtor runs
```

global variable:

```
ordered_set is; // ctor runs
int main() {}
// dtor runs
```

我们可以写一个能够 `delete` 所有由这个 class object 创建的 dynamic variables 的 destructor.

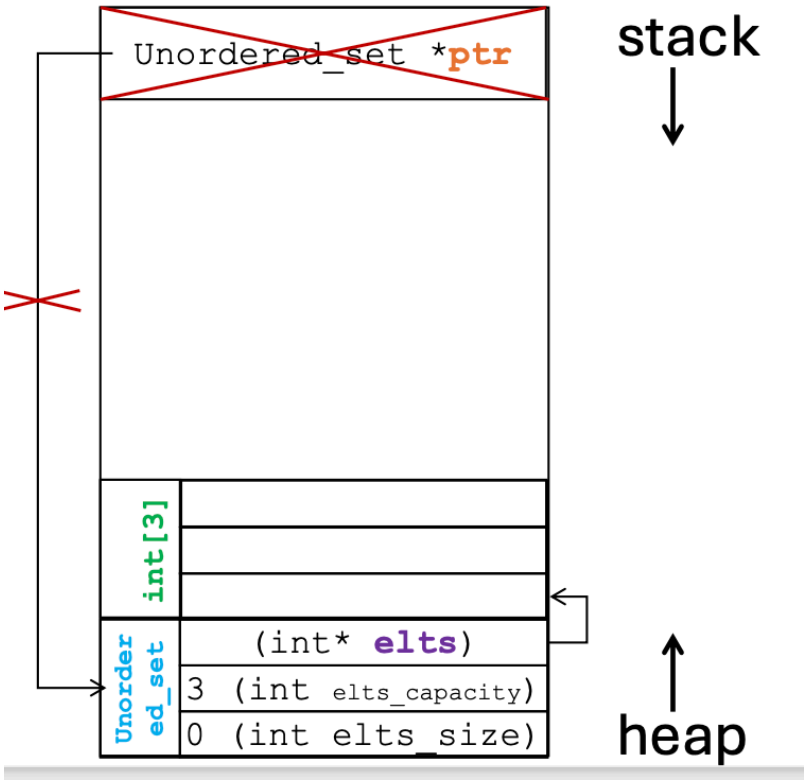
写法是：

```
class ordered_set {
public:
  //..
  ~ordered_set();
private:
  //..
};

ordered_set::~ordered_set() {
  delete[] ele;
}
```

19.1.3 如果 class object 本身是 dynamic 的: destructor 会在 `delete` class object 时自动运行

```
void foo() {
  ordered_set *ptr = new ordered_set(3);
  // memory leak
}
```



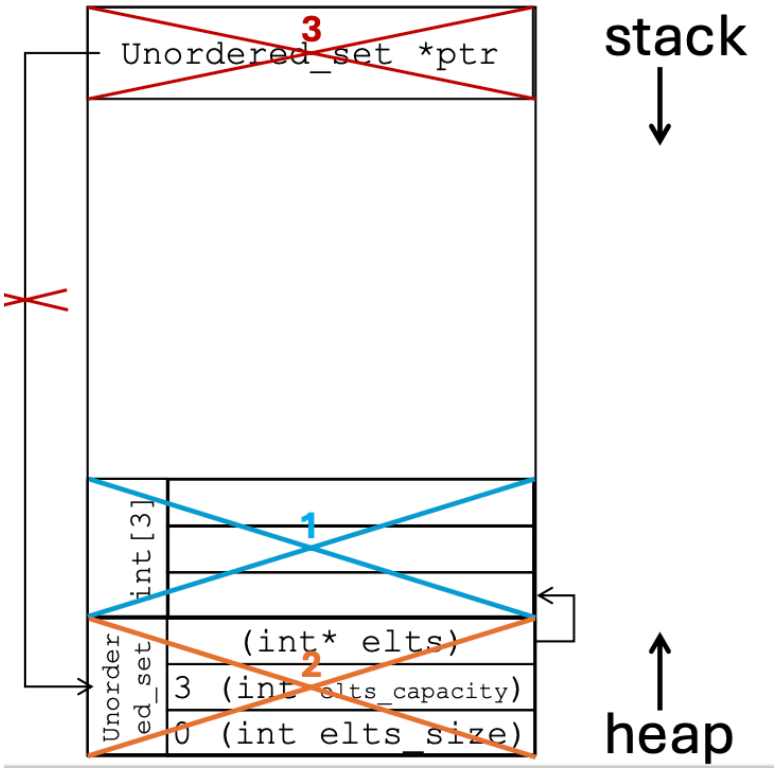
在这个函数运行结束后，有多少个 Bytes 的 memory 被 leak 了?

（假设 `int` 为 4B，`int *` 为 8B）

```
3 * sizeof(int) // int[3]
+ sizeof(int *) // ele
+ sizeof(int)   // current_capacity
+ sizeof(int)   // size
= 28B (leaked)
```

所以我们需要 `delete` 来防止这些 memory leak.

```
void foo() {
  ordered_set *ptr = new ordered_set(3);
  delete ptr; // ok
}
```



在我们 `delete ptr` ，也就是删除了 `ptr` 指向的 heap 中的 dynamic sorted_set object 之后，该 object 自动运行 dtor 删除了它的 dynamic variables.

19.1.4 RAII principle: Resource Acquisition is Initialization

RAII(Resource Acquisition is Initialization) 的意思是：一个 resource (比如一个 dynamic array) 的 life 应该被 bound to 需要这个 resource 的 object 的 lifetime (比如这个 dynamic array 对应的 sorted_set instance).

在 C++ 中就对应了使用 `new` 的 constructor 和使用 `delete` 的 destructor.

19.2 更改其他 function implementations 以适应 dynamic set

19.2.1 `insert`

由于对 dynamic array 的使用，我们应该更改原本的 `insert` 函数使其适应 dynamic set.

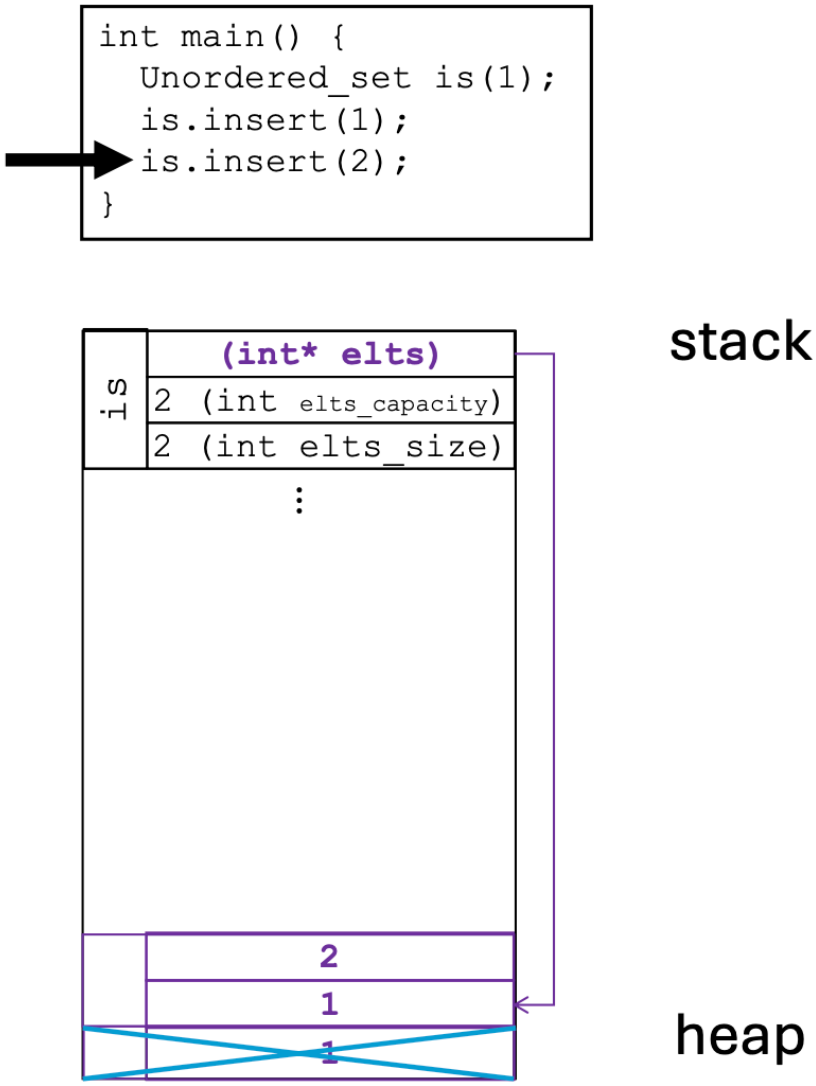
我们应该：

1. `new` 一个更大的 dynamic array.
2. 把原本的 dynamic array 复制进去.
3. `delete` 原本的 dynamic array.
4. 让 ptr 成员指向新的 dynamic array，并更新 capacity

```
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>
void ordered_set<T>::insert(T e) {
  if (contains(e)) return;
  if (size == current_capacity) {grow();} // grow capacity
  ele[size++] = v;  // increase size to catch up capacity; also inserts new element
}
```



19.3 Performance of Set

每当 `grow` 被 call 的时候，有多少 elements 被 copy 了?

```
int main() {
  ordered_set<int> is(1);
  int n;
  cin >> n;
  is.insert(0);
  is.insert(1); // grow(), 1 copy
  is.insert(2); // grow(), 2 copies
  //...
  is.insert(n); // grow(), n copies
  // total = 1+2+...+n = n(1+n)/2 copied
  // So the number of int-copy operations needed is in O(n^2)
}
```

我们发现这个算法的时间复杂度非常高

我们发现这个算法的时间复杂度非常高

应该优化一下。

注意到我们的条件：

```
if (size == current_capacity) {grow();} // grow capacity
```

直到 `size` 赶上 `capacity` 了，我们才使用 `grow()` 来重新建 array，重新 copy.

因而我们可以在 `grow` 中不要一格一格 grow，而是选择一次 `grow` 大一点，比如每次 `grow` 都把 array 的大小变成两倍

```
template <typename T>
void ordered_set<T>::grow() {
  int *tmp = new int[current_capacity * 2]; // grow 两倍
  for (int i = 0; i < size; i++) {
    tmp[i] = ele[i];
  }

  delete[] elts;
  elts = tmp;
  current_capacity *= 2;
}
```

这样的话：

```
int main() {
  Unordered_set is(1);
  is.insert(0);
  is.insert(1); // grow(), 1 copy, new capacity=2
  is.insert(2); // grow(), 2 copies, new capacity=4
  is.insert(3);
  is.insert(4); // grow(), 4 copies, new capacity=8
  is.insert(5);
  is.insert(6);
  is.insert(7);
  is.insert(8); // grow(), 8 copies, new capacity=16
  // …
}
```

现在对于创建 N 个元素的 set，copy 的总次数为：

$$\begin{aligned} T &= (N-1) + (N/2-1) + (N/4-1) + \cdots + 2 + 1 \\ &< N + N/2 + N/4 + \cdots + 2 + 1 = 2N \end{aligned} \tag{1}$$

因而 time complexity 马上下降到了 $O(n)$.

# elements	$(n^2-n)/2$	2n
1	0	2
8	28	16
64	2016	128
512	130816	1024
2048	2096128	4096

题外话: garbage collection 和 smart pointer

有一些 languages 允许 compiler 自动删除 dynamic variables，只要它们不再被需要。

这个行为叫做 **garbage collections.**

但是这个行为有一定的代价：首先肯定降低了运行速度，其次不清晰灵活：garbage collection behavior may vary, but manual deletion is a clear part of the standard.

我们现在目前为止讨论的都是 **raw pointers**. 而 C++ 11 以上的版本允许使用 **smart pointers**，这些 smart pointers 可以做一些 garbage collector 的事情。

`unique_ptr`: only a single pointer to the dynamic variable is possible

`shared_ptr`: multiple pointers to the dynamic variable are possible. **When all those pointers are destroyed, the dynamic variable is destroyed.**

`weak_ptr`: 只要所有指向一个 dynamic variable 的 `shared_ptr` 被 destroy，那么指向它的 `weak-ptr` 也马上被 destroy.