梅河口市第五中学信息学竞赛培训材料

第 6 次笔记

对应章节：Chapter 5

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

我不得不承认，前四章的学习之后，竞赛题目已经可以做了，如果运用得当，这些技术已经足够我们写大量程序。然而，现有的知识体系里总有那么几个“神奇”的地方，我们似乎懂了，又似乎没懂。

苏轼有言“**博观而约取，厚积而薄发**”。我想起大学线性代数课，期末考试要考的是 Jordan 标准型的算法，很简单的算法，几个小时就可以学会，然而这门课程花了近一个学期的时间从多项式理论到空间分解理论等等作了各种天花乱坠的铺垫，**它们和最后的考试看起来无关，却承担着“解释这个超级简单的算法是怎么来的”的使命**。我们即将开始学习的竞赛教材里，作者极力避免指针的使用，他也确实做到了，然而这样做有利也有弊。可以这样说，有了指针程序才有了灵性，从教材第五章页数是最多的这一点足以看出，指针对于 C 语言有多么重要。

然而，竞赛教材的处理方式并没有错误，指针用好了会带来极大的方便，**用不好则是“灾难性”的**，对指针理解的深度可以看出一个程序员的 C 语言功底，我身边也有这样的同学，他们在今天发现的一些问题事实上是当年没学好指针造成的。

综上我们得出结论：**指针必须学。而且要学好。**

因此我**强烈建议你更为细心地阅读教材和笔记，并留出一些时间进行复习和联想**，从而运用这一章的知识，彻底理解之前不敢触碰的神奇的地方。

不必阅读的部分：5.10、5.11（只需要读第 104 页）、5.12、5.6（替换为笔记第 7 条）

2. 运算符优先级的再次强调

我曾经写过，我们没有精力和必要精确掌握所有的运算符优先级，而且依赖于优先级的表达式会给代码的阅读、理解和维护带来相当大的麻烦，因此需要区分优先级的语句里请尽可能带上圆括号，毕竟圆括号优先级最高，多加了也无所谓。教材第 80 页下半部分写了这样几条语句：

```
*ip = *ip + 1;
*ip += 1;
++*ip;
(*ip)++;
```

这四条语句都表示使指针指向的对象进行自增操作，是等价的，但最后一条语句就必须加括号使*运算符正确地结合到 ip 上，类似的还有函数指针的声明和使用，加括号则是函数指针，不加括号则是指针函数，需要引起重视。

3. 指针声明

现将一些简单声明语句的含义列举如下，请参考教材进行理解：

声明语句	含义
`int i;`	定义整型变量 i
`int *p;`	定义指针 p，p 指向整型变量
`int **p;`	定义指针 p，p 指向*指向整型变量的指针*
`int a[n];`	定义整型数组 a，a 有 n 个元素
`int *p[n];`	定义指针数组 p，p 有 n 个*指向整型变量的指针*作为元素
`int (*p)[n];`	定义指针 p，p 指向含 n 个元素的一维整型数组
`int f();`	声明函数 f，f 参数列表为空且返回整型值

`int *f();`	声明函数 f，f 参数列表为空且返回一个指针，该指针指向整型变量
`int (*pf)();`	定义指针 pf，pf 指向函数，该函数参数列表为空且返回整型值
`int (*pf[])();`	定义数组 pf，pf 有 n 个*指向参数列表为空且返回整型值的函数的指针*为元素

其实看到这里之后，5.12 便没有必要再看了，复杂声明，再复杂也是会符合语法规则的，见招拆招即可，比如下面这条声明：

```
char *(*(*Func(char *))[])(char *);
```

看起来超级复杂，事实上，它声明了一个名为 Func 的函数，该函数有一个指向 char 型变量的指针作为参数，返回一个指针，该指针指向一个一维数组，该一维数组的元素为指针，这些指针指向函数，这些函数的参数为指向 char 型变量的指针，返回值是一个指向 char 型变量的指针。
返回值如此纠结的函数基本不会遇到，请放心。

4. 指针的算术运算

编程的世界里“1＋1≠2”的情况，除了数据越界或丧失精度，就是指针运算，这种特殊之处尤其需要理解透彻。现在我们参考一下“指针”这个概念的最原始定义：

A pointer is a variable that contains the address of a variable.（p.93 Line 1）

指针是一种保存变量地址的变量。（p.79 第一行）

进一步翻译，**指针是一种变量类型**，它的值是一个地址。什么是变量（Variable）？**变量要有四个基本属性：名字，类型，一定大小的存储单元，值**。指针是一种变量类型，也就是说指针的存储单元的大小就是固定的，我们在最初几讲里讲过 sizeof 运算符，可以用它来验证我们的认知是否正确（请自行完成），在 32 位编译器生成的程序中 sizeof(void *)的结果为 4（个字节），在 64 位编译器生成的程序中 sizeof(void *)的结果为 8。这个结果的区别表明指针变量在内存中被分配的空间大小有区别，也就是说它能表示的数据范围就有区别，于是我们就可以解释 32 位操作系统和 64 位操作系统的区别之一——前者最大支持 4GB RAM。

那么为什么一种变量有上面表格里天花乱坠的形式呢？因为——虽然这个变量很简单，在固定大小的存储单元里存了一个数表示地址，但是这个地址“指向”的是什么东西是不一定的——**同一个内存区域可以有多种解析方式**（这次笔记的很多部分都在讲解这句话）。因此指针变量指向的变量的类型对指针变量而言至关重要，因为它是程序“理解”那块内存的意义的方式（**也即，*运算符的返回值类型**）。好我们终于可以解释指针的算术运算这个问题——指针的加减**（偏移）**是以它指向的变量的类型的存储单元大小为单位进行偏移的，也只有这样该算术运算才是有意义的。

如果此时你还不确定 int、long、short、double、float、char、long long 等基本数据类型在内存中存储单元的大小（字节数），那么你应该赶紧做实验、翻书、百度或者请教别人了。

如果你是 ISO C/C++标准的制定者，请问，void *类型的加减运算应该怎样界定？
我们可以写一段代码测试一下这个问题（忽略了无关部分）：

```
void *p = NULL;
p++;
p += 1;
```

在 Visual Studio 2013 下编译，会发现带有这段代码的程序连编译都通不过，提示的错误信息是“error C2036：“void *”：未知的大小”，（Dev-C++ 64-bit 编译通过并可以运行，但会有两条警告“[Warning] ISO C++ forbids incrementing a pointer of type 'void*' [-Wpointer-arith]”、“[Warning] pointer of type 'void *' used in arithmetic [-Wpointer-arith]”。也就是说，进行指针算术运算的指针必须“指向完整对象类型”，从而让编译器推断出指针变量的值应该如何变化。至此 5.4 节的内容应该可以理解了。

5. 指针与数组，指针与函数参数

现在的你应该具备了小试牛刀的能力，请看下面一段代码，假设它在 32 位环境下编译运行，sizeof(int)值为 4，sizeof(void *)值为 4。

```
#include <stdio.h>
```

```
void foo(int p[4])
{
    printf("0x%X\n%d\n", p + 1, sizeof(p));
}

int main()
{
    int val[4] = {0};
    printf("0x%X\n%d\n", val, sizeof(val));
    foo(val);
    return 0;
}
```

已知程序输出的第一行是 0x34FACC，请独立思考并写出后三行输出结果。

答案：

16、0x34FAD0、4

教材里讲过，val（数组名）不是一个变量，但它可以把数组起始地址赋给指针，由于它的特殊属性，sizeof(val)的结果是 4 * sizeof(int)也就是数组在内存中占用空间的总大小。因此**判断数组个数也可以用 sizeof(val) / sizeof(int)**进行。故第二行答案为 16。

第三行为第一行值加 4，上一条笔记已经解释过。参数为 int p[]、int *p、int p[4]事实上是等价的，p 在 foo 内部就是一个 int *型变量，那么它对应的算术加法加 1 会被放缩到加 sizeof(int)也就是 4。第三行答案为 4 也是显然的，因为 sizeof(int *)结果为 4。

请注意，如果写成 int (*p)[4]，则结果会不同，严格的编译器甚至不允许 val 给它赋值，因为 int *与 int (*)[]不是同一种数据类型，**int (*p)[4]与 int p[][4]是等价的，事实上 p 是二重指针**。

我们讲过传值调用这一机制，通过指针间接修改函数外变量得益于指针的寻址操作，然而在发生参数传递的时候指针还是会拷贝一份（每个函数都有自己的栈帧，栈帧中保存了自己独有的一套参数和局部变量，上次讲过），所以指针与传值调用并不矛盾，指针也是变量也要遵守。

由此可见，在 C 语言的体系下，在函数中无法通过 sizeof()等方式获得数组的大小，只有到 C++引用机制出现后才可以在 foo 函数中得到 main 中一样的结果。因此，我们讲过传递数组的方式为：

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

其中参数一为数组首地址指针，参数二为数组长度。对于字符串，可以这样传递：

```
void Function(char *str);
```

之所以可以减少一个参数是因为（C 语言中）**一切合法的字符串都以'\0'表示结束**，所以我们不需要额外提供数组长度信息，可以在函数内部自行判断出来。

6. 指针与常量的关系

我们知道，const 修饰符表明声明的对象为常量，也即，只能被初始化一次，不能再被修改（更专业地说，**不能作为左值**(详见教材 p.172 下方 A.5 对象和左值)），例如下面的声明：

```
const int a = 0;
```

上面这句话声明了一个整型常量 a，初始值也是整个生命周期的值为 0，如果将它写为：

```
int const a = 0;
```

效果等价，编译器会给出同样的解释。这类数据效用上类似于宏定义但它们有根本区别——宏定义只是简单的替换，在生成代码之前替换掉源文件中的相应字符，而常量是不能作左值的变量，它有变量的基本属性，**在内存中有自己的地址**，但该地址中的内容是“只读”的，也就是起到了保护数据的作用。

根据上面一条我们知道，指针有两个属性，一个是自身的属性，一个是指向的变量的属性，因此指针与常量就有三种关系：指针常量，常量指针，常量指针常量。

a) 指针常量

与“整型常量”相同，指针常量就是不能作左值的指针，也就是说，整个生命周期里值保持不变，能且只能初始化一次，它可以指向任何类型的变量，不管是不是常量。如：

```
int * const ptr = NULL;
```

将声明一个指针常量，在它的一生中任何作左值的操作都是不可以的（意即尝试修改它的值）。

b) 常量指针

首先，常量指针不是指针常量，所以它可以作左值，可以指向不同的对象。形如：

```
const int *ptr = NULL;
```

将声明一个常量指针，**虽然 ptr 可以作左值，但是*ptr 不可以作左值**，也就是说由 ptr 进行寻址时，内存是只读（Read Only）的。

一般在函数参数列表里声明为 const 指针（常量指针），从而明确表示该函数内部不会修改该指针指向的数据（毕竟，传递指针让我们得以修改任意内存，但有时也不是我们希望的）。

c) 常量指针常量

它的名字已经明确了它的性质，它是指针常量，同时是常量指针，形如：

```
const int * const ptr = NULL;
```

将声明一个常量指针常量，ptr 不能作左值，*ptr 也不能作左值，它一生能且只能初始化一次且指向的内容是只读的。

囿于汉语的表达能力，你可能认为常量不常量是在咬文嚼字，但事实上**不同的限定将影响到访问内存的权限**，机器内部对待它们的方式是明确和严格的。遗憾的是，C 语言允许绝大多数隐式类型转换，在常量上的错误有时不可见，我们不评价这样做合理与否，然而 C++禁止绝大多数隐式类型转换，在常量上的差池是通不过编译的，因此还是要在开始学习的时候就引起重视。

7. 指针威力小探——客观对象的不同解析

我们专门用一个例子探讨上面出现过的一个观点：同一个内存区域可以有多种解析方式。C 语言中的指针可以相互进行类型转换（这里说的类型是指指针指向的数据类型），也提供了 void *表示任意指针（可以作为转换的媒介），考虑这样一个例子：

如何查看某变量在内存区域中的二进制表示？

分析一下例子，回忆进制转换以及第 3 次笔记中关于原码、反码和补码的内容，我们有一个简单的思路——把整数通过进制转换和补码转换即可得到内存中的二进制表示。然而，这个思路实现起来是麻烦的，况且，如果需要查看浮点数在内存中的二进制码，这样的进制转换根本行不通。可见没有指针的世界里程序的通用性会受到限制。

再思考一下，内存中的二进制码是客观存在的，只要我们知道要查看的变量的位置（地址），按“数”的方式“理解”那段内存，就总有办法获得其二进制表示（移位操作、0/1 判断等）。

我们知道，char 类型通常占用一个字节（8 位），那么我们可以写一个函数，它有两个参数分别为指向变量的指针和变量在内存中占用的字节数，函数功能是**从高到低**（方便查看）打印出每个字节的二进制数据，手段是把每个字节的数据拷贝一份继而分析这份副本，代码示例——

```
void PrintBinary(const void *ptr, int len)
{
    if (len < 1)
        return;
    const char *tmp = (const char *)ptr + len - 1;
    char val, loop;
    for (; len > 0; len--)
    {
        val = *tmp--;
        for (loop = 0; loop < 8; loop++)
        {
            putchar(val < 0 ? '1' : '0');
```

```
            val <<= 1;
        }
        putchar(' ');
    }
    putchar('\n');
}
```

这里采用了比较简单的方式也就是左移判断符号（有符号类型的最高位是符号位，0 表示非负，1 表示负），你还可以用学过的多种其他位运算方式实现该函数。

进而我们可以写测试程序测试这个函数，它的使用方法非常简单，对于任何类型的变量 val，只需调用**PrintBinary(&val, sizeof(val));**即可。

这个小函数还是可以极大地方便我们的学习的，你可以用它反向验证我曾经讲过的补码内容，现在我们探索些高端的东西。我曾说过 double 类型非常无奈地有正负两个 0，现在我们可以写如下代码——

```
#define PB(val) PrintBinary(&val, sizeof(val))

int main()
{
    double one = 1.0, pZero = 0.0, nZero = -0.0, result;
    PB(pZero);
    PB(nZero);
    result = pZero / pZero;
    printf("0.0 / 0.0 gives %llf\n", result);
    PB(result);
    result = one / pZero;
    printf("1.0 / 0.0 gives %llf\n", result);
    PB(result);
    result = one / nZero;
    printf("1.0 / -0.0 gives %llf\n", result);
    PB(result);
    return 0;
}
```

它产生了如下输出：

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0.0 / 0.0 gives -1.#IND00
11111111 11111000 00000000 00000000 00000000 00000000 00000000 00000000
1.0 / 0.0 gives 1.#INF00
01111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
1.0 / -0.0 gives -1.#INF00
11111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
```

double 的二进制表示比想象的复杂一些，有兴趣的同学可以自行查阅百科资料，深入探究已无必要，但上面常识性的知识还是说明一下。前两行体现出了正负两个零这一点，double 的负数是靠最高位符号位进行标识的。对于 0.0 除以 0.0，如果我们用文字常量来写，那么编译器会发现并报错，0 禁止用作除数，但变量是否为 0 编译器检查不到，因此可以运行起来。我们知道，0.0 除以 0.0 在数学上是没有定义的，但在计算机上进行这种计算不会引起崩溃（甚至，在 C++的异常机制中都捕获不到），CPU 计算后会生成一个特殊的“非数”，也就是上面写的**-1.#IND00**，它在内存中的二进制表示对应着一种“不可能”情况。用 1.0 分别除以 0.0 和 - 0.0 得到的结果为正负无穷，它们的二进制码如上。

正如物理既有理论又有实验，编程也是如此，我们演示了很多次如何设计程序双向验证知识，遇到不懂的问题，用懂的方式辅助攻破是一个很好的习惯~

8. 指针数组、多重指针与堆上的动态数组分配（替换教材 5.6 小节）

教材 5.6 节介绍的内容非常好，但已经好得超过了我们的需求，毕竟我们的目的不是在语言上纠缠太多。所以我们重写这一部分并将堆上的内存管理提前介绍给大家。

考虑这样一个问题：

如何接受用户任意行（不超过 100000，在第一行给出），每行任意长字符（不超过 9999）的字符串输入，将每一行按字典序排序，并从小到大重新输出到屏幕上？

再好的排序算法也要有交换操作，倘若几百行数据每行都有几百个字符，通过交换数组的方式（开一个新的临时数组，像交换两个变量一样遍历数组三次）进行排序的话将是灾难性的。因此我们有必要采用指针，**通过交换指针而不是交换数组提升运行效率**。

除时间开销外还有空间开销，对于此题，我们为了保存输入数据显然需要字符串的数组（也即数组的数组，二维数组），如果输入是最复杂的情况，我们需要 10 万个数组，每个数组的长度是 1000，一共 1 亿个元素，即 char [100000][1000];，但如果输入是很简单的情况，比如只有一行一个字符，那么我们将有数千万空间被浪费（事实上在栈上根本无法开辟近百 MB 的空间）。因此我们希望“动态”地（在编译期不确定地）创建数组，然而，在 C 语言里这样写（忽略 main()函数其他部分）：

```
int n;
scanf("%d", &n);
int a[n];
```

是无法通过编译的，因为 C 语言规定数组的大小必须是在编译期可以确定的常量。在如今的 C++标准中这种做法才被接受。那么想动态地分配内存，需要掌握两个函数——malloc()和 free()，原型分别是：

```
void *malloc(size_t size);
void free(void * Memory);
```

其中 size_t 类型依赖于编译器和操作系统，一般情况下在 32 位机器中为 4 字节无符号整数，在 64 位机器中为 8 字节无符号整数。它们的功能分别是，malloc 遍历计算机内存信息链表，寻找长度不小于 size 的可用空闲内存，如果找到，将这段内存标记为已占用并返回其地址，free 将把参数指针指向的内存区域回收留给以后使用。上一讲我们介绍堆栈时提到过，堆上的内存操作一般依赖程序员，如果 malloc 后没有 free 且操作系统没有作出处理，相应的内存区域将一直被占用无法回收，这就导致了“内存泄漏”的发生。因此，使用这两个函数前一定要注意，**它们是成对使用的**。

运用它们我们可以写出需要的动态数组代码：

```
int n, *a;
scanf("%d", &n);
a = (int *)malloc(n * sizeof(int));
...
free(a);
```

除此之外，库函数里还提供了 gets()函数用来接收字符串，它与 scanf()接收字符串的区别在于，scanf()以空白字符结束一个字符串的接收，而 **gets()只以回车为边界**（也就是可以接受空格、制表符等），同时，它们都会在字符串末尾自动补'\0'，换行符'\n'也都不会出现在字符串中。不过**程序员有义务保证传递给这两个函数的指针参数指向的数组是足够大的**，如果不够，它们不会重新分配空间。

输出字符串使用 printf()或 puts()函数均可。

至此，输入输出和数据存储都已经有了解决方案，最后的部分就是排序了。我们追求的除了程序的运行效率外，还有编程成本和调试成本，现场写快速排序稍有闪失是要付出代价的。非常幸福的一件事情是，库<stdlib.h>里已经提供了 qsort()即快速排序函数，它的原型如下：

```
void qsort(void *Base, size_t NumOfElements, size_t SizeOfElements,
    int (*PtFuncCompare)(const void *, const void *));
```

不要紧张，声明看着长，只是字母长了点而已。能进到库里的函数，是尽可能“通用”的，快排就做到了这一点，Base 指向待排序数组的起始地址（而不管数组元素的类型，甚至可以是指针数组、多重指针，只要自定义了比较函数就可以，这就是指针的神奇之处），参数二是元素的个数，参数三是单个元素占用的字节数（推荐采用 sizeof(Base[0])的写法），参数四是函数指针（阅读教材第 104 页），该函数必须返回整型值，正值表示目标关系（比如升序排序则参数一比参数二大时返回正值），负值相反，零值表示相等，这个函数是自定义的，从而使 qsort 可以针对各种情况。

函数指针的使用和指针非常相似，只是优先级上必须用圆括号区分开来。教材第 128 页讲解 typedef 功能时提到，下面这种表述：

```
typedef int (*PFC)(const void *, const void *);
```

声明了一种“类型”PFC，PFC 类型的变量为函数指针，该函数返回整型值，以两个常量指针作为参数。这样的话，qsort 的声明就可以简写为：

```
void qsort(void *Base, size_t NumOfElements, size_t SizeOfElements,
    PFC PtFuncCompare);
```

实际使用时并不受影响。教材 5.11 节有用的地方都在 104 页，掌握了即可。

于是我们需要用一个函数包装起 strcmp 来作为 qsort 的参数，问题也就解决了，参考代码：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_LINE 1000

int cmp(const void *a, const void *b)
{
    return strcmp(*(const char **)a, *(const char **)b);
}

int main()
{
    int lines;
    int i;
    char **src;
    char buf[MAX_LINE];
    scanf("%d", &lines);
    src = (char **)malloc(lines * sizeof(char *));
    getchar();        //跳过换行符
    for (i = 0; i < lines; i++)
    {
        gets(buf);
        src[i] = (char *)malloc((strlen(buf) + 1) * sizeof(char));
        strcpy(src[i], buf);
    }
    qsort(src, lines, sizeof(src[0]), cmp);
    for (i = 0; i < lines; i++)
    {
        puts(src[i]);
        free(src[i]);
    }
    free(src);
    return 0;
}
```

上面代码中主体部分都还好理解，cmp 函数中的类型转换可能比较晦涩。在此说明一点，qsort()库函数在使用函数指针时，**传递给函数的参数是指向需要比较的两个元素的指针（想想看，为什么不是传递变量的拷贝？）**，我们需要比较的两个元素是字符串，换言之，strcmp 库函数的参数是字符串首地址的指针，那么它的指针就是二重指针，因此我们需要把 void *转换为 char **然后再用*运算符寻址得到想要的指针进行比较。

我们写的函数，容易出错，效率也不见得比库函数高，C 的库非常精悍，但为了保证通用，它用到了一些更高的技术，我们理解好了，才能愉快地用它们。

既然库函数里有排序，为什么我还长篇累牍地讲排序？

——首先，排序算法对于各种概念的理解有极大帮助；另外，库尽力通用，然而万一我们的情境不能使用它，**必须要有自己写的能力**，实现这个能力不要靠背代码，理解思想才是最关键的。

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