六、阶段性总结和补充内容

到这里我们就学习完了CorScript编程语言的基础概念，让我们来看看我们都学习了什么。

1. 输入输出

输入：system.in.input()

输出：system.out.print(<参数>)　　(不换行)

system.out.println(<参数>)　　(换行)

2. 变量和类型

变量定义：var <变量名> = <表达式>

new运算符：new <类型>

基础类型：number　数值类型

boolean　逻辑类型

string　文字类型

array　数组类型

3. 作用域

目前我们学过的结构中会拥有独立作用域的是：

if、while、loop/loop-until、function、case/default

在独立作用域中的变量在外部不能直接访问。

4. 分支程序结构

```
if 条件
    语句块.
end
```

或

```
if 条件.
    语句块1.
else.
    语句块2.
end.
```

```
switch 表达式
    default [可选]
        语句块
    end.
    case 常量标签.
        语句块.
    end.
end.
```

5. 循环程序结构

```
loop
    语句块.
end
```

或

```
loop
    语句块.
until 条件.
end.
```

while 条件

语句块.

end.

6. 循环控制语句

break 跳出最上层循环.

continue 进入最下层循环的下一轮

7. 函数.

function <函数名>(<参数列表(可选)>).

函数体

end.

[](<参数列表(可选)>) -> 表达式

8. 返回语句

return 终止函数执行并返回0

return 表达式 终止函数执行并返回表达式的值.

9. 递归和RVO

递归(函数调用自身的行为) { 有限递归 — 有递归终点.
无限递归 — 无递归终点.

RVO: 返回值优化, 提高多层返回的性能

10、数组以及补充内容.

① 类型名：array

② 字面量：大括号括起的以逗号分隔的任意个数元素.

③ 初始值：{}（空数组）.

④ 访问元素：下标访问运算符：[] <数组名>[表达式].

下标起始值：0

下标最大值：<数组名>.size()-1

⑤ VLA：当访问范围外的元素时会自动增长，增长的部分将填零。

⑥ 补充内容：数组扩展功能

- 数组.at(下标)

  与下标访问运算符相似，但会检查下标是否越界，也就是说，不会自动增长。

- 数组.front().

  获取数组的首元素，注意如数组为空访问首元素的行为未定义。

- 数组.back().

  获取数组的尾元素，注意如数组为空访问尾元素的行为未定义。

- 数组.empty().

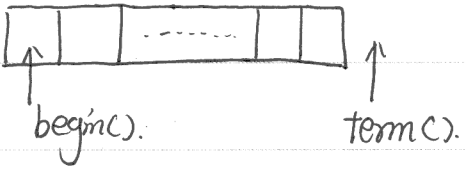
  判断数组是否为空，返回一个布尔值 逻辑量。

- 数组.size()
  返回数组容纳的元素数。
- 数组.clear()
  删除数组中的全部内容。
- 数组.push_front(元素)
  在数组前部插入元素。
- 数组.pop_front()
  删除首元素。
- 数组.push_back(元素)
  在数组后部插入元素。
- 数组.pop_back()
  删除最后一个元素。

数组支持迭代器，通过迭代器我们能够到操作任意位置的元素：

- 数组.begin()
  返回指向容数组第一个元素的迭代器。
- 数组.~~ex~~term()
  返回指向数组尾部的迭代器。

大家可能不太理解迭代器，我们先看下 begin() 和 ~~en~~ term() 到底指向何处。



begin() 指向首元素，比较容易理解。但 term() 实际上指向的是尾元素的后面的一个实际不存在的元素，——仅仅是为便于访问。至于为什么，我们继续往下看：

迭代器支持三个方法：

- 迭代器.forward()

  向前移动迭代器一个单位。

- 迭代器.backward()

  向后移动迭代器一个单位。

- 迭代器.data()

  获取迭代器指向的元素。

这里需要注意的是迭代器的"前"指的是相对于数组的"后"，也就是说"前移"其实是向数组后部移动。

有了迭代器我们就能够自由修改数组了：

- 数组.insert(迭代器，元素)

  在迭代器指向的元素之前插入新元素，返回指向插入的元素的迭代器。

0 数组、erase(迭代器).

~~删除迭代器指向的元素，返回指向要删除的元素的下一个元素的迭代器。~~

注意：一旦对数组进行插入和删除，原迭代器就会失效。另外，在操作迭代器之前要确定迭代器有效——即不等于 term()。所以 term() 的意义就是，帮助判定迭代器是否超出范围。

~~作业：① 实现插入删除数组中间元素。~~

~~② 用迭代器遍历数组，包括正序和反序。~~

#例 6.10.1

```
var  arr = {2,4,6,8}
var  it = arr.begin()
while it != arr.term()
    system.out.println(it.data())
    it.forward()
end
```

~~it.backward()~~

~~while it!=~~

~~system.out.println(it.data())~~

~~it.back~~

~~loop~~

~~it.backward()~~

~~system.out.println(it.da~~

```
loop
    it.backward()
    system.out.println(it.data());
until it == arr.begin();
end.
```

例 6.10.1 展示了如何使用迭代器进行正序遍历和反序遍历。思考：

○为什么正序遍历使用while循环而反序遍历使用loop-until循环？反序遍历能否改成while循环？

正序遍历时，我们拿到的起始迭代器指向首元素本身，结束迭代器指向尾元素右部，所以我们只需在循环开始处判定迭代器是否有效。

但反序遍历时，我们拿到的起始迭代器指向尾元素右部，而结束迭代器却指向一个实际存在的元素。若使用同样的循环，只是将it.forward()换成it.backward()，显然是不行的，因为it在一开始就等于arr.term()，循环并不会执行。就算是将条件换成it!=arr.begin()，然后我们首先将it前移一个单位，这个循环也会跳过首元素。

而解决这个问题的最佳方案无疑是 loop-until ——使循环直到 it 等于 it.begin() 时，跳出。。

作业：使用迭代器反向遍历一个数组并将其元素反转。

现在我们能使用迭代器遍历数组了。之所以使用迭代器而不是下标，有个重要的原因——迭代器可以随意修改数组。。

举例 6.10.2.

```
var arr={1,2,3,4,5,6}
var it=arr.begin()
while it!=arr.term().
    if it.data()%2==0
        it=arr.erase(it).
    else
        it.forward().
    end.
end.
```

例 6.10.2 实现了删除数组中所有的偶数。程序执行完成后，数组将变为 {1,3,5}。

再次强调，arr.erase(it)执行后原迭代器就会失效，去继续操作行为未定义。

再来看一个插入的例子

#例 6、10、3.

```
var n=system.in.input().
var arr=new array.
var tmp=0
var i=1
while i<=n.
    tmp=system.in.input().
    var it=arr.begin().
    while it!=arr.@term().
        if tmp<=it.data().
            break.
        else
            it.forward().
        end.
    end.
end.
```

arr.insert(i, tmp).

i=i+1

end.

例6.10.3 实现了将用户输入储存至数组中并同时进行升序排序

这个程序的思路是：找出第一个大于或等于用户输入的元素，然后将用户的输入插入到此元素之前。

作业：① 实现删除重复的连续元素

② 实现降序排序

11. 字符串以及补充内容。

① 类型名：string.

② 字面量：双引号括起的任意内容。

③ 初始值："" (空串)。

④ 补充内容：

连接两个字符串可用"+"运算符，如"Hello"+"World"

将任意变量转换为字符串可用 to_string(变量) 函数

作业：使用一个 system.out.println 输出例1.4.1的问候语