梅河口市第五中学信息学竞赛培训材料

第 5 次笔记

对应章节：Chapter 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

继上一章详细研究了控制流之后，我们掌握了一个函数(比如 main 函数)内部的写法，而本章内容是函数与程序结构，也就是掌握一个程序的设计方法。人类文明发展到今天，很重要的一个 idea 是**“复用”**，上次笔记研究了三种排序方法，我们之所以写成函数就是为了在需要的地方随时可以用这种方式进行排序，而不是哪里需要就在哪里重新写一遍，试想，如果 printf()等我们用到手软的函数都需要自己展开写的话，将是多么恐怖的一件事情！

得益于 C 语言的这一特性，代码以函数为单位得到了极大复用，因而人类构造出了更大规模的程序。也因此我们说 C 是面向过程的，专注于单入单出的程序设计，而 C++（这个命名是怎么来的？还记得自增运算符吗？）相比 C 语言，将数据和方法（典型地，函数）进一步封装到一起成为类，使类成为程序复用的基本单位，这让我们得以构造更更大规模的程序，也因此我们说 C++是面向对象的。

目前的阶段，尽可能培养良好的直觉，清晰地设计程序。

不必阅读的部分：第 60 页第三行到 4.1 末尾

粗读部分：4.5、4.7、4.11.2、4.11.3

2. 栈溢出（Stack Overflow）

本章提到了这个概念，我们简单解释一下（不需要彻底理解，知道这个事情就可以）。

程序在运行过程中，为了临时存取数据的需要，一般都要分配一些内存空间，通常称这些空间为缓冲区。由于 C 语言系列没有内置检查机制来确保复制到缓冲区的数据不得大于缓冲区的大小，因此当这个数据足够大的时候，将会溢出缓冲区的范围。由于缓冲区溢出而使得有用的存储单元被改写,往往会引发不可预料的后果，包括但不限于程序崩溃。**栈溢出是缓冲区溢出的一种。**

那么为了继续解释就要说明白堆栈之类的到底是什么……

一个由 C/C++编译的程序占用的内存分为以下几个部分：

***栈区（Stack）**——由编译器自动分配释放，存放函数、函数的参数值，局部变量的值等。

***堆区（Heap）**——一般由程序员分配释放，若程序员不释放，程序结束时**可能**由 OS 回收。

***全局区/静态区（Static）**——初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

***文字常量区**——比如字符串常量，程序结束后由系统释放。

***程序代码区**——存放程序的二进制代码，函数（非内联）无论调用次数只有一套，用到时压进栈中。程序结束后由系统释放。

起码在 Windows 系统中，**栈的分配是编译期即确定的一个常数**（通常为 1M，有说 2M，没较真过），堆有多大看可用的剩余内存有多大，它的分配方式类似链表（不急，下节就讲）。总之，栈的分配程序员在运行时无法干预，堆的分配几乎全依赖程序员的设计（也就产生了新概念——内存泄漏(Memory Leak)，指的是在堆上申请了内存而没有释放导致无法使用）。因此**栈溢出是危险的。**

什么样的操作会导致栈溢出呢？我们知道，栈存储着函数的相关信息，如果一个函数被调用了 5000 次以上（只是估计值），那么 1M 的空间已经存储不下这么多函数的信息了（表面上你调用了哪怕同名的函数，在内存中它的地址也是互不相干的，否则怎么区分），会爆栈。另一种情况是，在局部变量中使用超大规模数组，这样的数组也是要分配在栈上的，典型的解决方案有两种：**分配在堆上**（以后讲）或者**定义为全局变量**。如果有容易爆栈的情况请慎重权衡。

然而堆栈作为数据结构的概念和上面是不相同的，我们都会介绍到。

3. 函数的返回值

教材第 59 页中部“最简单的函数”在今天是不被允许的，第一周笔记关于 main()函数的写法也曾提到过，默认 int 在今天是不可以的，C++对此规定更为严格。因此在写函数声明和定义的时候，如果不返回值请写 void，否则按返回值类型显式声明。函数参数列表如果为空的话既可以写 void 也可以写()。另外，void 函数也可以有 return 语句，只是 return 后没有表达式而已。

教材第 59 页最下方提到了一种“出问题的征兆”，这样的函数在某些编译器进行编译时会抛出一条警告“函数 xxx：不是所有的控件路径都返回值”，如 if-else if-else 语句的某一条没有返回值，我们知道，if-else 并列起来至多执行一条语句（如果没有最终 else 则可能不执行），那么执行到没有返回值的语句函数将返回一个“意外的”值，因此在设计函数时请注意返回值和返回的位置是否符合预期。

4. 静态（static）变量

教材 4.6 节可能有点晦涩，关于静态变量需要掌握几点就够了：

*非函数的局部变量，作用域从定义处到.c 或.cpp 文件末尾，否则作用域在函数内部；

*生命周期与整体程序相同；

*初始化且只初始化一次，在定义处，再次遇到定义时会跳过。

其实语义都包含在 4.6 节的文字中。一个典型的应用是保存与函数相关的信息，比如函数被调用的次数或者上一次函数调用时确定的某个值，因为这种变量区别于自动变量的特性是生命周期长，函数出栈后仍保留数值和存储单元（由笔记第 2 条，它们根本不在一个存储区域里）。

5. 多文件编译（选读）

这部分内容在多处出现过，并不需要掌握因为竞赛时一般只写一个文件，以备不时之需介绍如下。

关于声明与定义的区别，请参考第 2 周笔记第 9 条，主要区别是分配或不分配存储空间。多文件编译与单文件编译相比只多了一步“链接”的操作，这就需要我们在头文件中“告诉”链接器有哪些东西是定义在“某个地方”需要寻找并链接起来的。同时，头文件有可能出现重复包含的情况，解决如下：

文件 ExFunc.h 代码：

```
#ifndef __ExFunc__HeaderFile__
#define __ExFunc__HeaderFile__

#include <stdio.h>

extern int foo();

#endif
```

文件 ExFunc.c 代码：

```
#include <ExFunc.h>

int foo()
{
    return 0;
}
```

文件 main.c 代码：

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

int main()
{
    printf("%d\n", foo());
    return 0;
}
```

这样做 main.c 文件就会很清爽，在大工程和代码复用中尤为有用。对竞赛用处不大。

#ifndef…#endif 和#ifdef…#endif 称为预编译指令，将在编译时进行判断，也称条件编译。竞赛时为了方便调试起见可能要输出中间结果，但删除起来又比较麻烦，那么可以这样做：

```
#include <stdio.h>

#define __Local
//#undef __Local

int main()
{
#ifdef __Local
    printf("...");//Debug info
#endif
    return 0;
}
```

然后在提交程序的时候把#undef __Local 这条语句的注释符号删掉，就不会有问题了。当然手动删也可以，宏定义这种东西就是这个样子ヽ(ノ ▽ ヽ) ノ。

6. 栈与队列（数据结构）

现在我们学习两种新的数据结构：栈（stack）、队列（queue）。所谓栈，就是符合先进后出（Last In First Out）规则的数据结构，有 push 和 pop 两种操作，push 为把元素压入“栈顶”，pop 为从栈顶把元素“弹出”。而队列是符合先进先出（First In First Out）规则的数据结构，可以想象成栈是一端封口的杯状结构，队列是两端开口的管状结构。它们都属于 C++中 STL 的一部分，后期会学，然而 C 语言实现栈也是没有问题的，教材 64 至 66 页给出了详细代码，其中用到了自增自减运算符如有问题请复习第二章。总体思想是，**用数组存储元素，但是额外使用一个变量标记空闲栈的位置，push 一次，则数组末尾被赋值，标记变量自增，pop 一次，标记变量自减，返回栈顶元素的值，也即做到了先进后出。**

一个用来区分程序员和普通人的小笑话是，push 的反义词是什么？回答 pull，普通人无疑，程序员一般都回答 pop……

7. 递归算法初步

现在我们已经接触到了算法竞赛中看似神圣的一大部分——递归，当然，这次是初步介绍…

其实递归你已经学过了，它**和数学归纳法很相似**。比如我们这样定义“正整数”：

（1）1 是正整数；

（2）如果 n 是正整数，则 $n+1$ 是正整数；

（3）正整数集是满足（1）、（2）的最小集（换言之，满足(1)、(2)的是正整数）。

这种方式就是“递归定义”，因为枚举出来所有情况进行定义几乎是不可能的工作量。所以我们现在就可以给出“递归”自身的定义——

递归：

参见“递归”

什么？这™不是什么都没说吗？！之所以给你这个感觉，因为这样的定义显然是“无限”的，参见“递归”可以无止境地“参见”下去，这就带来**写递归的重要原则——终止条件**。

回想，我们写程序的时候 return 语句被称为“返回”语句，它的作用是“终止”函数并交给主调者一个值（的拷贝），它标志着函数的“退出”，当遇到 return 语句的时候，函数后面有没有语句都不会被执行到，反之，没有遇到 return 语句（或返回 void 的函数的末端花括号，或 C++中非 nothrow 的函数中的 throw 语句）前，函数就要一直执行，等所有的语句都结束（函数都返回）之后自己才能退出去。

这次笔记第 2 条对栈已经作了铺垫，上面这段文字翻译成 B 格更高的版本也就是：递归调用的每个函数（无论名字相同否）都有自己的**栈帧（Stack Frame）**，调用函数则函数入栈，函数返回则出栈，栈帧销毁。栈帧里保存了什么信息？函数执行的环境——函数参数、函数的局部变量、函数执行完后返回到哪里等等，也就是说，**递归调用的函数有自己的一套参数、局部变量和返回地址**。

现在我们举一个栗子——皇上想计算 3 的阶乘，于是：

皇上：丞相，你给朕算一下 3 的阶乘！

丞相（知道结果是 3 乘 2 的阶乘）：知府，你给我算一下 2 的阶乘！

知府（知道结果是 2 乘 1 的阶乘）：县令，你给我算一下 1 的阶乘！
县令（知道结果是 1 乘 0 的阶乘）：师爷，你给我算一下 0 的阶乘！
师爷（已知 0 的阶乘是 1）：回老爷，0 的阶乘是 1。
县令（心算 1×1）：回知府，1 的阶乘是 1。
知府（心算 2×1）：回丞相，2 的阶乘是 2。
丞相（心算 3×2）：回皇上，3 的阶乘是 6。
皇上解决问题了。
刚才的过程，我们将“某官员计算 n 的阶乘”抽象为一个函数，那么这个函数长成这个样子：

```
int Fact(int n)
{
    if (n == 0)
        return 1;
    return n * Fact(n - 1);
}
```

而上述的皇上、丞相、知府、县令、师爷分别对应着 main()、Fact(3)、Fact(2)、Fact(1)、Fact(0)的栈帧，这种递归是有明确边界的即 n 为 0 时返回 1。根据刚才的理论解释，return n * Fact(n – 1);这条语句会等待 Fact(n – 1)执行完并返回值之后才返回值，也就是上面比喻的情形。
这么多段落其实就是说一句话——
递归，就是一个函数直接或间接调用自身（教材 p.73 最下方）。
根据结构化程序设计的基本原则，一切递归的程序都可以改写成非递归的版本，然而我们不能确定递归与不递归哪种效率更高，也不确定哪种写法设计和调试的时间成本更低。总体来看，递归能让算法设计更接近人类的思维方式，算法竞赛中经常考察。

8. 排序算法（续）

d) 快速排序

递归是为了做什么呢？这里我们隆重推出重要的算法策略——**分治算法（Divide and Conquer）：将一个规模为 n 的问题分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题相同。递归地解这些子问题，然后将各子问题的解合并从而得到原问题的解。**
历史经验表明，善于大事化小的人做事情是游刃有余的，程序也是一样。上节学过的二分查找就是一个分治的例子，现在我们来讲快速排序，快速排序是对冒泡排序的一种改进，是常用排序算法里最快的，但在特殊情况下会退化为冒泡排序，快速排序的优化以后再研究。
它的基本思想是：**通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序**，整个排序过程可以递归进行，以此达到整个数据变成有序序列。
也就是说，选取一个键值 key，用它将序列分为两部分，再对左右进行快排，递归直到序列长度小于 2（天然有序）停止。
再次强调，**不要背诵代码**，掌握原理后写起来并不难：

```
void QuickSort(int val[], int low, int high)
{
    if (low >= high)
        return;
    int first = low, last = high;
    int key = val[first];
    while (first < last)
    {
        while (first < last && val[last] >= key)
            last--;
        val[first] = val[last];
        while (first < last && val[first] <= key)
            first++;
        val[last] = val[first];
```

```
    }
    val[first] = key;
    QuickSort(val, low, first - 1);
    QuickSort(val, first + 1, high);
}
```

仍然用数组{4, 5, 3, 1, 8, 2}进行测试，那么每次交换两数后数组是这个样子：

```
2 5 3 1 8 5
2 1 3 1 8 5 （枢轴记录后 2 1 3 4 8 5）
1 1 3 4 8 5 （枢轴记录后 1 2 3 4 8 5）
1 2 3 4 5 5 （枢轴记录后 1 2 3 4 5 8）
```

赋值操作少了这么多！可见快速排序的威力！需要注意的是，上面代码已经比较简单，不推荐教材第 74 页的写法，尽管不同人写有不同的风格，但上面出现的“重复”的判断以及 val[…]与 key 的判断中的等号却是不可以修改的，想想看并设计一些测试案例进行测试，为什么？

e) 归并排序

归并排序是分治法的另一种典型应用，在效率上比快速排序略逊一筹，空间消耗较大。它的整体思想是，把原始数组分成若干子数组，对每个子数组进行排序，而后将子数组与子数组归并（两个顺序序列合并成一个顺序序列的操作）得到有序数组，重复直至全部合并。反过来看，它每次都将待排序数组分割为左右两部分，然后对两个子数组分别排序，再将有序的子数组合并起来，递归进行直到最终有序，这可以从下面的函数看出来：

```
void MergeSort(int src[], int tmp[], int first, int last)
{
    if (first < last)
    {
        int mid = (first + last) / 2;
        MergeSort(src, tmp, first, mid);
        MergeSort(src, tmp, mid + 1, last);
        Merge(src, tmp, first, mid, last);
    }
}
```

排序主体函数非常清晰，递归进行，辅助函数如下：

```
void Merge(int src[], int tmp[], int first, int mid, int last)
{
    int i = first, j = mid + 1, k = first;
    while (i <= mid && j <= last)
        tmp[k++] = src[i] < src[j] ? src[i++] : src[j++];
    while (i <= mid)
        tmp[k++] = src[i++];
    while (j <= last)
        tmp[k++] = src[j++];
    for (i = first; i <= last; i++)
        src[i] = tmp[i];
}
```

用数组{4, 5, 3, 8, 1, 2}进行测试，那么每次执行归并操作后数组是这个样子：

```
4 5 3 8 1 2
3 4 5 8 1 2
3 4 5 1 8 2
3 4 5 1 2 8
1 2 3 4 5 8
```

独立思考：<u>上面五次归并操作，每次都是谁和谁归并？如果想把数组排成降序，应该修改代码中的哪一部分？</u>

f) 未完待续，多种精彩的排序算法等着你~

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