

第 3 周笔记

对应章节：Chapter 2、7.8.1、7.8.2

说明：我们根据需要发布参考笔记，这部分内容作为对教材的说明解释或者额外补充，教材正确合理的部分不作赘述。对于竞赛，语言深度不要求很深，那些不必掌握的知识，笔记中会有注明。

1. 综述

完成了第一章的学习之后，C 语言除了指针的重要特性都有了一个基本的印象（其实指针是肯定用到了只不过没有明确提出来），已经可以写大量有用的程序了，接下来几章的学习，将填补我们现在的空白，逐步完善，构建一个严谨的编程体系。有些内容尽管你已经“学过”了，也建议不要跳读，这本书已经足够精炼，没有废话，还是抱着“虚心若愚”的心态好好理解~。

2. 驼峰命名法（CamelCase）

骆驼式命名法的命名规则可视为一种惯例，并无绝对与强制，为的是增加识别和可读性。它的含义和它的名字 CamelCase 一样。C 语言规定，空格区分变量，但是下划线算作正常字符，也就是说：

```
int camel_case_name;
```

这种表示是合理的，那么利用驼峰法（小驼峰法）还可以这么表示：

```
int camelCaseName;
```

对于函数名、类名、命名空间名（C++）等还有大驼峰法（把小驼峰的首字母也大写）：

```
void CamelCaseName();
```

不管是什么方法，目的只有一个，方便代码的阅读，下划线和大小写都可以区分不同的单词，根据个人喜好决定即可。编程过程中良好的规范会带给你高效率，这种产出随着代码量的增长而变得明显。对于变量名来说，如果是循环控制中的计数器变量，那么用“i”就好，太长的名字自己打字还费劲，如果是重要的变量或全局的变量，那么还是推荐更长一点的变量名，一可以防止冲突，二可以在语义上时时刻刻给自己一个提示，让思路更清晰。

随着计算机和编译器的进步，中文等文字作为变量名逐渐被允许，但是竞赛使用的编译器比较古老，因此我们在这里禁止使用中文。

3. 原码，反码和补码

计算机的世界是一个二进制的世界，而且计算机必须确保在不越界的前提下，给出正确的计算结果，本章 2.2 节提到了有符号和无符号概念，2.9 节提到了位运算，因此我们简单讲解一下整数在计算机内存中的表达方法。我们知道，数有正负之分，为了确定正负，计算机必须舍弃一位（位(bit)是最小的单位，值为 0 或 1，8 位为 1 字节(byte)前面提过）来存储数的符号，这就是“有符号”数，那么“无符号”数就规定最高位不存储符号仍然存储值，那么当然无符号的数要比同级有符号数表达的最大值大。这里我们考虑 signed char 类型，它在我们的计算机上占用一个字节来存储。

按照我们现在的逻辑定义原码，进行 1-1 也就是 1+(-1)这个运算，应该为：

```
00000001 + 10000001 = 10000010
```

在十进制，这句话的意思是 1+(-1)=-2，显然是错的。于是我们想到了反码，也就是对除符号位之外的所有位逐位取反（0->1，1->0），再进行同样的运算，应该为：

```
00000001 + 11111110 = 11111111
```

原码 10000001，反码 11111110，计算后的反码 11111111，原码 10000000，也就是-0。然而零怎么可以有正负两个零呢（事实上，双精度浮点数还是表达了+0.0 和-0.0 两个零，此处先不讨论）？其实我们可以再尝试一个运算：1-2 也就是 1+(-2)，应该为：

```
00000001 + 11111101 = 11111110
```

反码 11111110 转化成原码是 10000001，这个值是-1 是正确的。也就是说，反码计算在计算所有的负数的时候，都可以给出正确的结果，因为它的取值空间和原码是一一对应，完全相同的，那么为了解决-0 的问题，我们自然想到了加 1 的做法，也就出现了补码。补码由原码的反码加 1 得到，对于 1-1 这个我们希望为 0 的运算，它变成了这个样子：

```
00000001 + 11111111 = 00000000
```

00000000 就是 0，这里我们不说它是正零，而且习惯上这样表示零也比 100000000 更合理。对于其他的负数运算，和反码对问题的解决是一致的（因为只是后移了一位），完美解决正负两个零的问题。最终的结论是，计算机存储有符号整数时，对于负数，采用补码的方式存储和解析，这样做给 CPU 的运算带来了巨大的方便，1-1 可以直接当做 1+(-1)来计算，除此之外，在运算的时候符号位同样参与运算，规则一致，简化了不必要的操作。根据这些理论，我们也就解释了 INT_MIN 和 INT_MAX。——尽量看懂，看不懂就知道这个结论就可以，不影响我们写程序因为这些都是底层的事情。

4. 短路运算符

C 语言中的条件运算符&&和||都是短路运算符，它的含义是，求值从左至右，当前面的值已经可以确定整个表达式的值（比如逻辑与出现了假，逻辑或出现了真）时，停止求值并直接返回结果。这给编程带来了安全性的福利，比如说：

```
...(i < largestBound && array[i] != value)...
```

上一讲我们提过，如果要访问的数组元素下标超过了上界会引发访问冲突导致程序崩溃，如果逻辑与是非短路运算符，那么上面这条语句就是不安全的，因为当 i 为 largestBound（只是个符号，表示数组的最大下标）时仍然要找 array[i]，当然会引起冲突。这就是短路运算符的必要性和优势。

5. 宏替换初步

我们在很多章节接触到了宏替换也就是#define 指令，它的作用是把后面所有代码中出现的一个特定的符号替换为特定的值或者别的东西，这步操作在编译时进行，所以不会增加运行开销，如：

```
#define PI 3.1415926
...b = a * PI;...
```

在编译时这段代码会变成

```
#define PI 3.1415926
...b = a * 3.1415926;...
```

因此才叫“宏替换”，那么按照这个原理，宏替换不仅仅可以定义一个符号表达一个值，还可以让符号表达符号，比如：

```
#include <stdio.h>

#define max(a, b) a > b ? a : b

int main()
{
    int a = 1, b = 2;
    printf("%d", max(a, b)); //This line.
    return 0;
}
```

它输出的结果是 2，事实上在编译过程中，有注释的这行中的 max(a,b)就已经替换成了 a>b?a:b（三目运算符的相关知识见 2.11 节），因此 max()并不是一个函数，只是一个替换规则。在动态规划的学习

中将经常使用这个指令。此外，我们用`#define`指令也可以写一套类似函数的东西出来（但是绝对不推荐，写错了根本没法调试，而且怎么替换的我们也看不到，除了装 `thirteen` 之外没有实际意义）。

6. 字符串操作库函数

2.3 节中提到的 `strlen()` 函数在库中是的确存在的（当然不是这么写的），有关更多字符串操作的库函数，参见 7.8.1 和 7.8.2 节，使用前务必包含相应的头文件。涉及指针的概念不需纠结，现在做到会照搬使用就可以。

7. 运算符优先级

我们讲究基础但是不建议较真，运算符优先级这种东西，并不建议大家全记住，理解基本规则就可以。真正使用的时候，如果想让表达式的某一部分优先计算但不确定运算符优先级是否可以做到，那么为何要和自己过不去？加括号啊!!!! 括号肯定是优先的啊!!!!

另外，教材中强调了对未定义的（依赖编译器决定的）语句的警告，比如：

```
printf("%d %d\n", ++n, power(2, n));
```

这比走钢丝都危险，出了错调试都调试不出来，何苦和自己过不去。再比如：

```
a[i] = i++;
```

这种未定义的情况，不同的编译器产生的汇编代码有可能不同，分辨它的执行情况只能看汇编代码（机器指令），最常发生在实际水平不高还想让别人刮目相看的程序员身上，我们一定要重视这个警告。

另外，前置++和后置++是有严重区别的，不理解的同学请反复阅读 2.8 节或与同学交流。

8. 将数组作为参数传递给函数

由于没有学习到指针，这里我们只说明做法不说明原因。如果想把一个数组作为参数传递给函数，请采用如下的方式：

```
void Function(int array[], int n);
```

这样做，在内存中事实上数组还是原来的空间，没有拷贝出新的数组，所以函数内修改数组的元素后也会影响到主调函数，这与上一讲的笔记中有所不同，原因是这次我们传递的是指针的值，上次我们传递的是整型变量的值（以后讨论）。为什么要多一个参数 `n` 呢？因为对于字符数组，我们可以通过判断 `'\0'` 来获得数组内容结束的位置，而对于其他数组我们没有办法知道数组的界限，尤其是如果产生访问越界将导致崩溃。因此我们采用传递两个参数，一个参数表示数组在内存中的位置，一个参数表示数组大小的传递方式。

传参后，新的函数内使用数组的方式与原来完全相同。

（所有内容到此结束）