梅河口市第五中学信息学竞赛培训材料

第 2 周笔记

对应章节：1.6~1.10（含）、7.2、7.4

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

这部分介绍了 C 语言一些高级的特性，内容高度概括且非常核心和重要，正常情况下应该会感到不适应，突然间多了一堆不理解的东西。策略和上次一样，不必较真，哪怕连蒙带骗，程序能跑起来为主，先理解到一个“会照着用”的状态，以后各章会逐渐展开，详细介绍。

另外，尽管书上的代码不针对竞赛，但它各方面做得都很好，可以多读读，多理解理解。

不必阅读的部分：教材第 19 页中间，从“回顾一下”到 1.7 节末尾。

2. 数组元素的初始化

教材第 15 页，1.6 节开始介绍了一个统计数字和其他字符出现次数的程序，这个程序第一次使用了数组。数组是具有相同属性的一组元素的集合，大大方便了数据的定义和使用。但数组不灵活之处是，第一，数组在内存中必须是**连续的**，因为计算机在访问数组的元素的时候是以从首地址向后偏移的方式访问的，这也就造成了分配大规模数组会比较困难（因为物理上被占用的内存是零碎的，所以连续的大空间比较少），后期我们将用链表突破这个局限；第二，C 语言的数组比较底层，没有对异常作出处理，如果数组的长度是 10，我却非要访问下标为 10 的元素（请注意，数组下标**从零开始**），那么这将导致访问冲突，使程序在运行时崩溃掉，**我们必须在平时足够重视这个问题**；除此外的一些内存管理和数据结构的问题，我们留到后期讨论。

回到教材中的程序，main()函数的第 5 行和第 6 行写了这样一段代码：

```
for (i = 0; i < 10; ++i)
    ndigit[i] = 0;
```

这里我们强调一下，控制语句的后面如果不写花括号{}，**则只能控制其后一条语句**，你可以全加花括号{}防止出现错误。不同层次的语句请一定要用 Tab 键对齐，严禁使用空格对齐的方式。

这两句话的作用，显然是将数组中所有的元素初始化，但它的效率并不高，有多少元素，就需要循环多少次，因此我们更建议下面这种方式：

```
memset(ndigit, 0, 10 * sizeof(int));
```

这个函数定义在<string.h>头文件中，使用之前务必#include 该头文件。memset()函数原型为：

```
void *memset(void *s, int ch, size_t n);
```

函数的作用为，**向 s 指向的内存区域，用 ch 的值填充 n 字节的数据**。指针的概念还没接触到，不用在意，给大家看这个原型的目的是：**一定要注意第二个参数是要填进去的值，第三个参数是填进去的字节数**！不幸写反了，后果可就不好玩了。

使用数组和很多其他数据时，我们常常使用三个值：0，1 和－1。0 和 1 可以用来表示真假，在 if、else if、while、for 等地方会大显身手，－1 一般用来标记错误或意外的情况，因为很多情境里－1 都是一个“不可能”的值。因此我们定义了一个数组之后，可以先用 memset()函数让所有的元素都有一个初始值而后进行其他操作或者使用。我们写这两种：

```
memset(ndigit, 0, 10 * sizeof(int));
memset(ndigit, -1, 10 * sizeof(int));
```

程序都像预期的那样，ndigit 数组的每一个元素的值都变成了 0（第一种）、-1（第二种），但如果想用

这种方式让数组中每个元素都为 1，发现程序就会出问题了，数组的所有元素都变成了 16843009 这个鬼畜的值，也因此有教材认为这个函数不支持赋除了 0 和-1 外的值。其实函数没有错，函数的功能是“填充字节”，它是**逐个字节填充**的。1 个字节是 8 位，int 型变量是 4 个字节，32 位，那么填充之后，在内存里数组每个元素就都变成了这个样子：

```
00000001 00000001 00000001 00000001
```

一个元素，32 位，空格仅作便于区分字节之用，长得帅的人都已经学过进制转换了，这个值显然就是 16843009。**那么为什么填 0 和填－1 不会出现问题呢，为什么 int 的最小最大值是-2147483648 和 2147483647 呢，请同学们课后思考，并在下次同伴讨论时给出答案。**

额外提示一句，如果程序需要用到 int 的边界值且你不确定自己记忆中是正确的。那么可以写 INT_MAX 和 INT_MIN，它们在头文件<limits.h>中用#define 指令定义过，宏替换教材也已经讲过。

3. 数学库函数

教材第 17 页下方提到，标准库中提供了一个计算 x^y 的函数 pow(x, y)，这里我补充一下，对于第 17 页的程序，比较严谨的调用库函数的方法是：

```
...(int)pow(2.0, (double)i)...
```

其中涉及了强制类型转换下一章会探讨，这样做的原因是 pow()函数的声明为：

```
double pow(double x, double y);
```

现阶段，不这样做表面上没有问题，因为 C 语言提供隐式类型转换，也就是说，写 `pow(2, i)`，编译器会把 2 和 i 自动转化成双精度浮点数，但 `printf` 的第一个参数如果写成"%d"则不可以，会输出 0 作为结果。从 C 到 C++，国际标准摒弃了隐式类型转换，函数也有了重载机制，于是不允许出现 `pow(2, i)`这种不规范的写法。因此还是提出来，防止以后出错误。

有关更多数学库函数，参见附录 B.4，即教材第 228、229 页，附录 B 其他部分在现在不需要全阅读。

4. 函数引导

如果你没有看过《盗梦空间》，强烈推荐**在有空的时候**观看这部电影。（略去一万字）

我们写程序的时候离开函数了吗？从基础的 scanf()，printf()到 pow()，memset()等等，我们总是通过调用函数来实现某种功能，就像系统调用了 main()函数执行我们的程序一样。我们提供给函数一些东西（参数），函数做了某种事情（函数体），最后返回给我们一个东西（返回值），即完成了一次调用。

设计一个函数，首先要明确函数的功能，它应该合理地划分出来，既要够细够明确，又不能太受限，如果你设计了一个函数，它的参数列表里有十几项，那么说明这个函数在功能上就有不合理之处，更不要提拓展和通用了。其次，函数要明确自己的内部和外部，通过明确的接口与外界发生联系（接口，比如说你把 U 盘插在网口上，当然插不进去，因为他们必须有统一的接口）。

编程像写作，它有其规定的一整套逻辑和语法，但并没有限制你所用的修辞和采用的策略。对于函数，编译器只检查接口有没有错误，即返回值类型、参数列表是否严谨对应。编译之后，非内联（后期学）的函数，只有一套二进制代码作为“身体”，每次需要调用的时候，就会把这部分代码压到内存中然后执行这些指令。也就是说，一个函数内部可以调用别的函数（就像 main()调用了 printf()），一个函数甚至也可以调用自己本身，这也就是递归，作为重要的算法策略，以后介绍。

5. 传值调用与变量作用域

1.7、1.8 和 1.10 节都体现了变量作用域的规则。变量必须先声明后使用，这是无须解释的，然而，在函数内部定义的变量，在外部是否是可用的呢？我们看这样一个例子：

```
#include <stdio.h>

void Compare(int a, int b);

int main()
{
```

```
    int a = 2, b = 1;
    Compare(a, b);
    printf("(%d, %d)", a, b);
    return 0;
}

void Compare(int a, int b)
{
    if (a > b)
    {
        a = a + b;
        b = a - b;
        a = a - b;
    }
}
```

观察理解这段代码，我们先声明了一个函数 Compare()，它有两个整型变量作为参数，分别是 a 和 b，它的功能是，如果 a 大于 b，则交换 a 和 b，即保证 a 和 b 是非降序的（看一下它的函数体，相比于原来的写法，这样做不需要中间变量）。在入口 main()函数中，我们定义了两个整型变量 a 和 b，值分别是 2 和 1，然后调用 Compare(a, b)，最后打印出 a 和 b 的值，**请思考出自己认为的结果再往下看**。

答案是——**(2, 1)**，如果做错，请重新阅读 1.8 节。对于这段代码，尽管主函数中整型变量叫做 a 和 b，Compare()函数中的整型变量也叫做 a 和 b，但是此 a 非彼 a，此 b 非彼 b，换言之，当执行到 Compare()函数的内部时，main()函数中的 a 和 b 对于 Compare()函数是“不可见”的，这两套 a 和 b 对于两个函数而言都是“私有变量”或者说“局部变量”，它的有效范围在同一级花括号{}界定的范围中间，每一套 a 和 b 都有自己独立的存储空间，外部通过相同的名字无法访问。（这是一个经典问题）

请再看这样一个程序：

```
#include <stdio.h>

int main()
{
    int i = 2;
    printf("<");
    for (; i < 5; i++)
    {
        int i = 0;
        ++i;
        printf("%d,", i);
    }
    printf("%d>\n", i);
    return 0;
}
```

观察理解这段代码，**给出自己的结果，再到电脑上编译或阅读下面的部分**。

程序运行后，打印到屏幕上的内容是**<1,1,1,5>**，这段程序看起来就比较“闹心”，因此我在解释之前先声明——**就算你理解了所有原理，也不要这样“走钢丝”似地写程序**。

其实原理也不复杂，我们在 main()函数体也就是最外层的花括号{}界定的这一级上定义了一个整型变量 i 并给它分配了存储空间，它的值初始为 2，当执行到 for 循环控制的下一级，也就是中间这层花括号{}间时，又定义了一个整型变量 i，此时程序又分配了新的存储空间，这个 i 的作用域或者说有效范围是 for 循环控制的这对花括号{}界定的语句块，执行完这个语句块，i 对应的存储空间就会释放，再次执行，又要重新分配内存并赋初始值 0。在它生存的时候，由于 i 的名字和外面一层的整型变量 i 是一样的，**因此它屏蔽了外面一层的 i**。跳出 for 循环后，i 还是原来的 i，在上次作业的第一题我们知道，for 循环之后 i 的值变成了 5，因此最后输出了这样的结果。

再次强调，名字写成 i1 和 i2 又不会累个好歹，干嘛非和自己过不去……

6. 赋值表达式的值

教材 21 页，1.9 节例程中出现了大量利用赋值表达式的值的语句，如果不理解，请参考教材第 11 页的有关讲解。赋值表达式本身的值是运算符左侧变量的值，用在 return 语句处可以使代码更紧凑。但由于运算符优先级的问题，相应的圆括号不能省略。

7. 默认返回值

教材第 22 页第 4 段末尾提到，“默认返回值类型为 int，因此这里的 int 可以省略”，这句话在今天是错误的，因为标准对此已作修正（上次笔记也提过），函数返回值必须显式标记出来，就算函数不需要返回任何值，也要标记“void”。

8. 字符串

阅读到 1.9 节，你已经开启了编程中兼具魅力与危险的一片天地——字符串。可以参考 7.2 节和 7.4 节学习字符串的标准输入输出。字符串，事实上是字符元素构成的数组。数组的容量和字符串的长度不一定是一致的，比如，我在内存中申请和分配一段共 100 个 char 类型元素的数组，但是我只想表达一个只有十个字的字符串，那么我不需要使用到后面的几十个元素这当然可以，但必须要有一个标识告诉我字符串到哪里为止结束了。这也就是’\0’的用处。右斜杠是转义字符，’\0’在 ASCII 码中值为 0，和’0’的值是不一样的（否则，字符 0 就没办法在字符串中表示了）。可以运行下面两条语句：

```
printf("%d", '\0');
printf("%c", '\0');
```

它们输出到屏幕上的内容相同吗？不同。前者为 0，后者为空。这也就是“格式化输出”的意义——相同的一段内存，相同的二进制表示，我可以解析成不同的东西。

这是一个极容易出错的地方，凡需要字符串之处，请千万注意’\0’的问题，三思而后行。

9. 声明与定义

```
int a;
extern int a;
```

上面这两条语句十分相似，实际上却有着严重的不同，在教材第 25 页强调，“定义”和“声明”是两个不同的词。上面第一条是定义，第二条是声明。

回想，我们编写函数的时候也有类似的说法，在 main()函数之前出现的带分号的语句：

```
void Compare(int a, int b);
```

便是对函数的声明，它规定了函数的返回值，名称，和参数列表，编译器可以据此推断对该函数的调用是否正确。而 main()函数结束之后，没有分号并且紧跟着函数体的部分，便是函数的定义。我们当然可以把定义写在 main()函数之前，这样当编译到 main()函数的时候，编译器既知道函数的返回值和参数列表，又知道具体的函数体，也就省去了先声明的麻烦。但这样做不好之处是降低了代码可读性，因此**还是建议大家先声明，后定义**。

声明，是告诉编译器“有这个东西，它已经或者会被创造”，定义，是告诉编译器“在这里创造这东西”。如果我们写这样的一条语句：

```
extern int a = 0;
```

那么编译器会提示此处有错误，原因是“错误：不允许对外部变量的局部声明使用初始值设定项”，不对啊，说好的可以赋初始值呢？——区别在于，extern 标识符决定了这句话并没有创建一个整型变量 a 并分配存储空间，因此无法赋值——编译后“链接”的过程，其一就是循着 extern 标识符，寻找那个在“某处”被定义了的名字叫 a 的整型变量（无需了解什么是链接）——因此**声明可以有多次**，你可以在很多不同的位置告诉编译器“在某处有这个东西”，但**定义有且只能有一次**，如果编译器找遍了你所有的代码发现没有任何地方定义了 int a，那么会报错“无法解析的外部符号”，如果去掉了 extern 在有重叠的作用域里定义了多次，也会报错“找到一个多重定义的符号”。

10. 坚持：会当凌绝顶，一览众山小！

——————————————（所有内容到此结束）——————————————