Interoperate Swift with C

认识Swift指针家族类型 (https://www.boxueio.com/series/interoperate-swift-with-c/ebook/249)

C指针是如何桥接到Swift的 (https://www.boxueio.com/series/interoperate-swift-with-c/ebook/251)

使用Buffer视图改进内存访问

Back to series (/series/interoperate-swift-with-c)

在这一节，我们来了解 UnsafeMutableBufferPointer 的用法。它和传统意义上的 buffer不同，简单来说，它就像是在原始内存空间上的一层view。Buffer会以对应类型的 MemoryLayout<T>.stride 为单位，把内存空间看成是一个集合，进而让我们用类似处理集合的方式访问底层内存。这样的代码不仅更安全，也更符合Swift的编程习惯。

创建一个buffer

为了创建一个buffer，首先我们必须有一个 UnsafeMutablePointer 作为起始地址，然后把这个起始地址和内存地址的长度一同传递给它：

```
var head = UnsafeMutablePointer<Int>.allocate(capacity: 10)
var buffer = UnsafeMutableBufferPointer(start: head, count: 10)
```

这样，我们就可以借助 buffer 把之前分配在内存中的10个 Int 作为一个集合来处理了。接下来，先了解一些常用的获取buffer信息的方法。

获取buffer的常用信息

首先，对于一个“集合”来说，最基础的两个信息，就是是否为空以及包含多少个元素：

```
buffer.isEmpty // false
buffer.count   // 10
```

其次，是获取集合中的最大和最小元素：

```
buffer.max(by: >) // 10
buffer.min(by: <) // 1
```

这里， max 和 min 都有一个closure参数 by ， by 接受两个参数，**表示的是内存地址上要比较的两个值**。当closure返回 true 时， max 和 min 就选取 by 的第一个参数。

实际上， max 和 min 都还有一个带默认参数的版本，会把 by 指定为我们上面默认使用的比较操作符。因此，当我们要获取最大最小值的时候，直接调用 max() 和 min() 就好了。

第三，是获取内存集合起始地址的 buffer.baseAddress ，它的类型是 UnsafeMutablePointer<T>? 。不难理解，这个地址就是用于初始化buffer的 start 参数。要说明的是， **baseAddress 是只读的，我们不能修改它让buffer成为其它内存区域的view**。

访问buffer内容的方法

使用下标操作符

接下来，来看如何访问集合中的内容。首先， buffer 也支持使用下标操作符直接访问，包括单个位置的元素，以及某个 Range 的元素：

```
let m = buffer[0]
let n = buffer[0..<5]
```

其中， [0] 这种形式返回的是地址保存的值，而 [0..<5] 这样的形式返回的是buffer的一个“切片”，记录了buffer中选中区域的起始和结束位置，为了访问这个切片中的元素，我们可以这样：

```
for i in slice {
    print("s: \(i)")
}
```

要说明的是，这两种下标操作符的用法，**我们必须自行确保使用的位置是可用的，否则会导致运行时错误**。

使用first和last

除了下标操作符之外，如果要访问buffer“两头”的元素，还可以使用 `first` 和 `last` 方法，像这样：

```
let first = buffer.first // 1
let six = buffer.first(where: { $0 > 5 }) // 6
let last = buffer.last   // 10
```

其中， `first` 有两个版本，分别用户获取第一个，以及满足 `where` 条件的第一个元素。而 `last` 则只有一个版本，用于获取缓冲区最后一个元素。这三个方法都返回一个 `Int?` ，在取不到值的时候，返回 `nil` 。

修改buffer元素的方法

到目前为止，我们介绍的都是从 `buffer` 中获取内容的方法，除此之外， `buffer` 还提供了修改自身内容的方法。其中最简单的，就是通过下标操作符：

```
buffer[0] = 0
```

只是，我们要确保使用的下标索引是正确的。我们还可以对 `buffer` 中的所有元素进行排序：

```
buffer.sort(by: >)
let sorted = buffer.sorted(by: >)
```

和标准集合类型一样， `UnsafeMutablePointer` 的排序函数提供了两个版本，没有ed后缀的，表示直接在 `buffer` 原地排序，有ed后缀的，则会按照 `buffer` 中的值排序之后，返回一个 `[Int]` ，表示排序的结果，而不会修改 `buffer` 自身。

当然， `sort(by:)` 和 `sorted(by:)` 的closure参数都是可选的，它们的默认值，就是 `>` 。

另外一对修改方法是逆序 `buffer` 中的值：

```
buffer.reverse()
let reversed = buffer.reversed()
```

其中，ed后缀的含义，和 `sort` 是相同的。唯一不同的，是 `reversed()` 方法并不返回一个 `[Int]` ，而是一个表示原 `buffer` 逆序排列的view，类似 `buffer` 就是它指向内存的view。为了从view得到对应的 `[Int]` ，我们得使用 `Array.init` 方法：

```
let reversedArray = [Int](reversed)
```

使用buffer的迭代器

接下来，说到集合，不能不提的一个概念，就是迭代器，它把依次访问集合的动作进行了抽象和统一。既然要用起来像一个集合， `UnsafeMutableBufferPointer` 当然也提供了自己的迭代器。就像这样：

```
var iter = buffer.makeIterator()

while let value = iter.next() {
    print("Iter: \(value)")
}
```

我们先调用 `makeIterator` 方法，获取指向初始位置的迭代器对象，然后，不断调用它的 `next()` 方法读取当前位置的元素，并让 `iter` 移动到下一个位置，直到 `next()` 返回 `nil` 。

除此之外，有了迭代器对象之后，我们还可以对遍历 `buffer` 的结果进行进一步加工，这和我们对遍历集合类型时使用的 `map / filter / reduce` 是一样的：

```
/// [2, 4, 6, 8, 10, ...]
let doubleArray = iter.map { $0 * 2 }
/// [2, 4, 6, 8, 10, ...]
let evenArray = iter.filter { $0 % 2 == 0 }
/// 5
let sum = iter.reduce(0) { $0 + $1 }
```

其中， `map` 和 `filter` 都会返回 `[Int]` ，包含了变换后的结果，而 `reduce` 则返回 `Int` ，表示合并的结果，**它们都不会改变 `buffer` 的原始内容**。但是，一直以来都有开发者抱怨，必须通过迭代器才能遍历 `buffer` 是个设计缺陷。为什么不能像这样直接遍历呢？

```
let doubleArray = buffer.map { $0 * 2 }
let evenArray = buffer.filter { $0 % 2 == 0 }
let sum = buffer.reduce(0) { $0 + $1 }
```

于是，在Swift 4中， `UnsafeMutableBufferPointer` 就加入了这些API，在Xcode 9里，我们可以不用借助迭代器来遍历 `buffer` 了。

What's next?

以上，就是 `UnsafeMutableBufferPointer` 的概念和常见用法，它作为内存区域的一层view，为我们带来了更安全和便捷的开发体验。现在，我们手头已经积累了不少关于指针的知识了。下一节，我们来了解C中的指针是如何桥接到Swift的。

⏮ 认识Swift指针家族类型

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/249)

C指针是如何桥接到Swift的 ⏭

(https://www.boxueio.com/series/interoperate-swift-with-c/ebook/251)