Object Array

发行版本 *1*

Darwin Yuan

2022 年 09 月 11 日

目录

1 简介 **3**

1.1 删除 . . . 4
1.2 Replace . . . 5
1.3 遍历 . . . 5
1.4 查找 . . . 6
1.5 切片 . . . 7
1.6 Clear . . . 10
1.7 ScopeView . . . 11
1.8 CleanUp . . . 13
1.9 排序 . . . 13
1.10 SortView . . . 15
1.11 Rotate . . . 16
1.12 索引与序号 . . . 17
1.13 对象数组 . . . 19

2 Placement **21**

2.1 平凡性 . . . 21
2.2 如何实现 . . . 22

3 ObjectArray **27**

3.1 自动选择 . . . 28
3.2 copy/move 构造与赋值 . . . 28
3.3 析构 . . . 31
3.4 Rule Of Five . . . 32

4 ArrayView **33**

4.1 ConstArrayView . . . 33
4.2 ArrayView . . . 34

4.3 自动识别 34

5 ScatteredArray **37**

6 有序数组 **39**

6.1 OrderedObjectArray 39

6.2 IndexedOrderedArray 40

6.3 自动选择 40

6.4 接口 40

7 Mixin **41**

7.1 核心算法和扩展算法 41

7.2 concept 42

7.3 分类 43

7.4 mixin 47

7.5 组合 47

7.6 CRTP 49

7.7 可见性 55

7.8 存在性 57

8 性能 **59**

8.1 ForEach 59

8.2 Find 62

8.3 MinElemIndex 64

备注： Object Array 是一套对静态对象对象数组的封装。其目标在于：

1. 安全
2. 用户友好
3. 灵活
4. 高效

代码仓库：https://github.com/godsme/object-array

CHAPTER 1

简介

`ObjectArray<T, N>` 像 *C* 语言数组 `T[N]` 一样，容量固定，元素个数可变。(如果元素个数固定（与容量相同)，请使用 `std::array<T,N>`)。

因而，你可以定义一个如下的数组：

```
ObjectArray<int, 10> array;

ASSERT(array.GetNum() == 0);

int* p = array.Append(5);
ASSERT(*p == 5);
ASSERT(array[0] == 5);

ASSERT(array.GetNum() == 1);
```

其空间代价，和运行时代价，与定义一个如下的结构体没有任何差别。

```
struct Array {
   int array[10];
   uint8_t num;
};
```

而其中 `Append` 操作，则相当于程序员亲自操作：

```
o.array[o.num++] = 5;
```

当然，Append 会提供 **边界检查**，当元素个数达到容量时，Append 操作会返回 nullptr (禁止使用异常情况下的常规设计)。

这种 **边界检查**，还体现在访问时：

```
int* p = array.At(1);
ASSERT(p == nullptr);
```

但是，如果你直接使用 operator[] ，则 **不会**进行 **边界检查**（这是一个基于接口设计、异常、性能等各种因素综合考虑后的设计决定)。

1.1 删除

既然可以向数组中添加元素，当然也可以删除和清理。

首先你可以通过 Erase 和 Remove , RemoveIf 来删除一个元素：

Erase 的参数是索引：

```
array.Erase(2);
```

而 Remove 的参数是对象指针：

```
array.Remove(&array[2]);
```

RemoveIf 的参数则是一个 **谓词**:

```
array.RemoveIf([value](auto&& item) { return item == value; });
```

这三个接口都是只删除数组中的一个元素。而对于 **无序**且 **连续存储**的数组，由于删除而导致的空位，将由最后一个元素填补（如果被删除的不是最后一个的话)：将最后一个元素，通过 std::move ，而不是 copy 到空缺位置。(因而数组内对象的 **move 构造**的实现很重要)。

另外，值得注意的是: RemoveIf 提供的是 **谓词**，因而，数组内 **谓词**成立的元素有可能不止一个。但 RemoveIf 只会删除依先后顺序找到的第一个。

如果想删除所有满足谓词的元素，则需要使用 ClearIf :

```
array.ClearIf([](auto&& item) { return item == 3; });
```

1.2 Replace

你可以通过 Replace 将某个位置的对象替换为另外一个对象：

```
struct Foo {
  Foo(char, bool);
  // ...
};

array.Replace(2, 'a', true); // array[2] is replaced by Foo{'a', true}
```

请注意，如果你想通过 move 方式用一个已有对象替换数组中某个位置的对象，需要明确进行 std::move：

```
Foo foo{'b', false};

array.Replace(2, std::move(foo));
```

Replace 的返回值为替换后的对象指针。需要注意的是，对于无序数组而言，替换前与替换后，元素在数组中的位置并不会发生变化。但对于有序数组，其位置可能会因为保持有序而变化，从而导致返回的指针与替换前访问同一位置得到的对象指针不同。

另外，如果你希望 Replace 操作返回索引值，而不是对象指针，可以调用 Replace_I 。

类似地，Append 与 Append_I 分别返回的是 **对象指针**与 **数组索引**。如果 Append 失败，返回 nullptr ；如果 Append_I 失败，返回 std::nullopt 。

1.3 遍历

如果你直接使用标准库算法来遍历一个 *C* 语言数组，则写法如下：

```
std::for_each(o.array, o.array + o.num, [](int item) { /* ... */ });
```

或者，干脆用索引遍历：

```
for(auto i=0; i<o.num; ++i) {
   auto&& item = o.array[i];
   /* do sth with item */
}
```

而使用 ObjectArray<T, N> ，则至少（随后我们会谈到更多其它方式）有四种方式来遍历：

首先，同样是 lambda 方式：

```
array.ForEach([](int item) { /* ... */ });
```

其次，也是通过索引方式：

```
for(auto i=0; i<array.GetNum(); ++i) {
   auto&& item = array[i];
   /* do sth with item */
}
```

第三，则是 *range-for* 方式：

```
for(auto&& item : array) {
  /* do sth with item */
}
```

最后，如果你想使用 *range-for* ，同时又需要索引值，则可以：

```
for(auto&& [item, i] : array.WithIndex()) {
  /* do sth with item & i */
}
```

1.4 查找

如果你直接使用标准库算法去 *C* 语言数组中查找一个元素，则写法如下：

```
auto* p = std::find_if(o.array, o.array + o.num, [](int item) { /* ... */ });
if(p == o.array + o.num) {
  // not found
} else {
  // found, get value by *p
}
```

而使用 `ObjectArray<T, N>` ，则写法如下：

```
auto* p = array.Find([](int item) { /* ... */ });
if(p == nullptr) {
  // not found
} else {
  // found, get value by *p
}
```

当然，如果你想查到的结果用索引表示，使用标准库的做法是：

```
auto iter = std::find_if(o.array, o.array + o.num, [](int item) { /* ... */ });
if(iter == o.array + o.num) {
  // not found
```

(续下页)

(接上页)

```
} else {
  auto index = p - o.array;
  // using index.
}
```

而使用 ObjectArray<T,N> ，做法为：

```
IntOpt<uint8_t> index = array.FindIndex([](int item) { /* ... */ });
if(!index) {
  // not found
} else {
  // using *index.
}
```

当然这类的算法还有很多，即便对于同一种算法，也有不同的参数形式，这里就不再赘述。

而 ObjectArray<T, N> 这是把这些对于算法的经常性操作中的不必要的重复性因素都排除掉，让用户只提供和关注他必须提供和关注的因素。

1.5 切片

切片 (*slice*) 本身并不是数组，而是一种对于数组的 *view* 。这种概念对于数组的操作极为有价值，很多编程语言都内建了切片概念（*python* , *go* , *rust* 等），*C++ 20* 也通过 std::span 达到了部分对于 *slice* 的支持。

而 ObjectArray<T,N> 的 *slice* 则非常直接：

```
auto&& slice = array.Slice(3, -2);
```

这样，你就得到了一个 *slice* ，其表达的范围为: [3, n-2) 。

备注: 用负数作为一种索引方式，很多没有接触过切片概念的人会觉得陌生和唐突。尤其是用 -1 表达数组的最后一个元素，用 -n 表达倒数第 n 个元素。这与我们用 0 表达第一个元素的习惯，看起来是不一致的。

当然，首先是因为 0 已经被用来表达数组的第一个元素了。不可能再用它表达倒序的第一个元素。

其次，即便我们直接用正索引来表达数组的倒序元素，也是通过 array[num-1] 来表达数组的最后一个元素；用 array[num-n] 来表达倒数第 n 个元素。

因而，它本质上与我们之前的习惯是完全一致的。

如果依然觉得不一致，那就怪最初数组索引的设计者没有用 1 而是用 0 当作数组第一个元素的索引吧。

当然，如果你只想指定 *slice* 其中一边的边界，则可以通过：

```
array.From(3);  // [3, n)
array.Until(-2) // [0, n-2)
```

这里的切片索引方案与 *python* 一致。当给出两边的边界，比如 `Slice(m, n)` 时，其包含的元素索引边界为: *[m, n-1]* 或 *[m, n)* 。因而:

```
array.From(-2)   // last two items in the array
array.Until(-2)  // everything except the last two items
```

而切片也提供了几乎所有 *ObjectArray<T,N>* 的算法接口:

```
array.From(3).ForEach([](auto&& item) { /* */ });
array.Until(-2).MinElem();
```

当然，你也可以 `range-for`:

```
for(auto&& item: array.Slice(3, -2)) {
   // ...
}
```

在指定切片边界时，用户有可能给出超出数组事实边界的索引，比如:

```
ObjectArray<int, 10> array{1,2,3};

array.Slice(-4, -5); // both are beyond left boundary.
```

Object Array 会自动对超越边界的索引进行调整:

1. 如果超越左边界，会调整为左边界
2. 如果超越右边界，会调整为右边界
3. 如果用户给定的 *Slice* 范围不能构成有效范围，则整个 *Slice* 的范围为空。

比如:

```
ObjectArray<int, 10> array{1,2,3};

array.Until(10);    // [0, 2]
array.From(-10);    // [0, 2]
array.Slice(1, 10); // [1,2]

array.Slice(-10, -20); // empty
array.Slice(-10, 0);   // empty
array.Slice(10, 20);   // empty
```

另外，你不能在一个 **右值**数组上创建一个 *slice* 。也就是说下面的代码是不被允许的:

```
auto&& slice = ObjectArray<int, 10>{1,2,3).Slice(2,-3); // not allowed
```

因而，你也不能在 *range-for* 表达式里写如下代码：

```
for(auto&& item : ObjectArray<int, 10>{1,2,3}.Slice(2,-3)) { // not allowed
   // do sth.
}
```

注意： 这是因为，*slice* 仅仅是一个 *view* ，因而必然会引用一个数组。如果数组本身是一个 **右值**，其生命周期仅仅在那一行代码得以维持。在那行代码执行结束后，**右值**数组就被销毁了，导致 *slice* 变成了 *dangling slice* 。*ObjectArray* 通过设计，保证这样的代码会导致编译错误。

但是，如果你使用的是一个 *ArrayView* (我们会在后面章节介绍到)，则可以在右值对象上创建 *slice* :

```
int a[3] = {1,2,3};
uint8_t num = 3;

auto&& slice = ArrayView{a, num}.Slice(1, -1); // OK
```

注意： 当创建并使用一个 **左值** *slice* 期间，一定要保证 *slice* 所引用的数组内元素是不可被删除的。否则，将可能会导致不确定的行为。

比如，我们在一个元素个数为 3 的数组上创建了一个 *slice* : `array.Until(3)` 。这时 *slice* 所引用的范围为 `[0, 3)` ，范围内有 3 个元素。如果此时我们删除了数组内一个元素。此时数组内元素个数为 2 ，但 *slice* 的范围依然是 3 ，从而在访问时可能会引起问题。

虽然通过增加判断逻辑可以避免这类问题，但基于绝大多数情况下都不会出现这样的问题。因而基于性能考量，把约束的保证留给用户。

rust 通过语言的内建约束，来保证一个数组被至少一个 *slice* 引用时，数组本身是完全不允许修改的。

但本库的 *slice* 除了与删除元素有关的所有操作被禁止外，其它的修改操作：增加元素、或替换元素相关的操作依然是允许的。

不过需要注意的是：如果在创建一个 *slice* 时，如果 **没有**指定右边界 (通过 `From(n)`)，则 *slice* 的右边界将会随着元素的增加而增大。

1.6 Clear

在看到 *slice* 的索引方式之后，事实上 `Clear` 也可以在一个切片范围内清除：

如果想把数组所有元素清理掉，则可以调用 `Clear`：

```
ASSERT(array.GetNum() == 5);
array.Clear();
ASSERT(array.GetNum() == 0);
```

如果仅仅想清除 *[from, until)* 范围内的元素，则可以通过指定切片范围来调用 `Clear`：

```
ASSERT(array.GetNum() == 5);
ASSERT(array[0] == 1);
ASSERT(array[1] == 2);
ASSERT(array[4] == 5);

array.Clear(1, -1); // [1, 3] is cleared.

ASSERT(array.GetNum() == 2);

ASSERT(array[0] == 1);
ASSERT(array[1] == 5); // array[4] is moved to array[1]
```

像 *slice* 一样，在清理时，可以只指定范围的其中一端：

```
array.ClearFrom(2); // [2, n)
array.ClearUntil(-2); // [0, n-2)
```

除了无条件进行 `Clear` 之外，你还可以通过 `ClearIf` 以指定谓词的方式进行清理，即在一个范围内，所有满足谓词条件的，都会被清理。

```
array.ClearIf([](auto&& elem) { return elem > 2; });

// range [1, -1)
array.ClearIf(1, -1, [](auto&& elem) { return elem > 2; });

// range [1, n)
array.ClearFromIf(1, [](auto&& elem) { return elem > 2; });

// range [0, n-1)
array.ClearUntilIf(-1, [](auto&& elem) { return elem > 2; });
```

对于需要在一个范围内进行清理的操作，你也可以通过一个 **右值** `Slice` 进行：

```
array.Slice(1,-1).Clear();
array.From(1).ClearIf([](auto&& elem) { return elem > 2; });
```

但一个 **左值** Slice 不允许进行这种操作：

```
auto&& slice = array.Slice(1,-1);

slice.Clear(); // not allowed, compiling fail.
slice.ClearIf([](auto&& elem) { return elem > 2; }); // not allowed, compiling fail.
```

> **注意：** 这是因为，**左值** slice 的生命周期很久，你可以多次调用其不同接口，因而必须维持一个 Slice 的语义完整性。
>
> 但一个右值 Slice 在你调用 Clear 相关接口之后，就没有途径再次访问同一 slice，因而不需要维持其语义完整性。

1.7 ScopeView

切片仅仅能够指定一个数组两边的边界，从而对边界内的 **连续**范围内的元素进行访问。

但现实中，存在着一种需求：我们只对数组中非连续的 **散列**范围内的元素感兴趣。比如，我只对数组中的第 1, 3 , 7 个元素感兴趣。

此时，我们就可以通过一个 bitset 来指定范围，从而得到一个 *ScopedView*：

```
auto&& scopedView = array.Scope(0x4a);
```

而如果你想访问 **排除**了这些元素的其它元素，则可以：

```
auto&& scopedView = array.Exclude(0x4a);
```

当然对于这些 *ScopedView*，你同样可以使用所有的数组算法：

```
array.Scope(0x4a).ForEach([](auto&& item) { /* */ });
array.Exclude(0x4a).MaxElem([](auto&& l, auto&& r) { return l > r });
```

当然，你也可以 range-for：

```
for(auto&& item: array.Scope(0x3a)) {
   // ...
}
```

和 *slice* 一样，当你进行 range-for 时，如果需要索引，你得到的是数组的索引，而不是在 *ScopedView* 内的索引：

```
for(auto&& [item, i] : array.Scope(0xf4).WithIndex()) {
   // the 1st `i` is 2, 2nd `i` is 4, and so on...
}
```

更进一步的，你可以将两种 *view* 组合起来：

```
array.From(2).Scope(0xf4);
```

注意:

1. *Slice* 需要放在前面
2. *Scope* 里的位图仍然是以数组的索引，而不是 *Slice* 的范围来索引的；当然，在 *Scope* 里超出 *Slice* 范围的元素不在 *Scope* 的访问范围内。

而指定 *Scope* 范围的访问方式，不仅仅可以通过创建一个 *ScopedView* ，还可以直接通过算法参数来指定。比如：

```
array.ForEach(0xa5, [](auto&& item) {});
array.Scope(0xa5).ForEach([](auto&& item) {});

array.MinElemEx(0xa5);
array.Exclude(0xa5).MinElem();

array.From(3).MaxElem(0xa5);
array.From(3).Scope(0xa5).MaxElem();
```

以上三组例子，两种写法从作用上是等价的。

注意:

- *ScopedView* 可以通过算法参数来替代，但 *Slice* 不能；
- 当使用 *range-for* 时，*ScopedView* 不可能通过算法参数来替代。

另外，基于与 *slice* 同样的原因，你不能在一个右值数组对象上创建一个 *scope view* ：

```
auto&& view = ObjectArray<int, 10>{1,2,3}.Scope(0x0a}; // not allowed.
```

但你却可以在一个右值 *ArrayView* 上创建一个 *scope view* ：

```
int a[3] = {1,2,3};
uint8_t num = 3;
```

(续下页)

(接上页)

```
auto&& view = ArrayView{a, num}.Scope(0x0a}; // OK.
```

1.8 CleanUp

在了解了 *scope* 的概念之后，事实上 `CleanUp` 也可以在一个范围内清除：

```
array.CleanUp(0x0a);   // 1, 3 is cleared.
array.CleanUpEx(0x0a); // `Ex` Means `Exclude`, so 0, 2, 4 is cleared.
```

1.9 排序

对于任何一种可修改的 *array* 或者 *view* ，你都可以对其进行直接的排序：

```
ObjectArray<int, 10> array{3,1,4,2};

array.Sort();

ASSERT(array[0] == 1);
ASSERT(array[1] == 2);
ASSERT(array[2] == 3);
ASSERT(array[3] == 4);
```

或者进行降序排序：

```
array.DescSort();

ASSERT(array[0] == 4);
ASSERT(array[1] == 3);
ASSERT(array[2] == 2);
ASSERT(array[3] == 1);
```

你能直接进行排序的原因是：数组中的对象本身可以进行 < 操作。如果不能，你就需要通过 `lambda` 指明排序规则：

```
ObjectArray<Foo, 10> array{{3}, {1}, {4}, {2}};

array.DescSort([](auto&& l, auto&& r) { return l.a < r.a; });

ASSERT(array[0].a == 4);
```

(续下页)

(接上页)

```
ASSERT(array[1].a == 3);
ASSERT(array[2].a == 2);
ASSERT(array[3].a == 1);
```

如果你希望使用 **稳定排序**算法，则可以调用 StableSort：

```
ObjectArray<Foo, 10> array{{3}, {1}, {4}, {2}};

array.StableSort([](auto&& l, auto&& r) { return l.a < r.a; });

ASSERT(array[0].a == 1);
ASSERT(array[1].a == 2);
ASSERT(array[2].a == 3);
ASSERT(array[3].a == 4);
```

当对象本身支持 < 操作时，StableSort 也提供了降序排序接口 StableDescSort 。

备注： StableSort 比 Sort 性能要差，但却可以保证两个相等的对象在排序后，与排序前的顺序相同。

如果你只想对部分元素进行排序，即从整个数组中，排序出最大/最小的 N 个元素，则可以使用 PartialSort：

```
ObjectArray<Foo, 10> array{{3}, {1}, {4}, {2}};

array.PartialSort([](auto&& l, auto&& r) { return l.a < r.a; }, 3);

ASSERT(array[0].a == 1);
ASSERT(array[1].a == 3);
ASSERT(array[2].a == 4);
ASSERT(array[3].a == 2);
```

而排序不仅仅可以在整个数组范围内进行，还可以只在一个 *slice* 范围，或者（和）*scope* 范围内进行排序：

```
ObjectArray<int, 10> array{3,1,4,2};

array.Slice(1, -2).DescSort();

ASSERT(array[0] == 3);
ASSERT(array[1] == 4);
ASSERT(array[2] == 1);
ASSERT(array[3] == 2);
```

```
ObjectArray<int, 10> array{3,1,4,2};

array.Scope(0x06).DescSort();

ASSERT(array[0] == 3);
ASSERT(array[1] == 4);
ASSERT(array[2] == 1);
ASSERT(array[3] == 2);
```

```
ObjectArray<int, 10> array{3,1,4,2};

array.Slice(0, -2).Scope(0x06).DescSort();

ASSERT(array[0] == 3);
ASSERT(array[1] == 4);
ASSERT(array[2] == 1);
ASSERT(array[3] == 2);
```

1.10 SortView

对于数组而言，排序操作会导致对象在数组中的位置进行移动，如果对象比较大，这是一个昂贵的操作。

如果我们只是在某次需要时，对数组进行排序，但并不想改变数组本身的元素顺序，则可以通过 `SortView` 进行排序。

```
ObjectArray<int, 10> array{3, 1, 4, 2};

auto&& view = array.SortView().Sort();

ASSERT(view[0] == 1);
ASSERT(view[1] == 2);
ASSERT(view[2] == 3);
ASSERT(view[3] == 4);

// array itself still keeps its order.
ASSERT(array[0] == 3);
ASSERT(array[1] == 1);
ASSERT(array[2] == 4);
ASSERT(array[3] == 2);
```

当然，通过 *SortView* 也可以进行 `StableSort` 和 `PartialSort`：

```
auto&& view = array.SortView();

view.PartialSort(3);

ASSERT(view.GetNum() == 3);
ASSERT(view[0] == 1);
ASSERT(view[1] == 2);
ASSERT(view[2] == 3);
```

当然，你也可以连写：

```
auto&& view = array.SortView().PartialSort(3);
```

备注： *SortView* 本身是对数组的索引进行排序，而不是对对象直接排序，以降低数组元素移动所带来的成本。

而 *SortView* 也可以在 *Slice* (或/和) *Scope* 范围内创建：

```
ObjectArray<int, 10> array{3,2,4,1};

auto&& view = array.From(1).Scope(0x0c).SortView().Sort();

// indices are slice ones.
ASSERT(view[0] == 1);
ASSERT(view[1] == 4);
ASSERT(view.GetNum() == 2);
```

1.11 Rotate

如果你想对数组内某个范围的元素进行 **旋转** (rotate) 操作，可以直接调用数组的 `RotateLeft` 或 `RotateRight` 操作：

```
ObjectArray<int, 10> array{3,2,4,1};

array.RotateLeft(); // 2, 4, 1, 3
array.RotateRight(); // 1, 3, 2, 4
```

事实上，这两个函数都有一个参数: `n`，即旋转的次数（默认值为 `1`）。因而你可以：

```
ObjectArray<int, 10> array{3,2,4,1,5};
```

(续下页)

(接上页)

```
array.RotateLeft(2);  // 4, 1, 5, 3, 2
array.RotateRight(2); // 1, 5, 3, 2, 4
```

通过 RangeRotateLeft(from, until, n) 与 ``RangeRotateRight(from, until, n)，你可以在指定旋转的范围：

```
ObjectArray<int, 10> array{3,2,4,1,5};

array.RangeRotateLeft(1, -1, 2);  // 3, 1, 2, 4, 5
array.RangeRotateRight(1, -1, 2); // 3, 4, 1, 2, 5
```

像 Slice 一样，你可以只指定范围边界的一侧：

```
ObjectArray<int, 10> array{3,2,4,1,5};

array.RotateLeftFrom(1, 2);   // 3, 1, 5, 2, 4
array.RotateLeftUntil(-1, 2); // 4, 1, 3, 2, 5
```

自然，你也可以通过 Slice 来进行 Rotate：

```
ObjectArray<int, 10> array{3,2,4,1,5};

array.Slice(1, -1).RotateLeft(2);  // 3, 1, 2, 4, 5
array.Slice(1, -1).RotateRight(2); // 3, 4, 1, 2, 5

array.From(1).RotateLeft(2);   // 3, 1, 5, 2, 4
array.Until(-1).RotateLeft(2); // 4, 1, 3, 2, 5
```

1.12 索引与序号

当你通过 *range-for* 对各种 *Array/View* 进行遍历时，除了数组元素之外，你或许还需要“索引”值。

但对于索引的需要至少有两种：首先是它们在数组中的索引；其次，是遍历集合的序号。

比如，当你对一个 Slice 进行 *range-for* 操作。由于切片只是数组元素的一个子集，因而遍历过程中，每个元素在数组中的索引，与遍历序号是不一致的。但这两种需求都存在。

如果你需要的是一个元素在数组中的索引值，可以使用 WithIndex；而如果你需要的是遍历序号，则使用 Enumerate。比如：

```
auto&& slice = array.From(2);

for(auto&& [elem, i] : slice.WithIndex()) {
```

(续下页)

(接上页)

```
    // i start from 2
}

for(auto&& [elem, i] : slice.Enumerate()) {
    // i start from 0
}
```

而对于不连续的 ScopeView，比如：

```
auto&& scope = array.Scope(0x2a); // 1, 3, 5

for(auto&& [elem, i] : scope.WithIndex()) {
    // i is 1, 3, 5 in turn.
}

for(auto&& [elem, i] : scope.Enumerate()) {
    // i is 0, 1, 2 in turn.
}
```

而你如果通过 SortView 对一个数组进行排序，当你遍历 SortView 是，WithIndex 得到的依然是数组索引：

```
ObjectArray<int, 10> array{3,1,4,2};

auto&& sorted = array.SortView().Sort();

for(auto&& [elem, i] : sorted.WithIndex()) {
    // i is 1, 3, 0, 2 in turn
}

for(auto&& [elem, i] : sorted.Enumerate()) {
    // i is 0, 1, 2, 3 in turn
}
```

重要： 无论你对任何数组，或者 *view* 进行遍历时，WithIndex 总是得到在数组内的索引；Enumerate 总是得到序号。

除了在 *range-for* 时之外，如果用户在调用 ForEach 等接口时，如果传入的函数需要索引，得到的索引也和 *range-for* 一样，是数组的索引。

不过，用户调其它接口时，比如 operator[] 、At 、Replace 时，使用的索引则是在 *slice* 范围内的索引。

1.13 对象数组

不同于 *C* 语言数组，`ObjectArray<T, N>` 允许存放任意的 *C++* 对象。

一旦允许存放对象，则设计上的需要考量的复杂度将大幅上升。在后续章节里，我们将详细讨论与之有关的因素及设计决定。

CHAPTER 2

Placement

Placement 是 *ObjectArray* 的基石。而讨论 *Placement* 的作用及性质，则要从 **平凡性**谈起。

2.1 平凡性

关于平凡性的更多细节，可参考 平凡性 。这里仅仅谈论两个与 *Placement* 有关的：

1. 如果一个类是 **可平凡构造**的，系统不会为对象的创建，生成任何构造代码；
2. 如果一个类是 **可平凡析构**的，则系统不会为对象的销毁，生成任何析构代码。

因而，如果一个类，即是 **可平凡构造**，又是 **可平凡析构**的，你就无需担心在一个数组中直接存放这种类型。(事实上，一个类如果是 **可平凡构造**的，就必然是 **可平凡析构**的；但反过来不成立)。

因为静态数组的空间是固定的，但其中元素的个数却是可变的。你不希望在数组事实上还是空的时候，就将数组中所有的 *slot* ，都调一遍 **默认构造函数**（默认的并不一定是平凡的)；在数组销毁的时候，即使数组是空的，也会再将所有对象都销毁一遍。

更不用说，如果一个类没有 **默认构造函数**，则数组中的对象将必须由用户亲自明确构造。但事实上此时用户又不知道该怎样构造（毕竟还没有实际对应的对象)。此时用户只有两种选择：

1. 为其构造一套非法值，或者默认值（这相当于还是为其提供了 **默认构造函数**)；
2. 还是回到问题的本质：静态数组只是代表了空间的预留，而并非已有实际的对象，此时，就应该用一种类似于”占位符”（*PlaceHolder/Placement* ）的概念来表达此种语意。

而一旦选择了 `Placement` 的方式，将直接带来两种好处：

1. 让问题回归其本质；

2. 让非平凡的对象在不实际需要创建之前，系统什么都不用做。

完美！

理论上，即便一个类型其 **构造**和 **析构**都是平凡的，如果也使用 *Placement* ，除了语法上，用户需要通过通过 `Emplace` 来构造对象，通过 `operator*` 来访问对象之外也没有什么实际的坏处。

但用户总是觉得在不必要时，还是直接来得痛快。因而就需要选择：

1、当一个类型构造和析构都是平凡时，直接使用类型；2、否则，使用 `Placement` 。

但选择不是免费的，它需要程序员付出脑细胞的代价。更重要的是，世界是变化的，一个曾经无比平凡的类，也可能在某个时候悄悄地变得不再平凡。系统则在悄无声息中偷偷地增加了运行时代价。

面对这样的困境，至少有两种解决方案：

1. 放弃做出选择，一致使用 `Placement` ；
2. 让机器帮我们做出选择。

作为有追求的程序员，我们毫无疑问会直奔第二种方案。

2.2 如何实现

作为一种针对某种类型对象的占位符，`Placement<T>` 必须提供如下特质：

1. 其所占内存的大小必须和 `T` 的大小相等；
2. 其所占内存的对齐方式必须和 `T` 相同；

否则，将来就无法恰如其分的在此位置上创建对象。因而，我们快速给出如下的实现：

```
template <typename T>
struct Placement {
   alignas(alignof(T)) char storage[sizeof(T)];

   // ....
};
```

对于这样的 `Placement` ，你不可能给出任何有价值的 **非平凡构造**和 **非平凡析构**。毕竟，你没有任何额外的信息来记录在 `storage` 上究竟已经放置了一个有效的对象，还是依然保持无效。

除了什么都不做，你别无选择。

因而，对于任何 `Placement<T>` ，其都是 **可平凡构造/析构**的。即便 `T` 本身完全不平凡（这正是我们需要 `Placement` 的动机）。

这就导致了 `T` 本身的平凡性信息在这里被丢失了。

当然，在 `Placement` 内部，你是无从知道对象是否已经被创建的。在没有被创建的情况下，平凡性信息的丧失无足轻重（甚至就应该一切都是平凡的）；但在对象已经存在情况下，这种信息的丧失就会导致风险。

比如，一个类 Foo 持有一个 unique_ptr ，因而必然是 **非平凡构造/析构**的，更重要的是，它必然是禁止 *copy* 的。

而如果有一段框架代码通过判断一个类是否是可平凡拷贝，来自动进行 ::memcpy 的话，Placement<Foo> 会被判断为可拷贝的，导致一个对象被两个 unique_ptr 所有，最终导致系统的崩溃。

由于 Placement<T> 自身具备的 **二态性**（对象存在与否），导致无论你怎么看待它的平凡性，似乎都是不够通顺的。

而正是这个原因，在 *C++ 11* 之前，同样具备这种二态（甚至多态）性的 *union* 完全不允许持有任何 **非平凡类型**。

到了 *C++ 11* ，*union* 的这种约束被取消。而上述矛盾的解决办法则是：如果一个 *union* ，其内部任何一个成员，如果其某个特殊函数是 **非平凡**的，则整个 *union* 则会删除对应的特殊函数，从而自动丧失对应函数的平凡性。

比如下面的 *union* :

```
union {
  std::string           s;
  char                  c;
};
```

由于 std::string 拥有明确的 **自定义构造、析构、copy/move 构造/赋值**（因而这些特质都是非平凡的），即便最终你在此 *union* 上实际构造的是无比平凡的 char c ，但整个 *union* 的 **构造、析构、copy/move 构造/赋值**都依然会统统被删除。

现在已经不是平不平凡的问题，而是存不存在的问题。这些函数的删除，导致你完全无法对这个 *union* 做任何事情：无法构造，无法拷贝，无法移动，当然也就谈不上析构。

而一个对象能够被 **构造**和 **析构**是最基本的需求。这就强迫程序员必须手动为其明确定义构造和析构。而你一旦为其明确定义了 **构造**，其将不再是 **可平凡构造**的；同样，一旦你明确为其定义了 **析构**，则其 **默认构造**与 **析构**都不再平凡。

像 *Placement* 一样, *union* 自身并不具备哪个成员有效的信息。所以这种强迫性主要在 **匿名 union** 的场景下特别有意义。比如：

```
struct Foo {
private:
   enum class Kind {
     NIL,
     STRING,
     CHAR
   };

public:
   Foo() : kind{Kind::NIL} {}
   Foo(std::string const& str) : s{str}, kind{Kind::STRING} {}
```

(续下页)

(接上页)

```
    Foo(char ch) : c{ch}, kind{Kind::CHAR} {}

    ~Foo() {
        if(kind == Kind::STRING) {
            using namespace std;
            s.~basic_string();
        }
    }

private:
    union {
      std::string s;
      char        c;
    };
    Kind kind;
};
```

在匿名的场景下，其外围的类 `Foo` 变为其宿主，因而由 `std::string` 所带来的构造/析构/赋值函数的删除问题发生在 `Foo` 身上，而 `Foo` 作为一个 *class* ，可以拥有 *union* 自身所不可能拥有的谁有效的信息。这就会有效的强迫程序员必须明确的为 `Foo` 实现 **构造**和 **析构**（如果需要，还要实现 **copy/move 构造/赋值**函数，上述实现中它们依然处于被删除状态）。

但我们的 *Placement* 实现显然不能使用上述匿名 *union* 技术，因为我们的 *Placement* 有可能被用在不同场景（数组，*optional* ）等，因而 *Placement* 必须保持像 *union* 一样对自身状态的无知（否则就需要额外的内存来保存状态信息）。

所以，如果我们想让 *Placement* 携带足够的平凡性信息（以及其它诸如是否可拷贝/移动/复制等能力信息），同时又增加额外的内存开销，以保持 *Placement* 本身的职责，那么在 *Placement* 上添加任何 **非平凡构造/析构**就没有任何意义。

我们只需要在 *union* 上添加即可。

```
template< typename T
        , bool = std::is_trivially_default_constructible_v<T>
        , bool = std::is_trivially_destructible_v<T>>
struct UnionTrait {
    union Type {
        T obj;
    };
};

template<typename T>
struct UnionTrait<T, true, false> {
    union Type {
```

(续下页)

(接上页)

```
        ~Type() {}
        T obj;
    };
};

template<typename T>
struct UnionTrait<T, false, true> {
    union Type {
        Type() {}
        T obj;
    };
};

template<typename T>
struct UnionTrait<T, false, false> {
    union Type {
        Type() {}
        ~Type() {}
        T obj;
    };
};

////////////////////////////////////////
template<typename T>
struct Placement {
   Placement() = default;
   ~Placement() = default;
   //....
private:
    using Storage = typename UnionTrait<T>::Type;
    Storage storage;
};
```

通过简单的模版编特化技术，我们让 *Placement<T>* 可以被构造和析构（平凡与否则取决于 *T* 的构造/析构是否平凡），同时继续保持 *T* 所导致的其它特殊函数（*copy/move* 构造/复制）的状态。即，如果 *T* 所在的 *union* 导致它们中某些或全部被删除，则 *Placement<T>* 也拥有同样的性质。

而 *Placement<T>* 所携带的由 *T* 和 *union* 所导致的状态信息，会进一步传播到使用 *Placement<T>* 的类，至于它们会如何应对，则不再是 *Placement<T>* 所需关心的。

毕竟，所有 *T* 所拥有的非平凡特殊函数在 *Placement<T>* 上对应的函数都被删除了，这种信息已经强大到任何使用 *Placement<T>* 的类都在需要时不可能忽视掉（毕竟，还有什么比删除掉不让你用，你一用就编译出错更强大的信息呢?），因而也不会导致潜在的风险与错误。(不得不佩服 *C++ 11 union* 提案的深思熟虑)。

最后再强调一下：*Placement<T>* 自身并不说明其所持内存上对象的有效性。*Placement<T>* 在销毁时，也无法

自动调用 *T* 的析构。因而，对其所持对象的销毁，是用户的责任。而用户使用 *Placement<T>* 时，必须要自定义说明 *T* 有效性的方式。比如，像 *optional<T>* 那样，通过一个 *bool* 类型的标记来说明，或在数组中，通过数组中元素的数目来说明。

CHAPTER 3

ObjectArray

有了 `Placement` ，我们现在就可以设计我们的数组。

首先，让我们搭一个基本的架子：

```
template <typename OBJ, std::size_t MAX_NUM>
struct Array {
   using SizeType = DeduceSizeType_T<MAX_NUM>;
   using ElemType = DeduceElemType_T<OBJ>

   ElemType elems[MAX_NUM];
   SizeType num{};
};
```

在这个简单的架子中，有几个要点：

1. 模版参数很直接，说明了我们希望存放在数组中的对象类型，以及预留的个数。
2. 由于我们希望自动选择，对于 **可平凡构造**的类型直接存放对象类型 `OBJ` ；否则存放 `Placement<OBJ>` ；而这个选择，由 `DeduceElemType_T<OBJ>` 完成。
3. 由于我们希望尽可能不浪费内存，按照 `MAX_NUM` 我们可以自动推理出最小的 `SizeType` 。

3.1 自动选择

了解了 `Placement` ，我们就知道一个静态数组，是否使用 `Placement` 。简单说，原则就是：如果一个类型 `T` 本身的 **默认构造**是平凡的（默认构造的平凡性，必然意味着析构的平凡性），那么我们就让数组的元素类型直接使用 `T` ，否则，则使用 `Placement<T>` 作为数组元素。

```
template<typename T>
auto DeduceElemType() -> auto {
   if constexpr(std::is_trivially_default_constructible_v<T>) {
       return T{};
   } else {
       return Placement<T>{};
   }
}

template<typename T>
using DeduceElemType_T = decltype(DeduceElemType());
```

3.2 copy/move 构造与赋值

一个 *array* ，当然至少是应该能够 copy/move 构造的。

我们先增加比较简单的 **copy 构造**:

```
template <typename OBJ, std::size_t MAX_NUM>
struct Array {
   using SizeType = DeduceSizeType_T<MAX_NUM>;
   using ElemType = DeduceElemType_T<OBJ>
   using Trait    = ObjectTrait<ElemType>;

 private:
     auto ConstructFrom(ElemType* array) -> void {
         if constexpr (std::is_trivially_copyable_v<ELEM>) {
             ::memcpy(elems, array, sizeof(ELEM) * num);
         } else {
             for(int i=0; i<num; i++) {
                 Trait::Emplace(elems[i], Trait::ToObject(array[i]));
             }
         }
     }

public:
   Array(Array const& rhs) : num{rhs.num} {
```

(续下页)

(接上页)

```
        ConstructFrom(rhs.elems);
    }

protected:
    ElemType elems[MAX_NUM];
    SizeType num{};
};
```

对于数组的拷贝构造，很简单，是按照对方数组的实际数目，将对方的元素一个个 **copy 构造**给自己。

但这中间有一个重要的优化点。如果我们数组的元素类型是 **可平凡拷贝**的（这就意味着一定是可平凡默认构造的，按照我们的自动选择规则，*ElemType* 一定不可能是 *Placement<T>*；但即便是 **可平凡默认构造**的，也并不代表它是 **可平凡拷贝**的），我们就可以直接 `::memcpy`，这往往比一个个进行 **copy 构造**有更好的性能。

我们再增加 **move 构造**:

```
template <typename OBJ, std::size_t MAX_NUM>
struct Array {
    using SizeType = DeduceSizeType_T<MAX_NUM>;
    using ElemType = DeduceElemType_T<OBJ>;
    using Trait    = ObjectTrait<ElemType>;

 private:
     template<typename U>
     auto ConstructFrom(U* array) -> void {
         if constexpr (std::is_trivially_copyable_v<ELEM>) {
             ::memcpy(elems, array, sizeof(ELEM) * num);
         } else {
             for(int i=0; i<num; i++) {
                 Trait::Emplace(elems[i], std::move(Trait::ToObject(array[i])));
             }
         }
     }

     auto ClearContent(SizeType from) -> void {
         if constexpr (!std::is_trivially_destructible_v<ELEM>) {
             for(int i=from; i<num; i++) Trait::Destroy(elems[i]);
         }
     }

     auto Clear() -> void {
         ClearContent(0);
         num = 0;
     }
```

(续下页)

(接上页)

```
    auto MoveFrom(ObjectArrayHolder&& rhs) {
        ConstructFrom(rhs.elems);
        rhs.Clear();
    }

public:
   Array(Array const& rhs) : num{rhs.num} {
       ConstructFrom(rhs.elems);
   }

   Array(Array&& rhs) : num{rhs.num} {
       MoveFrom(rhs);
   }

protected:
   ElemType elems[MAX_NUM];
   SizeType num{};
};
```

move 构造，相对于 **copy 构造**，有两点重要的差异：

1. 如果不能进行直接拷贝，则只能将对方的元素一个个通过 `std::move` 移动过来；
2. 移动结束后，要将对方的数组清理掉（因为它的元素已经移动给我们了）。

需要注意的是，在 *move* 的阶段，我们重构了 `ConstructFrom` ，让它可以和 **copy 构造**复用。首先，我们将其改为了泛型函数，其模版参数 `U` 在 *copy* 的场景下，是 *const* 指针；而在 *move* 场景下，是 *non-const* 指针。其次，无论是 *copy* 还是 *move* ，我们都调用了 `std::move` ：

```
Trait::Emplace(elems[i], std::move(Trait::ToObject(array[i])));
```

在 *copy* 场景下，由于 *array* 是 *const* 的，`std::move(OBJ const&)` 的结果是 `OBJ const&&` 类型, 这毫无疑问会匹配到 `OBJ` 的拷贝构造。

在 *move* 场景下，*array* 是 *non-const* 的，`std::move(OBJ&)` 的结果是 `OBJ&&` ，如果 `OBJ` 提供了移动构造，则会毫不犹豫的与之匹配。否则，依然与 `OBJ` 拷贝构造匹配。无论如何都是我们期待的结果。

而在 `Clear` 阶段；我们再一次利用 **平凡性**进行了优化：如果 `OBJ` 是可平凡析构的，那就什么也不用做，只是简单的把 `num` 设置为 `0` 即可。否则，就老老实实一次将每个元素进行析构。

而对于 **copy/move 赋值**函数的实现，与构造类似，这里就不再赘述。

而这一切都是通过库自动判断完成，程序员完全不需要操心。

3.3 析构

下一个问题是，*Array<T,N>* 自身是否需要明确定义一个 **析构函数**?

首先，如果 *T* 本身是可平凡析构的，那么事实上我们在析构阶段什么都不用做（*num* 清零也没有意义）。

但如果 *T* 本身是不可平凡析构的，我们就应该在析构时老老实实将每个元素进行析构。否则，将是错误的程序行为。

因而，前者我们 **无需**提供析构函数，而后者则 **必须**提供。

当然也可以无脑全部都提供。但问题的麻烦在于，这不仅仅是增加一个析构函数那么简单。一旦我们明确为一个类定义了析构函数，它就肯定变成 **不可平凡析构**的。而这样的性质会一层层传播给它所有的持有者。本来大家可能都是 **可平凡析构**的，因而什么也不用做。现在倒好，每个它的宿主，无论直接还是间接的，都必须要在析构时被迫做工了。

sucks!!

所以，我们必须 **按需**为 *Array<T,N>* 提供 **析构函数**。C++ 20 可以通过 `requires` 语法直接决定一个函数的存在性。但在 C++ 17 时代，我们只能通过 **继承类**来解决这个问题。

```
template<typename OBJ, std::size_t MAX_NUM,
         bool = std::is_trivially_destructible_v<OBJ>>
struct ArrayExt : Array<OBJ, MAX_NUM> {
     using Parent = Array<OBJ, MAX_NUM>;
     using Parent::Parent;
 };

template<typename OBJ, std::size_t MAX_NUM>
struct ArrayExt<OBJ, MAX_NUM, false> : Array<OBJ, MAX_NUM> {
     using Parent = Array<OBJ, MAX_NUM>;
     using Parent::Parent;

     ~ArrayExt() { Parent::ClearContent(0); }
 };

 template<typename OBJ, std::size_t MAX_NUM>
 using ObjectArray = ArrayExt<OBJ, MAX_NUM>;
```

3.4 Rule Of Five

一旦我们为 *Array<T,N>* 根据需要明确提供了析构函数，按照 *C++* 的规则，**move 构造/赋值**也都不再自动生成默认函数（**copy 构造/赋值**则会依然默认生成）。

此时，即便 `T` 本身是 **可 move 构造**的，它也会转而匹配到 **copy 构造**，让本来可以通过 *move* 的得到的性能优化悄悄地丧失。

此时，就必须显式声明 **move 构造/赋值**（如果 `T` 支持的话）。哪怕声明为 `=default` 也必须显式声明。

可 *C++* 有另外一个规则，一旦你显式声明了 **move 构造**（和/或）**move 赋值**函数，那么 **copy 构造/赋值**的隐式声明也会消失，这等于是不再允许 **copy 构造/赋值**。如果这不是你的意图，则你必须也要显式声明 **copy 构造/赋值**（如果 `T` 支持的话）。

这种连环规则，其实就是 *rule of five* 由来的部分原因。一旦你明确定义一个，就必须同时考虑其它四个。

```
template<typename OBJ, std::size_t MAX_NUM,
         bool = std::is_trivially_destructible_v<OBJ>
struct ArrayExt : Array<OBJ, MAX_NUM> {
     using Parent = Array<OBJ, MAX_NUM>;
     using Parent::Parent;
 };

template<typename OBJ, std::size_t MAX_NUM>
struct ArrayExt<OBJ, MAX_NUM, false> : Array<OBJ, MAX_NUM> {
     using Parent = Array<OBJ, MAX_NUM>;
     using Parent::Parent;

     ArrayExt(ArrayExt const& rhs) = default;
     auto operator=(ArrayExt const& rhs) -> ArrayExt& = default;

     ArrayExt(ArrayExt&& rhs) = default;
     auto operator=(ArrayExt&& rhs) -> ArrayExt& = default;

     ~ArrayExt() { Parent::ClearContent(0); }
 };

 template<typename OBJ, std::size_t MAX_NUM>
 using ObjectArray = ArrayExt<OBJ, MAX_NUM>;
```

CHAPTER 4

ArrayView

除了我们自己需要明确定义和管理的数组之外，还有另外一族数组，它们属于已经定义好的 POD 数据结构中所包含的数组。

一个可以做出的假设是，由于这些数组都是 POD 的，因而必然是平凡的。

但是，为了操作的方便性，我们往往又会为这些 POD 数组加上一层 wrapper ，让它们可以在不增加任何开销的情况下，变为逻辑功能更加内聚的对象。

这些数组，根据是从外而来的消息，还是由内往外的消息，又可以分为只读和可写的。

结合这些需求，需要定义如下两种 ArrayView：（之所以被称做 *view* ，是因为 *view* 本身并不拥有数据，它只是查看或操作别人持有的数据。

4.1 ConstArrayView

```
template <typename OBJ, std::size_t MAX_NUM, typename ELEM = OBJ>
struct ConstArrayView {
   using SizeType = DeduceSizeType_T<MAX_NUM>;
   using ElemType = ELEM;
   using ViewTrait = typename ArrayViewTrait<OBJ, ELEM>::Type;

   ConstArrayView(OBJ const* array, std::size_t n)
      : elems(ViewTrait::ConstObjToElem(array))
      , num(std::min(MAX_NUM, n))
```

(续下页)

(接上页)

```
    {}

    ElemType const* elems;
    SizeType num;
};
```

4.2 ArrayView

```
template <typename OBJ, typename SIZE_TYPE, SIZE_TYPE MAX_NUM, typename ELEM = OBJ>
struct ArrayView {
    using ElemType = ELEM;
    using ViewTrait = typename ArrayViewTrait<OBJ, ELEM>::Type;

    ArrayView(OBJ* array, SIZE_TYPE& n)
        : elems(ViewTrait::ObjToElem(array))
        , num(n)
    {}

    ElemType* elems;
    SIZE_TYPE& num;
};
```

4.3 自动识别

除非你明确定义，否则，你不会直接使用 ConstArrayView 。你只需要使用 ArrayView ，然后依靠模版类的自动类型推演来自动决定。比如:

```
struct Foo {
    int array[10];
    uint16_t num;
};

Foo foo{{1,2,3,4}, 4};

ArrayView view1{foo.array, foo.num}; // ArrayView

view1.Append(5); // so you can append

Foo const& constRef = foo;
```

(续下页)

(接上页)

```
ArrayView view2{foo.array, foo.num}; // ConstArrayView

view2.Append(5); // compiling error.
```

ArrayView<T,N> 可以使用 ObjectArray<T, N> 的一切接口和算法。

CHAPTER 5

ScatteredArray

常规的数组，在空间上是连续的。如果一个常规数组的元素是可以增删的，那么在删除元素时，如果数组本身是无序的，则只需要将数组最后一个元素移动到被删除的 *slxot* ，以继续保持数组的连续性。这相对于有序数组要求将所有后续元素整体前移已经是一个重大的性能促进。

但即便只移动一个元素，如果元素较大，也是一种运行时代价。对于需要频繁增删的无序数组而言，将数组空间看作一种可分配的内存资源，每次需要增加元素时，从中分配一个空间的 *slot* ，删除时，只是将对应的 *slot* 释放回去。这样做的结果，有效的数组元素的空间将不再保证连续，而是散列在整个数组空间里。

这样的数组，称为 **散列数组** (*ScatteredArray*) 。

```
template<typename OBJ, std::size_t MAX_NUM>
struct ScatteredArrayHolder {
   using SizeType    = DeduceSizeType_t<MAX_NUM>;
   using ElemType    = DeduceElemType_T<OBJ>;

   ElemType elems[MAX_NUM];
   BitSet<MAX_NUM> occupied{};
};
```

`ScatteredArray` 和 `ObjectArray` 一样，都要考虑 copy/move 构造/赋值，是否需要提供非平凡析构函数的问题等等，其解决思路与 `ObjectArray` 一致，这里就不再赘述。

注意： 对于 *ScatteredArray* ， 你不能创建 *Slice* ， 跟不能创建 *ScopedView* , 因为基于 *ScatteredArray* 的性质，这些操作本身没有意义。

> 但是，其它的算法，比如 *Find*, *MinElem* , *ForEach*, *range-for*, *Sort*, *SortObject* 都是支持的。

ScatteredArray 的 *range-for* 以及 *ForEach* ，只会遍历所有有效的元素。

CHAPTER 6

有序数组

在很多场景下，我们需要数组元素是有序的。对于这类需求，有两种满足的方法：

1. 让数组保持无序；在需要时，临时进行（部分）排序；
2. 让数组本身保持有序；

排序需要时间。而数组排序时，大对象在数组空间内的拷贝/移动也代价高昂。因而，无论是临时性的排序，还是让数组本身保持有序，都有两种选择：

1. 对于小对象，可以让对象自身进行移动；
2. 对于大对象，则最好使用索引进行排序。

对于元素本身保持无序，需要时临时进行排序的数组，可以通过 `Sort` 或者 `SortObject` 进行排序；而对于自身需要维持有序的数组而言，本库提供了以简单 **插入排序**为算法的有序数组。

6.1 OrderedObjectArray

`OrderedObjectArray<T,N,COMPARE>` 是一种维持数组元素自身有序的一种数组。

模版的第三个参数 `COMPARE` ，决定了数组元素的顺序。其本身的默认实现为 `T::operator<` ，但你可以指定任何一种对比类型，比如：

```
struct Foo {
    int a;
    explicit Foo(int a) : a{a} {}
```

(续下页)

(接上页)

```
    ~Foo() { a = 10; }
};

auto FooLess = [](Foo const& l, Foo const& r) {
    return l.a > r.a;
};

OrderedObjectArray<Foo, 10, decltype(FooLess)> array;
```

6.2 IndexedOrderedArray

`IndexedOrderedArray<T,N,COMPARE>` 通过 **索引**来维持数组元素顺序，而数组元素本身是完全无序的，甚至是非连续的。这样保证了数组元素一旦在某个 *slot* 上创建，在整个生命周期存在期间，完全不会移动位置（无论是删除操作，还是 `Replace` 相关操作）。取而代之，变化和移动的，是数组元素的索引。

因而，`IndexedOrderedArray` 要比 `OrderedObjectArray` 耗费更多的空间（索引所需的空间，以及空间占用位图），但在维持有序方面，性能更高。

> **注意:** 事实上，`ObjectArray` 也有它的索引版本: `IndexedArray` 。它们都是将 `ScatteredArray` 当作一个数组空间分配器，并通过索引数组来保持 *Contiguous* 属性。
>
> 这样的做法，可以让需要频繁修改，比如进行 `Append`，`Erase`，`Sort`，`Rotate` 等操作，但同时对象的 `move` 成本又比较大的数组，具备性能优势。

6.3 自动选择

上述两种有序数组，提供了完全一致的接口。程序员拥有根据需要自主选择的权利。

但如果你有选择困难症，则可以使用 `OrderedArray<T,N,COMPARE>` ，它会根据一定的条件，自动从二者之中选择一个（而条件是可以集中进行统一维护）。

6.4 接口

对于有序数组而言，由于本身已经有序，因而不再提供 `Sort` 相关的任何接口，也不能创建 `SortObject` 。除此之外，它们拥有与普通数组完全一致的接口。

CHAPTER 7

Mixin

正如我们之前所见，*ObjectArray* 库里包含多种 *array* 以及 *view* 。

而相关的算法，一些在这些 *array/view* 中均可复用，而另外一些只能在少数 *array/view* 中可复用。有少数甚至只能在个别 *array/view* 中单独使用。如何解决这种变化多端的复用问题?

组合式设计是解决多种变化方向问题的利器。

而对于 *ObjectArray* 库，这种组合式设计的手段是 *mixin* 。

7.1 核心算法和扩展算法

对于很多算法而言，只要你实现了最核心的一个或几个，其它的都可以基于它进行实现。比如:

```
template<typename PRED>
auto FindIndex(PRED &&pred) const -> std::optional<SizeType>;

template<typename PRED>
auto Find(PRED &&pred) const -> ObjectType const&;
```

只要你实现基于谓词的 `FindIndex` 和 `Find` （事实上，如果不考虑 `Find` 与 `FindIndex` 在算法上可以给出不同实现的因素，二者只需要实现其中一个，另外一个即可以据它实现），你就可以扩展出下列算法:

```
auto FindIndex(ObjectType const& obj) const -> std::optional<SizeType> {
   return FindIndex([&](auto&& elem) { return elem = obj; });
}
```

(续下页)

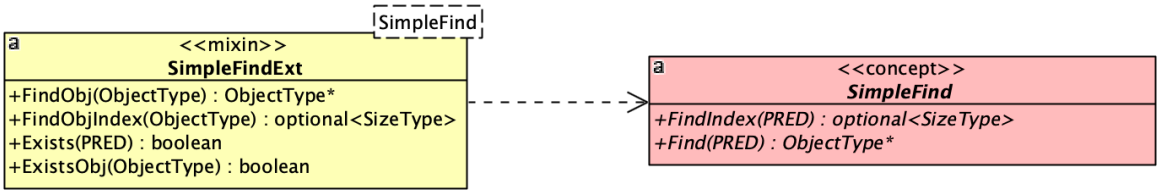
(接上页)

```
auto Find(ObjectType const& obj) const -> auto* {
    return Find([&](auto&& elem) { return elem == obj;});
}

template<typename PRED, __pReD_cHeCkEr>
auto Exists(PRED &&pred) const -> bool {
    return Find(std::forward<PRED>(pred)) != nullptr;
}
```

7.2 concept

我们随后以 SimpleFind 和 SimpleFindExt 这一族算法的 *mixin* 设计为例，来讨论整个 *mixin* 体系的设计。

首先，SimpleFind 和 SimpleFindExt 的之间的关系如下：



如果你仔细看这张图，会发现 SimpleFindExt 本身是一个 *mixin*，而它所依赖的是一个抽象的被称作 *concept* 的元素。

concept 不是 *class* 或 *template*，它是一个被 *C++ 20* 标准化了的概念。但事实上，即便这个概念没有被标准化，没有任何语法元素支持，它也是早就存在的设计概念。

简单说，*concept* 就相当于我们熟悉的用于 **运行时多态** 的 *interface*，或 **纯虚类**。但它不是用于 **运行时多态**，而是用于 **编译时多态**。因而它是一个静态接口，没有任何运行时代价。

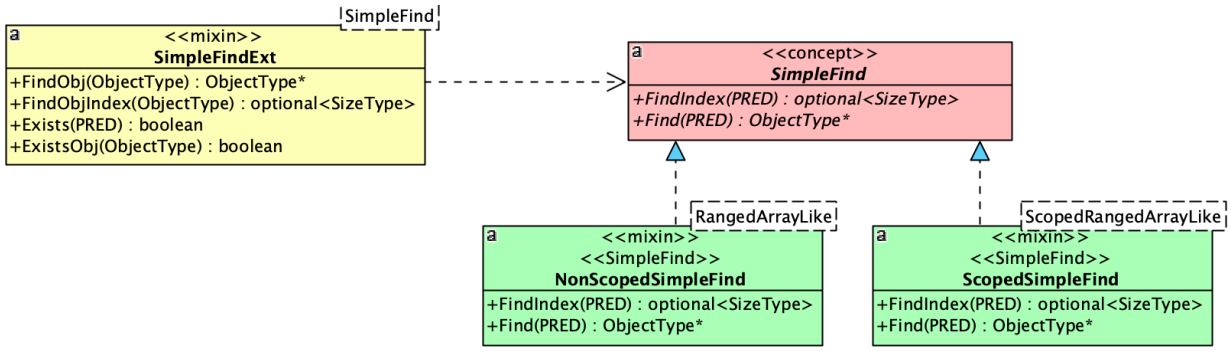
由于编译时多态是基于特征的（而不是函数接口，虽然特征也可以是函数接口），因而 *concept* 所描述的是一个模版对于它要依赖的类型的特征要求。所有可以用来实例化这个模版的类型都必须满足这些特征。

事实上，在 *C++ 20* 之前，这些要求也是存在的（否则就不可能编译通过），但由于没有语法元素来明确描述这些要求，因而一则对于代码阅读者无法通过一份接口规范来清晰地知道这些约束是什么；二则，当编译出错时，晦涩的出错信息让程序员很难快速定位出出错的原因。

无论如何，即便你仍在使用 *C++ 17*，甚至更早的编译器，你无法直接通过语法元素直接表达，但这并不妨碍从设计概念上，*concept* 依然是存在的。

现在回到我们的问题。为何 SimpleFindExt 依赖的是一个抽象的 *concept*，而不是一个具体的类?

像所有的多态一样，因为变化。对于 `SimpleFind` 这样一个 *concept* ，我们的库里至少有两种不同的需求，因而有两种不同的实现：



7.3 分类

在进一步讨论之前，我们先将我们的 *Array* 和 *View* 进行一下分类：

1. *NonScopedArrayLike*
    - *ScatteredArray*
    - *ScopedArrayView*
2. *ScopedArrayLike*
    - *ObjectArray*
    - *ArrayView* / *ConstArrayView*
    - *Slice*

之所以会有这两种不同的 `SimpleFind` ，是因为 *ScopedArrayLike* 而言，它们自身都有一个 `Scope` ，而对于 *NonScopedArrayLike* ，则没有 `Scope` ，而这两种的 `Find` 实现是不一样的。

但是一旦各自给出了最核心的 `Find` ，那么所有基于它们的扩展算法（`SimpleFindExt` ）则是完全一样的。

而这其中最有趣的一点是：`ScopedSimpleFind` 所依赖的 `ScopedFind` ，本身属于 *NonScopedArrayLike* 一族的正常接口，因为我们可以写出如下代码：

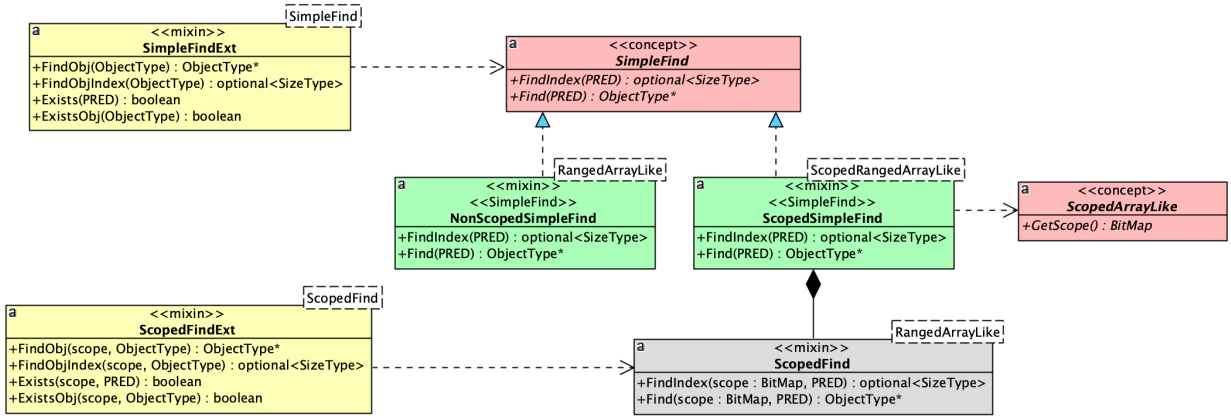
```
ObjectArray<int, 10> array;

array.Find(0xa5, [](auto&& item) { return item == 5 }); // 1st argument is a scope.

array.Scope(0xa5).Find([](auto&& item) { return item == 5 });
```

但是却不允许对 *ScopedArrayLike* 一族提供上述的接口（因为它们已经属于 *Scoped* ）。

因而，对于 *NonScopedArrayLike* 一族，存在 `ScopedFind` 以及它的扩展 `ScopedFindExt` 等一族接口。但 *ScopedArrayLike* 却没有这样的接口。

但有趣的地方也正在于此，*ScopedArrayLike* 一族的 `SimpleFind` 实现却可以通过复用 `ScopedFind` *mixin* 来实现。

它们的关系如下图所示：



7.3.1 NonScopedSimpleFind

而具体到 `NonScopedSimpleFind` 的实现，它也需要依赖一个 *concept*：`RangedArrayLike` 。因为 `Find` 与 `FindIndex` 的算法实现，仅仅需要依赖两类元素：

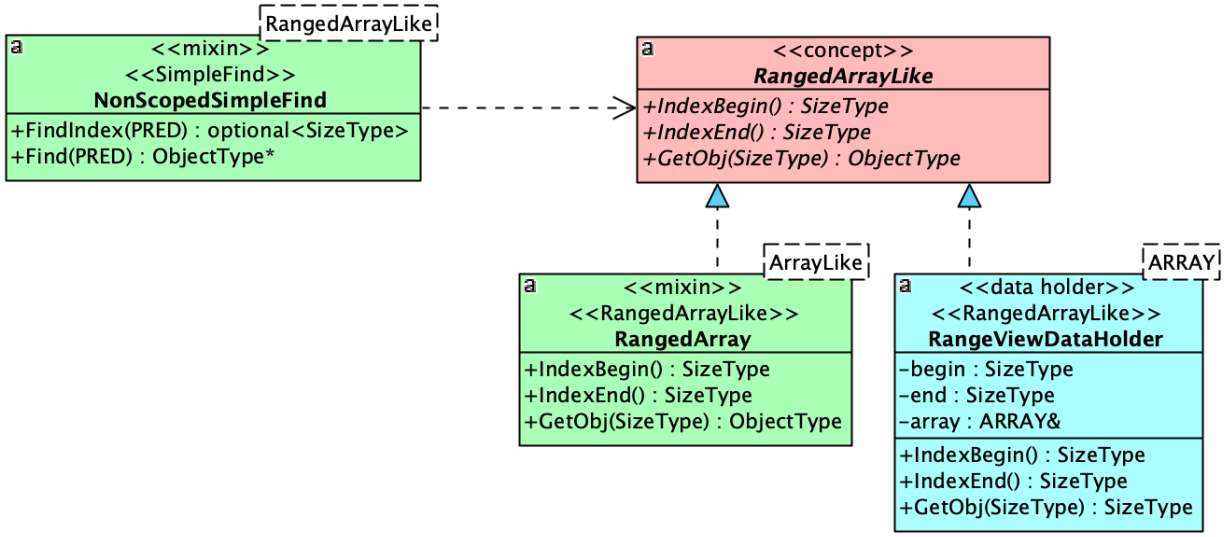
1. 搜索的 *Range*：*[begin, end)*，对应如下两个方法：
   - `IndexBegin() -> SizeType`
   - `IndexEnd() -> SizeType`
2. 每个索引位置的对象：
   - `GetObj(i) -> ObjectType const&`

而拥有这三个接口的概念为 `RangedArrayLike` 。而对于此 *concept* 的实现，可以分为两类：

1. `ObjectArray`, `ArrayView` ，它们的 *range* 是 *[0, num)*；
2. `Slice` 的 *range* 是切片创建时指定的 *range*；

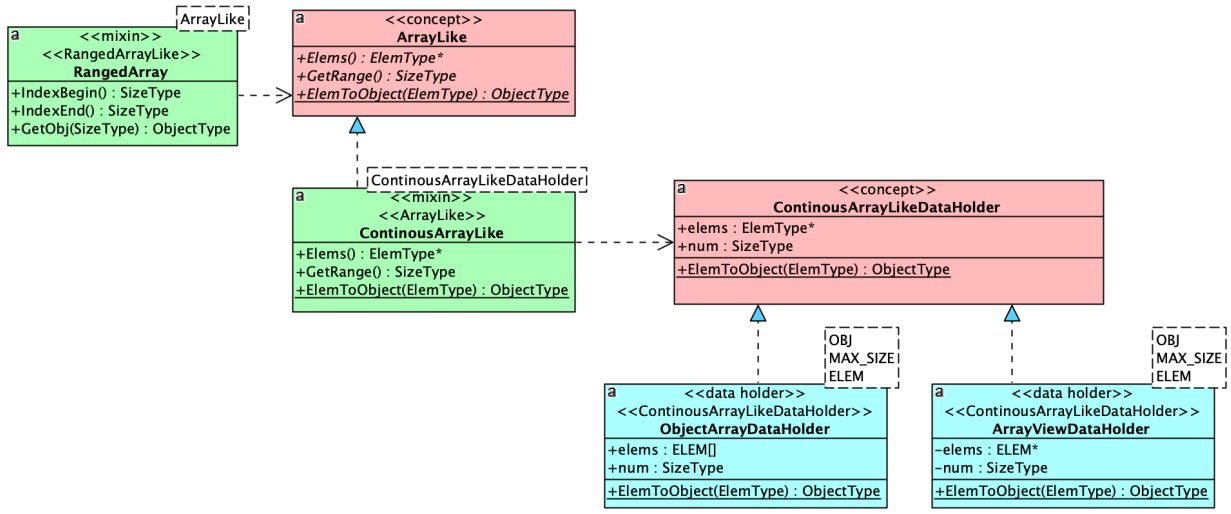
由于现在我们讨论的是 *NonScopedSimpleFind* ，因而 *ScopedArrayLike* 一族的因素暂时不予讨论。

因而，它们的关系如下：



图中，*RangedViewArrayHolder* 对应的就是 *slice* 的数据类。而 `RangedArray` 这个 *mixin*，则由 `ObjectArray` 和 `ArrayView` 组合。

如果只考虑 *NonScopedArrayLike*，那么 `RangedArray` 到具体的数据类的关系如下：



在这样的关系下，`RangeArray` 三个方法的实现如下：

```
template <_concept::ArrayLike T>
struct RangedArray {
   auto IndexBegin() const -> SizeType {
      return 0;
   }

   auto IndexEnd() const -> SizeType {
      return (ArrayLike const*)(this)->GetRange();
```

(续下页)

(接上页)

```
    }

    auto GetObj(SizeType n) const -> ObjectType const& {
        return ArrayLike::ElemToObject((ArrayLike const*)(this)->Elems()[n]);
    }
};
```

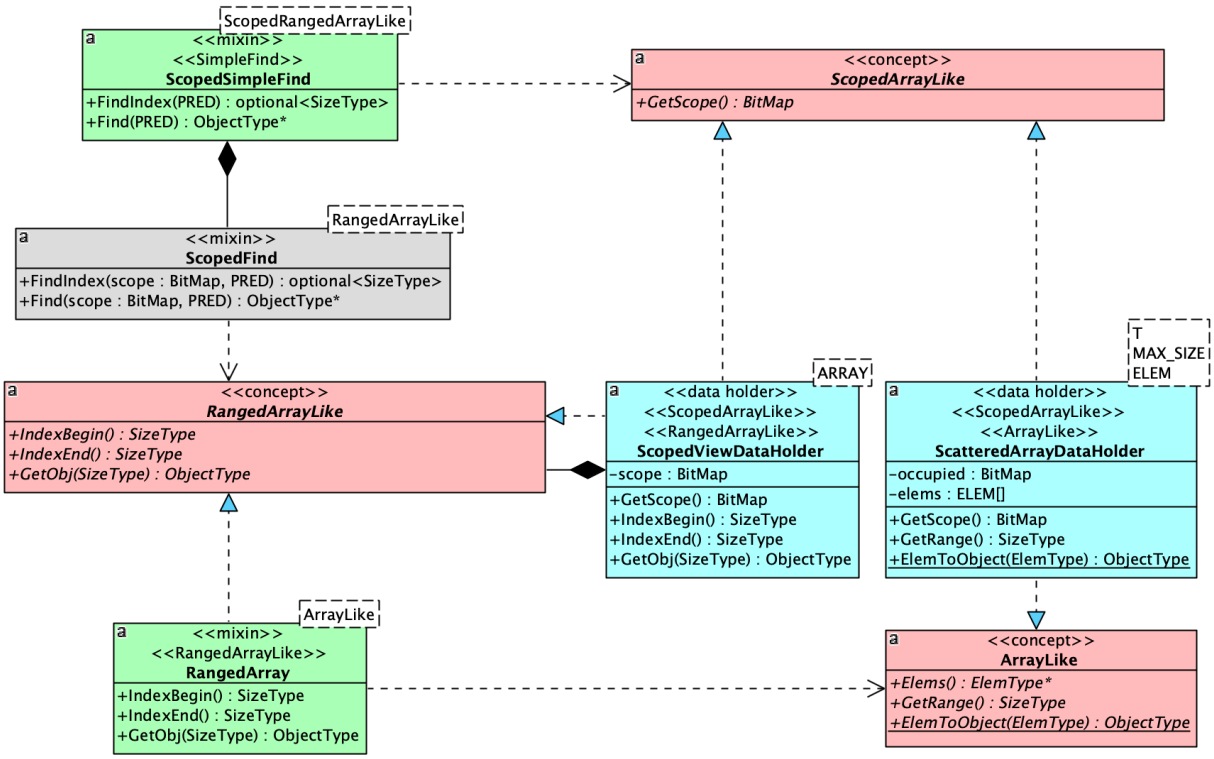
为何 IndexEnd() 调用的是 GetRange，而不是直接返回数组元素的个数：num？

这是因为，ArrayLike 这个 *concept* 不仅仅是 *NonScopedArrayLike* 才有的概念，*ScopedArrayLike*（ScatteredArray）同样有这个概念。ScatteredArray 的 GetRange 返回的不是 num (它没有这个属性)，而是 MAX_SIZE （代表它遍历的范围是整个数组空间)。

7.3.2 ScopedSimpleFind

ScopedSimpleFind 相对于 *NonScopedSimpleFind* ，复杂度稍微上升了一点，因为它需要一个额外的接口: GetScope 。而 Find 操作，只能在 *scope* 指定的范围内进行。

而 *ScopedView* 与 *ScatteredArray* 都属于此类。它们的关系如下:



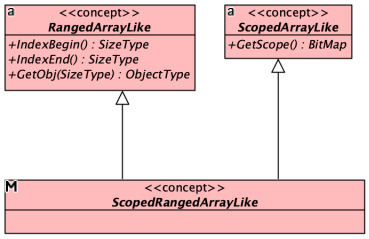
其中 *ScopedView* 聚合了一个提供了 *RangedArrayLike* 概念的对象，无论那个对象是一个 *ObjectArray* , *ArrayView* 还是一个 *Slice* ， *ScopedView* 都并不关心，只要它们都提供了 *RangeArrayLike* 概念所要求的接口。

而 *ScopeView* 只是通过转调它们的 *RangedArrayLike* 接口来让自己也成为一个满足 *RangedArrayLike* 概念的对象。

而 *ScatteredView* 则通过 *RangedArray mixin* 来让自己满足 *RangedArrayLike* 概念。

而两者都通过自己所持有的 `BitMap` 类型的数据来满足 *ScopedArrayAlike* 概念。

如果一个对象既满足 *RangedArrayLike* 概念，又满足 *ScopedArrayAlike* 概念，从语义上就满足了 *RangedArrayLike + ScopedArrayAlike* 概念。而 *C++ 20* 则通过 `RangedArrayLike && ScopedArrayAlike` 来表达这种概念上的组合关系。我们将这个组合后的概念，定义为一个新概念 *ScopedRangedArrayLike*：



7.4 mixin

到了现在，我们需要来谈一下什么叫 *mixin* 了。

简而言之，*mixin* 是个可以与对象进行组合的 *class/template class* 。它本身不应该有任何数据，因而可以通过 `std::is_empty_v<MIXIN>` 谓词断言。

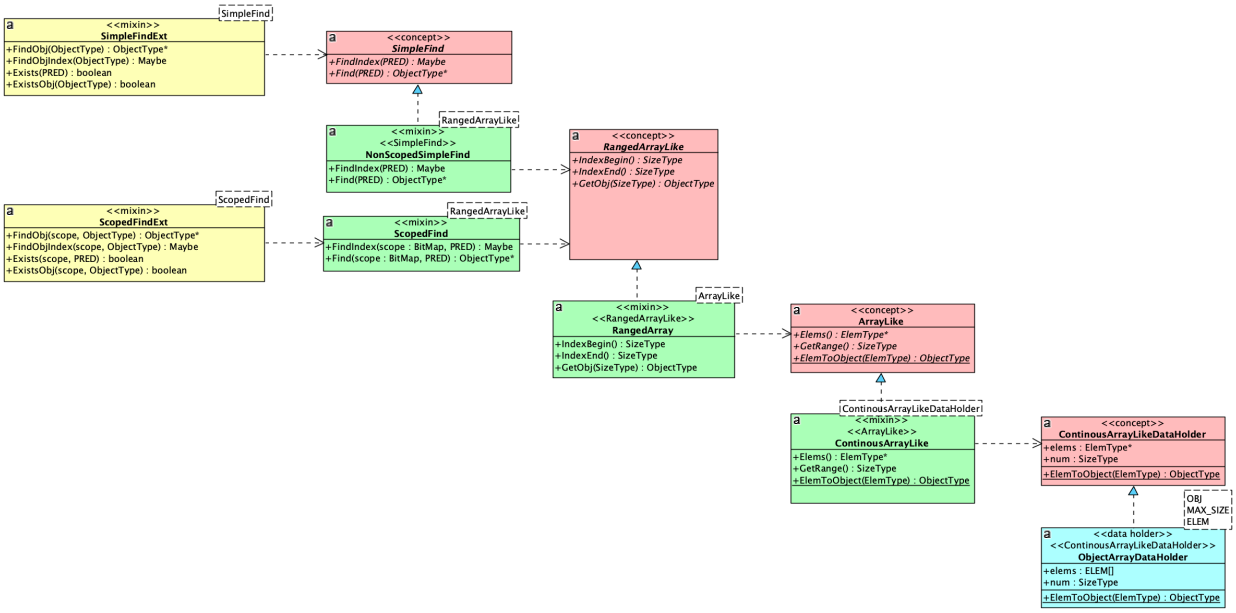
由于它本身没有任何数据，因而拼接在对象身上不会引起对象二进制结构（内存布局）的任何变化。所以，它们的 `this` 指针的位置与整个对象的 `this` 指针位置一致。

这样的 *mixin* 与其它语言比如 *scala* 所提供的 *trait* 概念上很相似。（*scala trait* 允许提供算法实现，并且其 *trait* 组合顺序与声明顺序一致）。

之所以使用 *mixin* 这样的概念，是为了让一个 *mixin* 所提供的实现，能够在不同的对象间方便的复用。

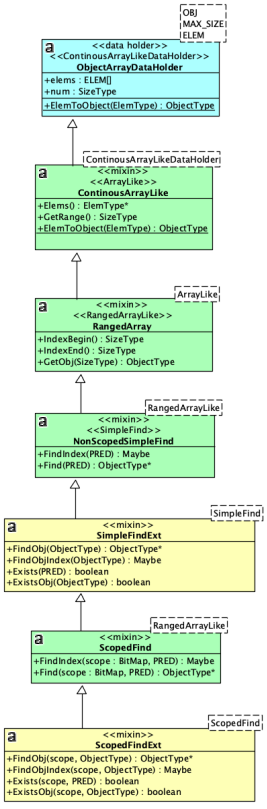
7.5 组合

现在到了我们进行组合的时候了。我们先来看看 `ObjectArray`：



从图中优美的线条我们看出，这是一个层层依赖的结构。

因而，对于 *mixin* 我们可以使用单线继承的方式来进行组合：



而其中每个 *mixin* 都是类似于下面的定义：

```
template <Concept T>
struct Mixin : T  {
   using Self = T;
   using Self::method; // import T::method
   using typename Self::Type // import T::Type
   // more imports

   // its own algorithm.
   auto DoSth() -> Bar {
      // ...
   }

   // more algorithms.
   // ...
};
```

我们总是将被依赖的 *mixin* 放在父类的位置；如果相互双方没有任何依赖关系（比如 `SimpleFindExt` 与 `ScopedFindExt`），那么它们在继承线上先后顺序也无所谓。

这样的组合方式，有一个明显的问题：子类对于父类同名函数的遮掩问题（比如 *SimpleFindExt* 与 *ScopedFindExt* 里都有 `Find`，虽然它们的参数列表并不相同）。

对于这样的问题，没有自动解决的办法，只能要么避免各个 *mixin* 间出现同名函数（需要全局知识）；要么放在下面的 *mixin* 如果知道放在上面的某个 *mixin* 存在与自己同名的函数，就负责明确地通过 `using` 来 *import*（这也需要全局知识）。

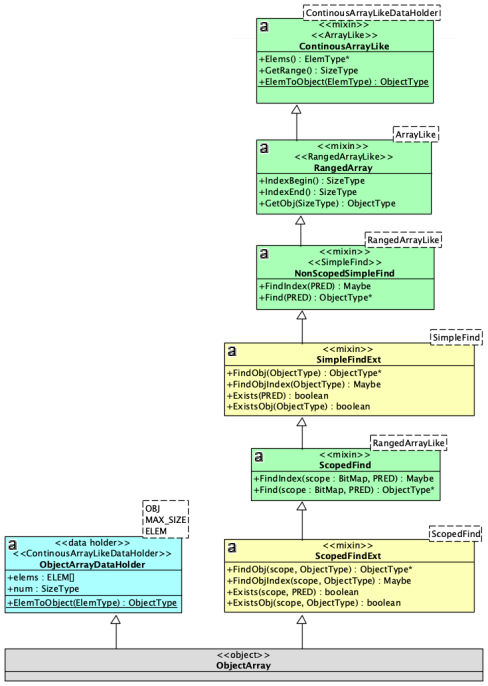
这当然是一个令人讨厌的地方，但对于所有 *mixin* 都在我们的控制之中（我们只是想通过分解为 *mixin* 达到复用目的），这一点并不会带来明显的设计和维护负担。

7.6 CRTP

使用这种方式进行组合的另外一个缺点是，这几乎总是导致数据类（*DataHolder*）被放在最顶层。

这本身没有任何问题，但却会导致 *debug* 时，如果需要查看数据，需要点开太多的层次（由于类层次很深）。每次层层点击打开的过程都让人精疲力尽，不厌其烦。

解决这种问题的办法是，我们将 *DataHolder* 从继承线的顶部移动到底部，变为下面的结构：



而这样的结构变化，让那些 *mixin* 如何访问 *DataHolder* 上的数据和方法编程了一个问题。

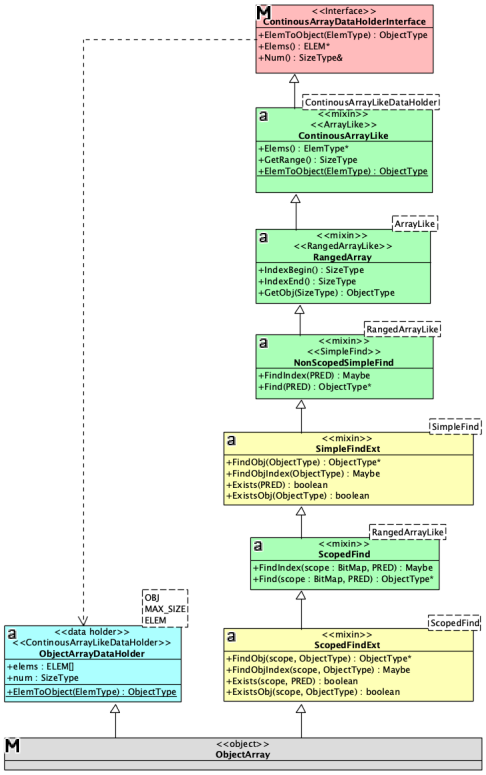
但 *C++* 范型有一个非常有趣的模式，叫做 *CRTP* （*Curiously Recurring Template Pattern* ）。即，一个作为父类，或者兄弟类的模版类，可以访问其子类的成员。

```
template <typename T>
struct Base {
    auto interface() -> void {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
    }

    static auto static_func() -> void {
        // ...
        T::static_sub_func();
        // ...
    }
};

struct Derived : Base<Derived> {
    void implementation();
    static void static_sub_func();
};
```

我们之前已经讲过，由于我们的 *mixin* 都是没有任何数据的模版类，它们的存在与否并不会影响整个对象的

内存布局。因而我们安全的将某个 *mixin* 的 `this` 指针强行转化为 *DataHolder* 的指针。

另外，由于我们只是把 *DataHolder* 从继承线上移出，*mixin* 们仍然保持了继承结构。我们我们只需要让 *DataHolder* 提供一个的替身：它提供了 *DataHolder* 希望对外暴露的接口，但本身又是一个类似于 *mixin* 的空类。我们将其成为 *DataHolder interface* 。如下图所示：



而 *DataHolder interface* 对 *DataHolder* 的访问，则是通过 *CRTP* 来完成：

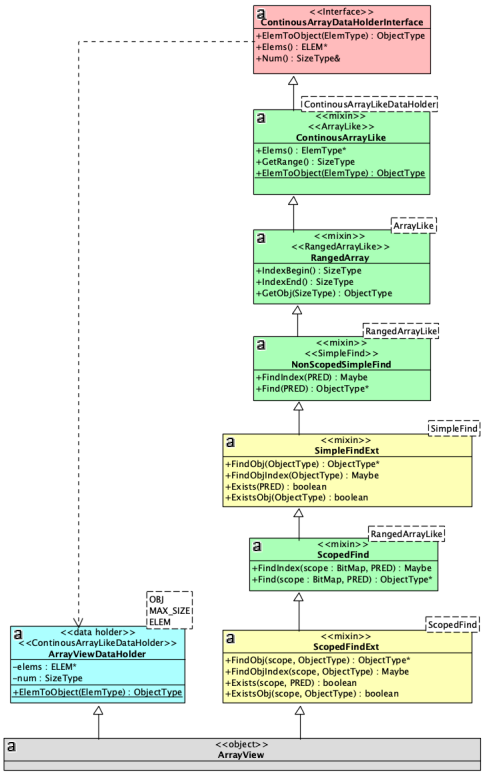
```
template <typename DATA_HOLDER>
class ContinuousArrayDataHolderInterface {
    auto This() const -> DATA_HOLDER const* {
        return reinterpret_cast<DATA_HOLDER const*>(this);
    }
    auto This() -> DATA_HOLDER* {
        return reinterpret_cast<DATA_HOLDER*>(this);
    }
public:
    using SizeType = typename DATA_HOLDER::SizeType;
    using ElemType = typename DATA_HOLDER::ElemType;

    auto Num() -> SizeType& {
       return This()->num;
    }
    auto Elems() -> ElemType* {
```

(续下页)

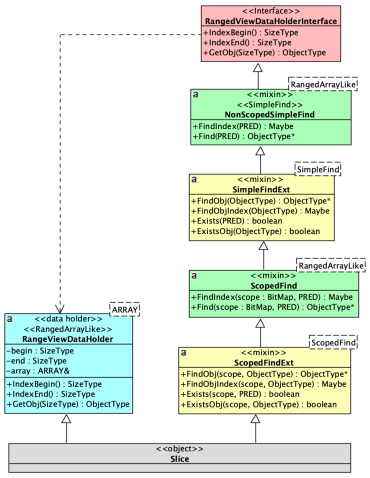
(接上页)

```
        return This()->elems;
    }
    static auto ElemToObject(ElemType& elem) -> ObjectType& {
        return DATA_HOLDER::ElemToObject(elem);
    }
};
```

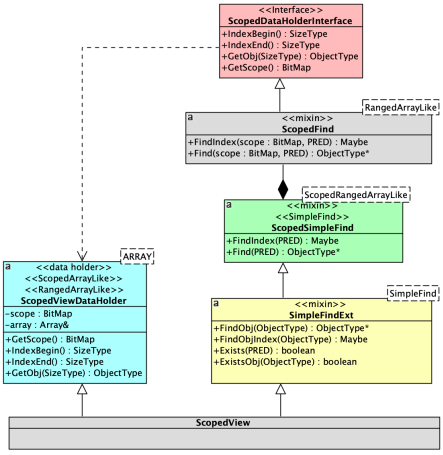
在组合了 *ObjectArray* 之后，我们发现 *ArrayView* 的组合方式与 *ObjectArray* 是一致的，除了 *DataHolder* 不同之外：



而 *Slice* 需要的 *mixin* 要比 *ObjectArray* 少一些，因而它比较简单：

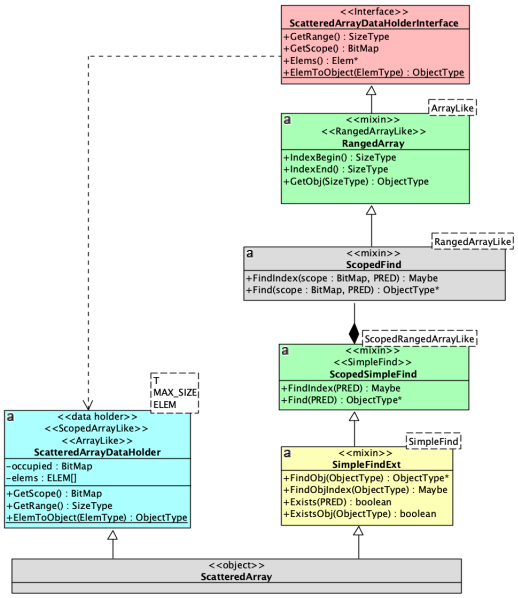


ScopedView 是一个不同的物种，如下图：



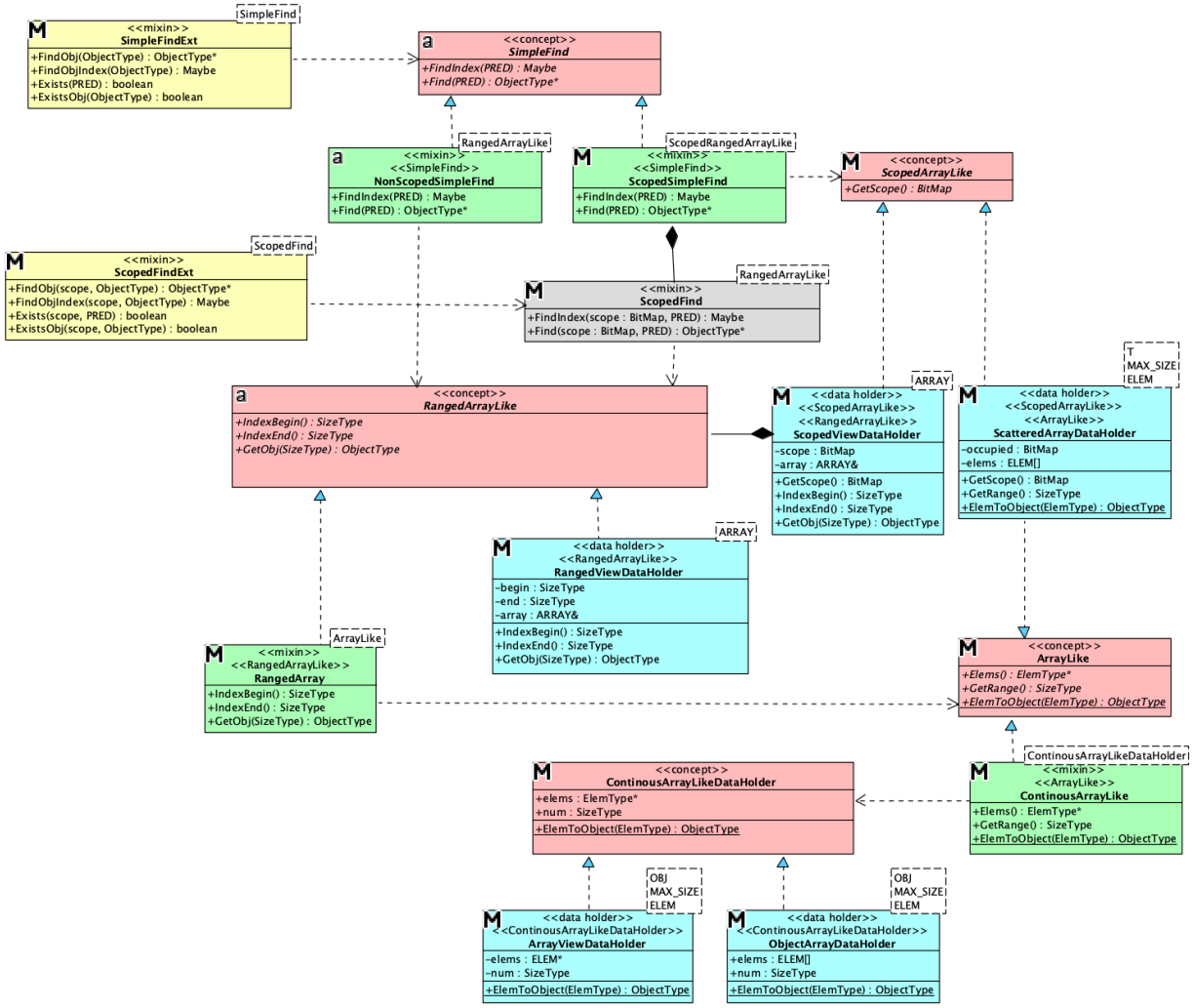
正如我们之前所讨论的，它本身不再提供 *ScopedFind* 接口，但却会利用 *ScopedFind* 来实现 *SimpleFind* ，因而即便从具体实现上这依然是通过继承来实现，但在图中我们用组合关系来表达。

而 *ScatteredArray* 则与 *ScopedView* 属于同一类：



实际上，每一种对象最终拼合的 *mixin* 要比这里举例的要多得多。但它们依据的方法是一致的。这里就不再赘述。

最后给出与 *Find* 有关的全景视图，仔细查看它们的方法，会有助于理解为何会存在这些 *mixin* 以及为何它们之间是那样的关系。



7.7 可见性

我们的每一种 *array/view* 都组合了多个 *mixin*，但并不是所有的 *mixin* 所提供的接口都应该是用户可见的。

所以我们就面临一个问题：如何让用户仅仅可以访问我们允许他访问的接口？

一种最直接的办法是，把所有的 *mixin* 组合都首先设置为 `protected` 或 `private`，然后在最下面的类通过 `using` 指令来暴露我们想暴露的接口。

这样的方法简单直接，易于控制。但缺点也是显而易见的：

首先，我们需要手工 `using` 每一个方法。这不仅会导致在不同的 *array/view* 上重复的代码，并且还经常在维护过程中会导致遗漏。

更重要的是，当我们手动 `using` 了之后，就没办法自动禁止掉一些接口。这一点我们会在后面讲到。

因而，我们希望能有一种自动措施：所有被声明为 `public` 的 *mixin*，其 `public` 接口会自动暴露给用户；否则，将自动隐藏。

为了达到这一目的，一种方法是把所有 *mixin* 全部水平继承，这样就可以精准的控制每一个 *mixin* 的可见性。

但水平继承的缺点是：首先，你很难有一种不带来负担的方式，精准的指明 *mixin* 间的依赖关系；其次，每一个 *mixin* 的实现都要依赖 *CRTP* ，这回导致 *mixin* 的编写工作量增加；另外，虽然并不是特别重要，但由于水平铺开的所导致的类型名字，比垂直继承所导致的类型名字要显著增长，一旦编译错误，满屏的类型信息会导致排错时间增长。

因而，我们还是希望回到垂直继承的路上。垂直继承的特点是，你一旦 *private* 或者 *protected* 继承了某一个 *mixin* ，那么所有之前的 *mixin* 都变得让用户不可访问。

我们无力改变这一点，但我们可以通过把所有不对用户可见的 *mixin* 放在前面即可解决。

那么紧接着的问题是：如何指明 **不可访问**与 **可访问** *mixin* 的边界？

当然这有很多种方案。但其中最好的方案一定是完全正交的方案：即完全不用修改任何 *mixin* 代码，仅仅靠简单的独立声明就可以做到。

于是有了这样的方案：

```
template<typename T>
struct ___public_mixin_delimiter___ : protected T {};
```

就这么一个简单的仿 *mixin* ，放在整个 *mixin* 列表里即可。比如：

```
using SliceMixins = mixin::Mixins<
        mixin::RangedArrayLike,
        mixin::ObjectIndex,
        mixin::ArrayElemVisit,
        mixin::ScopedFind,
        mixin::ScopedForEach,
        mixin::ViewedArray,
        mixin::___public_mixin_delimiter___, // 分界线
        mixin::IndexedRefAccessor,
        mixin::ByIndexAccessor,
        mixin::RangedElemCount,
        mixin::IterableArrayLike,
        mixin::NonScopedSimpleFind,
        // more mixins ...
        mixin::ArraySortExt
    >
```

有它所画出的分界线，之上的全是对用户不可见的内部 *mixin* ，其后则是对用户可见的 *public mixin* ，而其它 *mixin* 对此一共所知。

7.8 存在性

还有另外一个更严重的问题是 **存在性**问题：一些接口，尤其是 *non-const* 的修改相关的接口，当 *array/view* 本身的内部数据是不可修改时（但 *array* 本身是 *non-const* 的），那些修改对象状态相关的接口就不应该存在。比如：

```
ObjectArray<int const, 10> array;
// array itself is non-const, but its element type is const.
```

对于这个定义中的 *array* 本身不是 `const` 对象，按照 *C++* 语义，所有的 *non-const* 接口它都可以调用。但是，由于其内部的 *array* 是 *const* 的，事实上真的修改它们又是不允许的。这样的代码最终必然会导致编译错误。

所以，最好的方法是：一旦发现 `ObjectArray<T, N>` 内部的数据是不可修改的，那么所有 *non-const* 接口都应该自动消失。对于我们基于 *mixin* 组合的设计而言，则意味着那些相关的 *mixin* 都自动消失。

但如何做到？是否像上一节所讨论的 **可访问性**一样，存在一个非侵入的、完全正交的声明式方案？

答案是 *YES*：

```
template<typename T>
struct ___mutable_mixin_delimiter___ final : T {
    constexpr static bool IS_CONST = T::CONST;
};
```

其中 `T`，即继承线上，之前的任何一个 *mixin* 有义务来说明自己所持的数组是否是一个 *const* 的。

而这个仿 *mixin* 的诀窍则在于将自己设为 `final` 的。其语义为：我不再允许任何继承，我就是最后一个 *mixin* 。

而 *mixin composer* 一旦发现某个 *mixin* 是 `final` 的，则查看其给出的常量 *IS_CONST*：如果为真，则放弃组合后面所有的 *mixin*；如果为 `false`，则继续组合后面的 *mixin* 。但无论是哪一种情况，这个仿 *mixin* 都会被丢弃（否则 `final` 会导致继承真的被禁止了），它的存在只是给 *mixin composer* 一个指示而已。一旦职责完成，就不再有存在的必要性。

所以，对于任何一个可读写的 *array/view*，它的 *mixin* 列表都会存在这两个 *delimiter* 。比如：

```
using SliceMixins = mixin::Mixins<
        mixin::RangedArrayLike,
        mixin::ObjectIndex,
        mixin::ArrayElemVisit,
        mixin::ScopedFind,
        mixin::ScopedForEach,
        mixin::ViewedArray,
        mixin::___public_mixin_delimiter___,
        mixin::IndexedRefAccessor,
        mixin::ByIndexAccessor,
        mixin::RangedElemCount,
```

(续下页)

(接上页)

```
        mixin::IterableArrayLike,
        mixin::NonScopedSimpleFind,
        mixin::SimpleFindExt,
        mixin::SimpleForEach,
        mixin::SimpleForEachExt,
        mixin::RValueScopedViewFactory,
        mixin::RValueIndexedViewFactory,
        mixin::RValueSortViewFactory,
        mixin::ScopedFindExt,
        mixin::ScopedForEachExt,
        mixin::SimpleMinElem,
        mixin::SimpleMinElemExt,
        mixin::ScopedMinElemExt,
        mixin::___mutable_mixin_delimiter___,
        mixin::ViewAppend,
        mixin::AppendExt,
        mixin::RangedReplace,
        mixin::ReplaceExt,
        mixin::RValueArraySort,
        mixin::ArraySortExt>;
```

简洁，完美，搞定!!!

CHAPTER 8

性能

我们不直接给出性能测试结果。而是对照使用 `ObjectArray` 库，与不使用时亲自编写相关操作，生成的指令对比。

- 平台: *x64*
- 编译器：*clang 12.0.5*
- 编译优化选项：*-Os*

8.1 ForEach

假设我们定义一个如下的 *array*：

```
struct Foo {
   int a;
   int b;
};

struct Bar {
   Foo foo[7];
   uint8_t num;
};
```

当我们按照常规去遍历一个数组时，如下是我们的常规做法：

```
auto g1(Bar const& array, uint8_t n) -> uint32_t {
   auto sum = 0;
   auto num = std::min(array.num, (uint8_t)7);
   for(auto i=0; i<num; i++) {
       sum += array.foo[i].a;
   }
   return sum;
}
```

对应生成的机器指令如下：

```
0000000100003f0f <__Z2g1RK3Barh>:
100003f0f: 55                          pushq   %rbp
100003f10: 48 89 e5                    movq    %rsp, %rbp
100003f13: 8b 4f 38                    movl    56(%rdi), %ecx
100003f16: 80 f9 07                    cmpb    $7, %cl
100003f19: b8 07 00 00 00              movl    $7, %eax
100003f1e: 0f 42 c1                    cmovbl  %ecx, %eax
100003f21: 84 c0                       testb   %al, %al
100003f23: 74 14                       je      0x100003f39 <__Z2g1RK3Barh+0x2a>
100003f25: 0f b6 c8                    movzbl  %al, %ecx
100003f28: 31 d2                       xorl    %edx, %edx
100003f2a: 31 c0                       xorl    %eax, %eax
100003f2c: 03 04 d7                    addl    (%rdi,%rdx,8), %eax
100003f2f: 48 ff c2                    incq    %rdx
100003f32: 48 39 d1                    cmpq    %rdx, %rcx
100003f35: 75 f5                       jne     0x100003f2c <__Z2g1RK3Barh+0x1d>
100003f37: eb 02                       jmp     0x100003f3b <__Z2g1RK3Barh+0x2c>
100003f39: 31 c0                       xorl    %eax, %eax
100003f3b: 5d                          popq    %rbp
100003f3c: c3                          retq
```

如果我们使用 `ArrayView`，同样的功能，我们可以这样写：

```
auto g2(Bar const& array, uint8_t n) -> uint32_t {
    auto sum = 0;
    ArrayView{array.foo, array.num}.ForEach([&](auto&& ref) {
       sum += ref.a;
    });
    return sum;
}
```

注意，`ArrayView` 在构造时，同样会进行 `std::min(num, MAX)` 的操作，所以程序员不需要亲自写，却同样可以保证访问范围是安全的。

而生成的机器指令如下：

```
0000000100003f3d <__Z2g2RK3Barh>:
100003f3d: 55                          pushq   %rbp
100003f3e: 48 89 e5                    movq    %rsp, %rbp
100003f41: 8b 4f 38                    movl    56(%rdi), %ecx
100003f44: 80 f9 07                    cmpb    $7, %cl
100003f47: b8 07 00 00 00              movl    $7, %eax
100003f4c: 0f 42 c1                    cmovbl  %ecx, %eax
100003f4f: 84 c0                       testb   %al, %al
100003f51: 74 14                       je      0x100003f67 <__Z2g2RK3Barh+0x2a>
100003f53: 0f b6 c8                    movzbl  %al, %ecx
100003f56: 31 c0                       xorl    %eax, %eax
100003f58: 31 d2                       xorl    %edx, %edx
100003f5a: 03 04 d7                    addl    (%rdi,%rdx,8), %eax
100003f5d: 48 ff c2                    incq    %rdx
100003f60: 48 39 d1                    cmpq    %rdx, %rcx
100003f63: 75 f5                       jne     0x100003f5a <__Z2g2RK3Barh+0x1d>
100003f65: eb 02                       jmp     0x100003f69 <__Z2g2RK3Barh+0x2c>
100003f67: 31 c0                       xorl    %eax, %eax
100003f69: 5d                          popq    %rbp
100003f6a: c3                          retq
```

你可以看得出来，使用 `lambda` 并没有带来任何开销。甚至可以生成更加精简的指令。

当然，如果我们可以使用 *range-for* , 会觉得舒适得多：

```
auto g3(Bar const& array, uint8_t n) -> uint32_t {
   auto sum = 0;
   for(auto&& ref : ArrayView{array.foo, array.num}) {
       sum += ref.a;
   }
   return sum;
}
```

对应生成的指令如下：

```
0000000100003f6b <__Z2g3RK3Barh>:
100003f6b: 55                          pushq   %rbp
100003f6c: 48 89 e5                    movq    %rsp, %rbp
100003f6f: 8b 4f 38                    movl    56(%rdi), %ecx
100003f72: 80 f9 07                    cmpb    $7, %cl
100003f75: b8 07 00 00 00              movl    $7, %eax
100003f7a: 0f 42 c1                    cmovbl  %ecx, %eax
100003f7d: 84 c0                       testb   %al, %al
100003f7f: 74 19                       je      0x100003f9a <__Z2g3RK3Barh+0x2f>
100003f81: 0f b6 c8                    movzbl  %al, %ecx
100003f84: 48 c1 e1 03                 shlq    $3, %rcx
100003f88: 31 d2                       xorl    %edx, %edx
100003f8a: 31 c0                       xorl    %eax, %eax
100003f8c: 03 04 17                    addl    (%rdi,%rdx), %eax
100003f8f: 48 83 c2 08                 addq    $8, %rdx
100003f93: 48 39 d1                    cmpq    %rdx, %rcx
100003f96: 75 f4                       jne     0x100003f8c <__Z2g3RK3Barh+0x21>
100003f98: eb 02                       jmp     0x100003f9c <__Z2g3RK3Barh+0x31>
100003f9a: 31 c0                       xorl    %eax, %eax
100003f9c: 5d                          popq    %rbp
100003f9d: c3                          retq
```

8.2 Find

同样的，我们先给出一个普通数组的常规做法：

```
auto g1(Bar const& array, uint8_t n) -> uint32_t {
   auto num = std::min(array.num, (uint8_t)7);
   for(auto i=0; i<num; i++) {
      if(array.foo[i].a == 2) return i;
    }
    return 0xFF;
}
```

```
0000000100003f1c <__Z2g1RK3Barh>:
100003f1c: 55                          pushq   %rbp
100003f1d: 48 89 e5                    movq    %rsp, %rbp
100003f20: 8b 47 38                    movl    56(%rdi), %eax
100003f23: 3c 07                       cmpb    $7, %al
100003f25: b9 07 00 00 00              movl    $7, %ecx
100003f2a: 0f 42 c8                    cmovbl  %eax, %ecx
100003f2d: b8 ff 00 00 00              movl    $255, %eax
100003f32: 84 c9                       testb   %cl, %cl
100003f34: 74 17                       je      0x100003f4d <__Z2g1RK3Barh+0x31>
100003f36: 0f b6 d1                    movzbl  %cl, %edx
100003f39: 31 c9                       xorl    %ecx, %ecx
100003f3b: 83 3c cf 02                 cmpl    $2, (%rdi,%rcx,8)
100003f3f: 74 0a                       je      0x100003f4b <__Z2g1RK3Barh+0x2f>
100003f41: 48 ff c1                    incq    %rcx
100003f44: 48 39 ca                    cmpq    %rcx, %rdx
100003f47: 75 f2                       jne     0x100003f3b <__Z2g1RK3Barh+0x1f>
100003f49: eb 02                       jmp     0x100003f4d <__Z2g1RK3Barh+0x31>
100003f4b: 89 c8                       movl    %ecx, %eax
100003f4d: 5d                          popq    %rbp
100003f4e: c3                          retq
```

```
auto g2(Bar const& array, uint8_t n) -> IntOpt<uint8_t> {
   return ArrayView{array.foo, array.num}.FindIndex([&](auto&& ref) {
      return ref.a == 2;
   });
}
```

需要注意的是，这里的算法返回值，与之前算法的返回值不一样，前面按照 *C* 语言使用者的常规做法，直接用 `0xFF` 作为非法值。

而后者的做法则是使用接口与 `std::optional` 完全一致的，针对整型的 *optional* 语义的实现。

```
0000000100003f4f <__Z2g2RK3Barh>:
100003f4f: 55                          pushq   %rbp
100003f50: 48 89 e5                    movq    %rsp, %rbp
100003f53: 8b 47 38                    movl    56(%rdi), %eax
100003f56: 3c 07                       cmpb    $7, %al
100003f58: b9 07 00 00 00              movl    $7, %ecx
100003f5d: 0f 42 c8                    cmovbl  %eax, %ecx
100003f60: b0 ff                       movb    $-1, %al
100003f62: 84 c9                       testb   %cl, %cl
100003f64: 74 17                       je      0x100003f7d <__Z2g2RK3Barh+0x2e>
100003f66: 0f b6 d1                    movzbl  %cl, %edx
100003f69: 31 c9                       xorl    %ecx, %ecx
100003f6b: 83 3c cf 02                 cmpl    $2, (%rdi,%rcx,8)
100003f6f: 74 0a                       je      0x100003f7b <__Z2g2RK3Barh+0x2c>
100003f71: 48 ff c1                    incq    %rcx
100003f74: 48 39 ca                    cmpq    %rcx, %rdx
100003f77: 75 f2                       jne     0x100003f6b <__Z2g2RK3Barh+0x1c>
100003f79: eb 02                       jmp     0x100003f7d <__Z2g2RK3Barh+0x2e>
100003f7b: 89 c8                       movl    %ecx, %eax
100003f7d: 5d                          popq    %rbp
100003f7e: c3                          retq
```

仔细对比，会发现二者生成的指令几乎完全相同。唯一的差别是两个立即数：一个是 255 ，一个是 -1 。事实上，对于 *8-bit* 的整数而言，它们是等价的。

因而，我们可以再次得出结论：至少短小的、可内联的 *lamdba* 本身并不会带来任何性能损失。

```
auto g3(Bar const& array, uint8_t n) -> uint32_t {
   for(auto&& [ref, i] : ArrayView{array.foo, array.num}.WithIndex()) {
       if(ref.a == 2) return i;
   }
   return 0xFF;
}
```

```
0000000100003f7f <__Z2g3RK3Barh>:
100003f7f: 55                          pushq   %rbp
100003f80: 48 89 e5                    movq    %rsp, %rbp
100003f83: 8b 47 38                    movl    56(%rdi), %eax
100003f86: 3c 07                       cmpb    $7, %al
100003f88: b9 07 00 00 00              movl    $7, %ecx
100003f8d: 0f 42 c8                    cmovbl  %eax, %ecx
100003f90: b8 ff 00 00 00              movl    $255, %eax
100003f95: 84 c9                       testb   %cl, %cl
100003f97: 74 1d                       je      0x100003fb6 <__Z2g3RK3Barh+0x37>
100003f99: 0f b6 c9                    movzbl  %cl, %ecx
100003f9c: 48 c1 e1 03                 shlq    $3, %rcx
100003fa0: 31 d2                       xorl    %edx, %edx
100003fa2: 83 3c d7 02                 cmpl    $2, (%rdi,%rdx,8)
100003fa6: 74 0b                       je      0x100003fb3 <__Z2g3RK3Barh+0x34>
100003fa8: 48 ff c2                    incq    %rdx
100003fab: 48 83 c1 f8                 addq    $-8, %rcx
100003faf: 75 f1                       jne     0x100003fa2 <__Z2g3RK3Barh+0x23>
100003fb1: eb 03                       jmp     0x100003fb6 <__Z2g3RK3Barh+0x37>
100003fb3: 0f b6 c2                    movzbl  %dl, %eax
100003fb6: 5d                          popq    %rbp
100003fb7: c3                          retq
```

通过仔细对比，会发现第三种写法，比前两种多了一条 `shlq` 指令，其它地方虽然看似也有少许差异，但本质上是相同的。而多出的这条指令，并不在循环路径上。

8.3 MinElemIndex

下面我们来看看当查找一个最小元素的索引算法：

```
auto f1(Bar const& array) -> uint8_t {
   if(array.num == 0) return 0xFF;

   uint8_t min = 0;
   auto n = std::min(array.num, (uint8_t)10);
   for(auto i=1; i<n; i++) {
      if(array.foo[i].a < array.foo[min].a) {
```

(续下页)

(接上页)

```
            min = i;
        }
    }
    return min;
}
```

同样的，这里使用 0xFF 当作非法索引值。

```
0000000100003ec7 <__Z2f1RK3Bar>:
100003ec7: 55                           pushq   %rbp
100003ec8: 48 89 e5                     movq    %rsp, %rbp
100003ecb: 8a 47 38                     movb    56(%rdi), %al
100003ece: 84 c0                        testb   %al, %al
100003ed0: 74 35                        je      0x100003f07 <__Z2f1RK3Bar+0x40>
100003ed2: 3c 0a                        cmpb    $10, %al
100003ed4: 0f b6 c8                     movzbl  %al, %ecx
100003ed7: b8 0a 00 00 00               movl    $10, %eax
100003edc: 0f 42 c1                     cmovbl  %ecx, %eax
100003edf: 3c 02                        cmpb    $2, %al
100003ee1: 72 28                        jb      0x100003f0b <__Z2f1RK3Bar+0x44>
100003ee3: 44 0f b6 c0                  movzbl  %al, %r8d
100003ee7: ba 01 00 00 00               movl    $1, %edx
100003eec: 31 c0                        xorl    %eax, %eax
100003eee: 8b 0c d7                     movl    (%rdi,%rdx,8), %ecx
100003ef1: 0f b6 f0                     movzbl  %al, %esi
100003ef4: 89 d0                        movl    %edx, %eax
100003ef6: 3b 0c f7                     cmpl    (%rdi,%rsi,8), %ecx
100003ef9: 72 02                        jb      0x100003efd <__Z2f1RK3Bar+0x36>
100003efb: 89 f0                        movl    %esi, %eax
100003efd: 48 ff c2                     incq    %rdx
100003f00: 49 39 d0                     cmpq    %rdx, %r8
100003f03: 75 e9                        jne     0x100003eee <__Z2f1RK3Bar+0x27>
100003f05: eb 06                        jmp     0x100003f0d <__Z2f1RK3Bar+0x46>
100003f07: b0 ff                        movb    $-1, %al
100003f09: eb 02                        jmp     0x100003f0d <__Z2f1RK3Bar+0x46>
100003f0b: 31 c0                        xorl    %eax, %eax
100003f0d: 0f b6 c0                     movzbl  %al, %eax
100003f10: 5d                           popq    %rbp
100003f11: c3                           retq
```

然后我们再来看看直接使用标准库的实现：

```
auto f2(Bar const& bar) -> uint8_t {
    auto num = std::min(bar.num, (uint8_t)7);
    auto found = std::min_element(bar.foo, bar.foo + num,
        [](auto &&l, auto&& r) { return l.a < r.a; });
    return found == bar.foo + num ? 0xFF : found - bar.foo;
}
```

```
0000000100003f12 <__Z2f2RK3Bar>:
100003f12: 55                          pushq   %rbp
100003f13: 48 89 e5                    movq    %rsp, %rbp
100003f16: 8b 47 38                    movl    56(%rdi), %eax
100003f19: 3c 07                       cmpb    $7, %al
100003f1b: b9 07 00 00 00              movl    $7, %ecx
100003f20: 0f 42 c8                    cmovbl  %eax, %ecx
100003f23: 0f b6 d1                    movzbl  %cl, %edx
100003f26: 4c 8d 04 d7                 leaq    (%rdi,%rdx,8), %r8
100003f2a: 49 89 f9                    movq    %rdi, %r9
100003f2d: 80 f9 02                    cmpb    $2, %cl
100003f30: 72 28                       jb      0x100003f5a <__Z2f2RK3Bar+0x48>
100003f32: 48 8d 4f 08                 leaq    8(%rdi), %rcx
100003f36: 48 8d 14 d5 f8 ff ff ff     leaq    -8(,%rdx,8), %rdx
100003f3e: 48 89 fe                    movq    %rdi, %rsi
100003f41: 8b 01                       movl    (%rcx), %eax
100003f43: 49 89 c9                    movq    %rcx, %r9
100003f46: 3b 06                       cmpl    (%rsi), %eax
100003f48: 72 03                       jb      0x100003f4d <__Z2f2RK3Bar+0x3b>
100003f4a: 49 89 f1                    movq    %rsi, %r9
100003f4d: 48 83 c1 08                 addq    $8, %rcx
100003f51: 4c 89 ce                    movq    %r9, %rsi
100003f54: 48 83 c2 f8                 addq    $-8, %rdx
100003f58: 75 e7                       jne     0x100003f41 <__Z2f2RK3Bar+0x2f>
100003f5a: 44 89 c8                    movl    %r9d, %eax
100003f5d: 29 f8                       subl    %edi, %eax
100003f5f: c1 e8 03                    shrl    $3, %eax
100003f62: 4d 39 c1                    cmpq    %r8, %r9
100003f65: b9 ff 00 00 00              movl    $255, %ecx
100003f6a: 0f 45 c8                    cmovnel %eax, %ecx
100003f6d: 0f b6 c1                    movzbl  %cl, %eax
100003f70: 5d                          popq    %rbp
100003f71: c3                          retq
```

最后是使用 ArrayView 的实现:

```
auto f3(Bar const& bar, uint8_t n) -> IntOpt<uint8_t> {
    return ArrayView{bar.foo, bar.num}.MinElemIndex(
        [](auto &&l, auto&& r) { return l.a < r.a; });
}
```

```
0000000100003f72 <__Z2f3RK3Barh>:
100003f72: 55                          pushq   %rbp
100003f73: 48 89 e5                    movq    %rsp, %rbp
100003f76: 8b 4f 38                    movl    56(%rdi), %ecx
100003f79: 80 f9 07                    cmpb    $7, %cl
100003f7c: b8 07 00 00 00              movl    $7, %eax
100003f81: 0f 42 c1                    cmovbl  %ecx, %eax
100003f84: 84 c0                       testb   %al, %al
100003f86: 74 08                       je      0x100003f90 <__Z2f3RK3Barh+0x1e>
100003f88: 3c 01                       cmpb    $1, %al
100003f8a: 75 08                       jne     0x100003f94 <__Z2f3RK3Barh+0x22>
100003f8c: 31 c0                       xorl    %eax, %eax
100003f8e: eb 26                       jmp     0x100003fb6 <__Z2f3RK3Barh+0x44>
100003f90: b0 ff                       movb    $-1, %al
100003f92: eb 22                       jmp     0x100003fb6 <__Z2f3RK3Barh+0x44>
100003f94: 44 0f b6 c0                 movzbl  %al, %r8d
100003f98: ba 01 00 00 00              movl    $1, %edx
100003f9d: 31 c0                       xorl    %eax, %eax
100003f9f: 0f b6 f0                    movzbl  %al, %esi
100003fa2: 8b 0c d7                    movl    (%rdi,%rdx,8), %ecx
100003fa5: 89 d0                       movl    %edx, %eax
100003fa7: 3b 0c f7                    cmpl    (%rdi,%rsi,8), %ecx
100003faa: 72 02                       jb      0x100003fae <__Z2f3RK3Barh+0x3c>
100003fac: 89 f0                       movl    %esi, %eax
100003fae: 48 ff c2                    incq    %rdx
100003fb1: 49 39 d0                    cmpq    %rdx, %r8
100003fb4: 75 e9                       jne     0x100003f9f <__Z2f3RK3Barh+0x2d>
100003fb6: 5d                          popq    %rbp
100003fb7: c3                          retq
```