前言

按照往期的试卷来看，上机题目中并不会出现带有接口的“核心代码”类型题（这种题可以在PTA上的选做二中看到，被称为函数题），而是全部为正常的ACM类型（即写出完整程序），所以以下全部的情况将针对这一类型的题使用。我将为“完全菜鸟”提供一种程序设计的方法，以及为“半熟菜鸟”提供一种debug的方法。

本教程可以保证能解决绝大多数的问题，只要你了解数据类型（bool、int、char、float、double以及string和数组）以及逻辑语句（if-else、for、while），同时知道那么几个输入和输出的方法，但是天有不测风云，也许有意外情况我们无法解决。程序设计是一门需要逻辑的学科，而这里的逻辑往往不是编程语言本身，而是如何将难以理解的复杂问题切割成显而易见的子问题，这个能力是需要具备且难以形容的，虽然我在后续的篇幅会尝试让你们对此有所提升。

最后，对于一切的高手，本教程可能显得稚嫩而无用，我随时欢迎大家的批评指正，但是不要纠结于本文中部分不严谨的表达，假如有任何的想要补充的内容务必告知本人，我会进行添加。

本文中一切的若是还不了解的内容可以问自己的组长，或者在搜索引擎搜索，我可以保证剩余的内容都是一些简单到不能在简单的理解内容。

程序设计

前言

诚然诸如变量的设置此类问题可以随心所欲，语句的执行先后可以稍有不同，代码的规范可以各不相同（毕竟C++主要靠;分割语句），但是对于程序的初学者（100%的那种，对于他们来说可能上课的进度已经难以跟上，面前还在处理的是不需要函数就可以解决的简单编程问题，但是老师已经在讲纯虚与抽象类，甚至部分的他们在指针这一章节就已经掉队），我愿意提出一种在高手眼中不算高明，甚至臃肿的方法，但是可以将我们在PTA考试中面临的大多数问题通过一种程式化的方法解决。

假如说你是一名初学者，对于下面谈到的内容仍然心存疑惑，请先耐心心来接受下面的观点，后续的“附录”中会有一道题作为实战进行逐步的拆解与示范。

设计

在面对一个全新的编程问题时，新手往往会没有思路，而一上来便急于编写代码，对于高手来说他们可能在见到题目的第一刻便有了思路，但是对于大多数人来说往往不会如此轻松，因此急于求成带来的只有混乱。

我们常说“编程语言”，这让我们理解诸如C++一样的奇怪的符号拼凑在一起的格式为某种“语言”，能够清晰的意识到这一点是很重要的。然后我们就会发现那些编程高手就好像我们身边的英语大神一样，这门外语已经可以被运用得如同母语一般，但是作为新手，我们往往还是需要回到那个初学英语时的状态：**“了解自己想说什么，用中文表达出来，翻译成英文。”**

这个过程在编程中被称为书写伪代码，用一种在程序中实际出现的形态来称呼，则可以称之为**写注释**，我们通过自己的理解，将面临的问题拆分成几部分，然后用//的注释用自己能看懂的话写出来。

举一个简单到不能再简单点例子，输入两个值a和b，输出他们的和：

```
int main()
{
 //输入a和b
 //计算a和b的和
 //输出结果
}
```

虽然说我们可以很高兴的宣布我们知道如何输入，如何计算和，但是请不要着急，事实上每一个问题都可以被工整的分为三部分：**输入、解决问题与输出**，而且谁都知道怎么输入和输出。不过中间解决问题的过程往往比较复杂，难以一蹴而就，所以还是先静下心写完注释，不要乱了阵脚，这时候你正在用一种自己能看懂的语言解释将要诞生的程序的运行机制，这无疑会让你对于整体的程序有了一种更宏观的认知。

同时，就像是在前言中提到的一样，我们书写伪代码的过程，便是那个将复杂问题切割成简单问题的过程。

变量

此时我们已经对整体的程序如何运行有了一个大概的想法了，我们愿意看到伪代码是这个样子：

```
int main()
{
 //input
 //step 1
 //step 2
  //step 2-1
  //step 2-2
 //step 3
 ……
 //output
}
```

接下来的步骤不言而喻，我们需要逐步的实现我们用伪代码写下的豪言壮志，为了避免代码的混乱不堪，规定变量的规范是至关重要的，在遵守变量的命名规范的同时，我们将变量分为了四种类型：**输入值、临时值、传递值以及输出值。**

输入值

为了让程序更加脉络清晰，我希望你保持全部的输入的值的名称与题目里的描述一致（如题目中可能提到，输入一个值K，则命名这个变量为K），并且放在main函数的开头，与其他语句之间用换行隔开。

临时值

这是一个需要隆重介绍的概念。在编程的时候除了输入值和输出值之外我们还会创建很多的变量，而我愿意把它们分为两部分，也就是临时值和传递值，不过在介绍之前让我再重申一遍一个概念：程序块，并且在下面的阐述中它会听上去很像一种面向对象的想法。

我们想象这样的一条产业链，负责生产老坛酸菜方便面，其中一个链条是，农场生产小麦，面粉厂接受小麦并且最后产出面粉，面饼厂接受面粉最后产出面饼，方便面厂接受面饼，同时还接收酸菜料包以及包装盒，产出包装好的方便面，最后方便面被送向市场卖给顾客。

这无疑是一种很天才的设想，每一个处在某工厂内的工人只需要完成自己的工厂对应的职责，农民不需要知道怎么用研磨机磨出面粉，市场销售也不需要了解怎么割下小麦，方便面公司不需要某个方便面仙人，对于每一个环节了如指掌，而方便面就可以源源不断的被摆上货架……**每个环节的部门只需要接收它需要的，履行自己的职责，然后产出它应该产出的就足够了**，代码块也是如此。

在设计阶段我们已经把代码分割成了一个个的子问题，这每一个子问题我们就可以认为是一个代码块，**它们接收变量，处理变量，然后再输出变量。**

我们希望看到的是面粉厂公司的员工老老实实的待在面粉厂里，而不是同样出现在面饼厂里，尽管他们不会参互面饼厂的工作，但是假如说面饼厂里有一个人的名字和他们一样，他便很可能被拉去工作，但是他所了解的知识确是怎么做面粉，于是便会导致生产线的损坏。

放在代码中，临时值便是这个面粉厂的员工，我希望它老老实实的待在这个代码块中，所以我们干脆直接在这个代码块中创建这个变量，让它就地生成与销毁。其中的典型是for循环中的i，我们不希望看到i在main函数的开头被定义在了函数内的“全局”，而是在for中被创建，然后再结束for的时候被销毁，不带走一片云彩，如下：

```
for(int i = 0; i < n; i++)
{
 ;
}
```

有的时候一些值无法被销毁（我们暂时不去复习类似于delete和new的操作），我也希望你明确这个概念，手动的让这些值不要跑出去，这些值便被称为临时值，从格式上，我希望你在这个代码块的伪代码下面的第一行及以后去声明这些变量（当然，假如代码块本身是一个for之类的那么更好，我希望你在大括号里面的第一行声明这些变量），同时在之后空一行，一行之后我们将声明那些被称为传递值的变量。

通常我们愿意将临时值命名为temp_function（function指该临时值对应的功能，如temp_cnt，cnt意为count，这是一个计数器）或者i、j、k等。

传递值

正如上面的例子所说，除了员工之外，我们同样需要一些运输产物的卡车，对于代码块来说，它们也需要知道上一个代码块的运行结果如何，这些值穿梭于两个相邻的代码块之间（是的，相邻的代码块，尽管它们事实上可以存在在任何地方，一个值可以贯穿始终，但是为了让代码更加的清晰，我们希望它仅穿梭于两个相邻的代码块），传递结果。

从格式上假如说存在for一类的操作，我希望它们被声明在伪代码与for的起始之间，不然我希望它们位于临时值空一行之后的下面。

同时我们愿意把传递值命名为其对应的功能。

输出值

顾名思义，输出值负责将最终的结果输出，从格式上我希望它们的名字和题目中要求的一致，并且位于输入值空一行之后的下面。

初始化

与此同时，在运行程序的时候我们也希望看到程序不会出现任何在我们意料之外的值，而这么做最好的方法就是在创建这个变量的时候变给予它一个初始值，这个过程我们称之为初始化。

进行了这些操作之后我们很高兴的发现现在的程序已经变成了这样子（以下是一个例子）：

```
int main()
{
    int M = 0, K = 0;
```

```
    int num[10] = {};

    char output[10] = {};

    //input
    cin >> M >> K;
    for(int i = 0; i < M; i++)
    {
        cin >> num[i];
    }

    //step1
    int temp = 0;

    int trans1 = 1;
    //step1的功能实现xxxxx

    //step2
    int trans2 = 2;
    for(int i = 0; i < M; i++)
    {
        int temp_cnt = 0;
        //step2的功能实现xxxxx
    }

    //output
    for(int i = 0; i < M; i++)
    {
        cout << output[i];
    }
}
```

实战

纵使我说了很多，但是在新手的眼中可能感觉都是一些形式主义的东西，让我们找到一道题，然后通过上述的方法进行实施，我会尽力做出讲解。

> 使用结构体数组求10个学生三门课总平均成绩，及最高分学生信息
>
> 学生结构体数组管理10名学生的信息，分别是学号、姓名、3门课的成绩(double型)，从键盘输入10名学生的全部信息，打印出总的平均成绩与最高分学生信息
>
> 输入格式:

> 依次输入10名学生的信息，包括学号char num[6]、姓名char name[8]、三门课分数 double score[3]
>
> 输出格式:
>
> 输出10名学生总的平均成绩，与3门课程总成绩最高的学生分数，所有成绩输出保留小数点后2位小数

首先我们分析这个问题，找到我们需要的信息，我们需要创建一个结构体数组，记录下来十组信息，然后求出平均分以及最高分的学生，于是我们可以写出以下的伪代码。

```
#include <iostream>
using namespace std;
//创建一个学生结构体，其中包括学号，姓名以及三门课的分数
int main()
{
    //声明一个长度为10的学生结构体数组

    //输入信息

    //求出平均分

    //求出最高分的学生

    //输出结果
}
```

事已至此我们能发现其中的一些问题，诸如输入信息或者创建结构体，都是我们已经会的，但是还有一些内容我们虽然写出来了但是依然意义不明，比如说如何求出平均分呢?

以下的这一步是C++程序设计唯一考验的能力，你只需要有能力完成这一步就可以，就是去将一个复杂问题再次拆分成更加简单的子问题。

所以我们设想了这样一种方法，我们先求出来全部的分数的总和，然后最后除以一共的分数份数（30），这样就可以求出来平均分了。

同理，求出最高分的想法也很简单，先记录一个学生的总分以及他的编号（在数组中是第几个，我们一般会选择先记录第0个学生），用这个学生的总分依次和其他学生比较，如果其他的学生的分数比他大，则记录下这个比较大的学生的总分和编号，继续向后比较（没有必要从头比较一遍，因为前面的学生已经比刚才记录的学生要小了，所以现在记录的学生一定比前面的大），最后我们会得到一个编号，这个编号就是最高分的学生的编号。

于是我们可以修改伪代码了:

```
#include <iostream>
using namespace std;
//创建一个学生结构体，其中包括学号，姓名以及三门课的分数
int main()
{
    //声明一个长度为10的学生结构体数组

    //输入信息

    //求出平均分
    //-求出全部的分数的总和
    //-求出平均分

    //求出最高分的学生
    //-记录一个学生的总分和编号
    //-依次进行比较
    //--如果在比较的过程中有人更大，改为记录这个人的总分和编号

    //输出结果
}
```

然后我们创建变量：

（在这里我们有一个讨巧的操作，使用string代替char数组，这样在输入的时候会方便不少，且string创建之后无需初始化）

```
#include <iostream>
using namespace std;
//创建一个学生结构体，其中包括学号，姓名以及三门课的分数
struct student
{
    string num;
    string name;
    double score[3] = {};
};
int main()
{
    //声明一个长度为10的学生结构体数组
    student stu[10] = {};
    //输入信息
    for(int i = 0; i < 10; i++)
    {

    }
    //求出平均分
```

```
    int temp_sumAll = 0;
    for(int i = 0; i < 10; i++)
    {
        //-求出全部的分数的总和
    }
    //-求出平均分
    double averageScore = 0;

    //求出最高分的学生
    //-记录一个学生的总分和编号
    double temp_stuScore = 0;
    int stuNum = 0;
    //-依次进行比较
    for(int i = 0; i < 10; i++)
    {
        //--如果在比较的过程中有人更大，改为记录这个人的总分和编号
    }
    //输出结果
}
```

之后便可以编写主要程序的内容了，此时整体的程序已经被我们分解的成为了简单到不能再简单的子问题了。

```
#include <iostream>
using namespace std;
//创建一个学生结构体，其中包括学号，姓名以及三门课的分数
struct student
{
    string num;
    string name;
    double score[3] = {};
};
int main()
{
    //声明一个长度为10的学生结构体数组
    student stu[10] = {};
    //输入信息
    for(int i = 0; i < 10; i++)
    {
        cin >> stu[i].num;
        cin >> stu[i].name;
        cin >> stu[i].score[0];
        cin >> stu[i].score[1];
        cin >> stu[i].score[2];
    }
```

```
    //求出平均分
    double temp_sumAll = 0;
    for(int i = 0; i < 10; i++)
    {
        //-求出全部的分数的总和
        temp_sumAll += stu[i].score[0];
        temp_sumAll += stu[i].score[1];
        temp_sumAll += stu[i].score[2];
    }
    //-求出平均分
    double averageScore = 0;
    averageScore = temp_sumAll / 30;

    //求出最高分的学生
    //-记录一个学生的总分和编号
    int temp_stuScore = 0;
    int stuNum = 0;
    temp_stuScore = stu[0].score[0] + stu[0].score[1] +
stu[0].score[2];
    stuNum = 0;
    //-依次进行比较
    for(int i = 1; i < 10; i++)
    {
        //--如果在比较的过程中有人更大，改为记录这个人的总分和编号
        if(temp_stuScore < stu[i].score[0] + stu[i].score[1] +
stu[i].score[2])
        {
            temp_stuScore = stu[i].score[0] + stu[i].score[1] +
stu[i].score[2];
            stuNum = i;
        }
    }
    //输出结果
    printf("%.2lf\n", averageScore);
    cout << stu[stuNum].num << " " << stu[stuNum].name << " ";
    printf("%.2lf %.2lf %.2lf", stu[stuNum].score[0],
stu[stuNum].score[1], stu[stuNum].score[2]);
}
```

进阶

在我们看完实例之后可能心中还是存在一些疑惑，虽然最后的代码实现过程不算困难（假如说输出部分有看不懂的见附录部分），但是我们还是不知道程序的生成是如何出现的，为什么在一些奇怪的地方我们要使用for，这一点是怎么想到的。这时候我们需要一些思维的帮助，以及一些小技巧，不过在处理目前的简单问题时显然不需要任何的算法对应的思维。

遍历

遍历其本质上是一个操作，指的是从某一处依次访问直到另一处停止，一般我们会使用for来进行这个操作。通常有很多情况我们具有的是一组数据（比如说实例中的情况，或者说一长串的字符串也可以说是一组数据），而我们想要知道这组数据中的某一个属性（我们暂且抽象的称之为属性，它可以是全部值的和，一个最小值，或者某一个值所在的位置），这个时候要求我们对于整体的数据需要拥有一个认知，这时候就需要我们依次访问数据之中的每一个值，这时候使用for总是没错的。

count与flag

有的时候我们需要知道某一个程序块之中发生了什么，但是外界的下一个程序块无法直接知道，所以我们需要创建一些具有代表性的变量，我愿意称之为count与flag。

count是一个计数器，比方说当你需要计算1到100的数字中有几个是质数的时候，每次检测到质数后让count++无疑会是一个好想法。

而flag是一个指示灯，而当我们需要知道一串字符中是否存在字母A的时候，我们在检测到A之后让flag = 1然后break，自然可以解决这个问题，而也正是因此，对于**flag与count的初始化问题需要格外重视。**

不要使用库函数

在处理问题的时候我们有的时候会依赖一些库函数去解决问题，但是一般我们不推荐你这样做，作为新手，使用一个看似便捷但是事实上你都不清楚内部的运行原理的库函数的时候，报错之后往往会让你无从下手，有的时候手动实现这个方法虽然看上去臃肿且貌似很笨，但是事实上程序的运行时间也不会慢上很多，并且方便后续的Debug（纠错）。

当然，一些重要且常用的还是要记住的，这里一般是数学函数，尤其是sqrt()。

不要记忆Ascii码

虽然说使用Ascii码能显得你是一位高手，但是往往会降低程序的可读性（无论是谁看到一串数字也要过一会才能反应过来其对应的字母），得益于编译器，我们可以字符和其对应的Ascii码可以自由的转换，所以当我们下一次需要把一串内容是数字的字符转成真正的数字的时候，不要再使用`str[1] - 48`了，使用`str[1] - '0'`会方便很多。

即时输出

有的时候我们需要处理的问题要求我们找到某事物并输出，按照我们理论来说上述的方法，我们需要创建一个用于记录全部的找到的事物的“容器”并且在程序的最后输出，但是往往容器创建多大、如何知道要输出多少，这些问题对于新手来说反而更容易出现问题，所以此时可能需要稍微打破上述的规矩，当我们遇到这种情况时，立刻输出我们找到的事物。

背诵一些必要的模板

对于高手来说，一些操作他们可以下意识一步步做出来，但是对于新手来说还是需要一些背诵，记下来那些常用的功能如何实现：

求一个数字是否是质数

以下是一种较为省时但是方便理解的方法，诸如埃式筛法等更加美妙的算法请自行到搜索引擎搜索，假如学有余力。

```
#include <cmath>
bool isPrime(int n)
{
    if(n == 0 || n == 1)
        return false;
    for(int i = 2; i <= sqrt(n); i++)
    {
        if(n % i == 0)
            return false;
    }
    return true;
}
```

从字符转化为数字

```
//本内容直接写在main函数中
char a[10];//假设已经知道了这个十位的数组，其中已经有了确定的内容
int num = 0;
for(int i = 0; i < 10; i++)
{
    int temp = a[i] - '0';
    for(int j = 0; j < 10 - i - 1; j++)
        temp *= 10;
    num += temp;
}
```

排除故障

前言

本人虽然不敢说擅长编程，但是帮助不少的同学检查过他们错误的程序，并且成功找到错误，我将在下面列举这些常见错误，理论来说只要你按照这个顺序依次检查，能够检查出来99%新手常见的问题。

临时值归零

假如说不按照我上述的方法进行创建临时值，我们通常会遇到一种情况，比方说在上述的“实战”问题中，我们要找到总分大于200分的学生，并且输出他们的学号，按照上述的“进阶”中“及时输出”的原则，我希望你能理解下面这个代码片段。

```
int sum = 0;
for(int i = 0; i < 10; i++)
{
    //计算总分
    sum += stu[i].score[0] + stu[i].score[1] + stu[i].score[2];
    //如果大于200分就输出
    if(sum > 200)
        cout << stu[i].num << endl;
}
```

有了之前的提醒，应该很容易发现我们应该直接写`sum = stu[i].score[0] + stu[i].score[1] + stu[i].score[2];`或者在这句之前加上`sum = 0;`进行归零处理，在处理count类型的临时变量的时候我们一般都会遇到这种情况，只不过在一些时候我们确实没有使用赋值符号的地方，而后续的操作之中我们又忘记了在开始的时候加上归零，这时候一匹脱缰的野马就诞生了，这个计数器会无限的上升，冲破一切的束缚，并且让程序偏离原来的走向。

临界条件

有的时候我们需要考虑临界条件的存在，也就是在for这种问题上，我们的第二个式子之中应该是<还是<=或者是其他的东西，再加上一些奇怪的内容，这方面想不清楚往往也会是报错的主要原因之一，在这里我们举两个经典的例子：

1.边界去留

```
#include <cmath>
bool isPrime(int n)
{
    if(n == 0 || n == 1)
        return false;
    for(int i = 2; i < sqrt(n); i++)
    {
        if(n % i == 0)
            return false;
    }
    return true;
}
```

这是一个很像上面提到的判断质数问题的函数，但是它报错了，当我们输入9的时候，它给出的结果居然是true，甚至输入4的时候我们也会得到true的结果，这就是对于边界条件的去与留思考不仔细的结果，代入程序的逻辑看一下，我们就会发现假如说对于n，n的平方根是一个质数，那么这个值无疑是需要被考虑的，比如上面提到的4、9，所以毫无疑问`i < sqrt(n)`这里的小于号应该改为小于等于。

2.边界对齐

```
char a[10];//假设已经知道了这个十位的数组，其中已经有了确定的内容
int num = 0;
for(int i = 0; i < 10; i++)
{
    int temp = a[i] - '0';
    for(int j = 0; j < 10 - i - 1; j++)
        temp *= 10;
    num += temp;
}
```

依然引用了上面的程序，我们都知道这个程序最为惊险的地方就在于`j < 10 - i - 1`的选择，为什么不是10 - i，或者是10 - i -2，我们在这里提供一种思路，就是对于临界情况的考虑。

对于整个数组的最后一个数字，i = 9，那么这个时候我不希望它进行任何一次的乘以10的操作，于是for应该是恰好不执行，也就是说此时恰好是j < 0的情况，也就是10 - i - bias(一个未知的偏移量)此时等于0，那么不难求出bias = 1，当然，假如你不知道应该减i，可以设成10 + ki + bias，然后寻找两个i的情况，建立一个方程组进行求解。

如此，这个问题也就有了答案，而这一切基于的思想是“for的操作是连续的”，假如有精力可以理解一下这句话的含义。

堆栈溢出_段错误

有的时候我们会面临这种问题，这是因为PTA编译器本身的一些原因，我们只需要把那些看上去很大的嵌套了很多层的变量声明放到全局（在main函数的外面）一般就可以解决这个问题。

小问题

同时编程中我们还会遇到一些小问题，其中比较经典的就是`cin`与`cin.getline`的冲突问题：

```
int a = 0, b = 0;
string c;
cin>>a>>b;
getline(cin,c);
```

此时会出现错误，当你按照要求输入了两个数字，然后按下回车，之后输入一个字符串，或者按下空格再输入字符串，当你输出这串字符串的时候你会得到一个前面多出了一个你输入的空格的字符串，或者一个空的字符串，这是因为`getline`的读取特性，此时你只需要在`cin`和`getline`之间添加一句`cin.ignore();`便可以解决问题。

输出格式

字面意思，输出格式的问题，题目会对于输出格式有各种奇怪的要求，在输出之前检查空格、大小写以及回车，不要在这里出现错误。

结语

也就是说，当我们发现一个报错，而且我们面临的不是全错，我们可以确定我们的想法是没有错的，那么我们注意每一个被创建的临时变量的处理，在for和if中判断的边界条件是否合理，并且注意一下题目的输出要求，就可以解决大多数的问题，是不是很简单？但是一定要仔细，这是我帮助别人检查程序而不是自己疯狂报错的不二法门。

附录

输入

没什么好说的。

输出

使用`cout`的时候直接使用即可，但是有的时候我们可能需要输出保留一定位数的小数，或者需要在整数的前面填充一定的0，这时候使用`printf`对于我来说是一种更加简单的方法。

printf()

`printf`在基本使用时的组成格式大概是这样子的：

```
printf("");
```

这一部分是一个基础的框架，假如说你有任何的想要输出的文本之类的东西，就都放在""里面。而同时假如说你需要输出一些变量，你就需要使用**说明符**了。

说明符的使用格式为%+其本身，而说明符不同则对应着你选择输出的变量的数据类型不同，在这里给出几个常见的说明符：

说明符	对应数据类型	备注
d	整型	
c	字符型	
f	单浮点型	
lf	双浮点型	
s	字符串型	char*

这时候假如说你需要输出这个变量，在你希望它在的地方打了占位符后，在""的后面写,+变量名或者直接使用这个变量，就可以完成输出，同时为了进行多个变量的输出，可以连续使用多个说明符，之后在""后面连续使用,+变量名的输出格式。

给一个极端的案例来方便理解一下

```
#include <stdio.h>
using namespace std;
int main()
{
    char a='l';
    printf("H%cl%co W%cr%cd",'e',a,'o',a);
    return 0;
}
```

输出结果：Hello World

同时补充一个转义符，即，在`printf`的双引号中写下`\n`可以代表换行。

关于输出固定位数的小数

格式：%+说明符的完整格式是%+宽度+.精度+类型符

也就是说，假如说你要输出两位小数，你可以写`%0.2lf`或者`%.2lf`，此处假如说不输入宽度或者精度对应的格式，或者输入值为0，将输出实际位数。

宽度的意义在于这个数字的长度，假如说本身的数字宽度小于要求的宽度，那么会在它前面输出空格直到宽度为所需宽度，其中小数点也算是一位宽度度。若大于，则无事发生。

而精度的意义则在于，为单双浮点为保留到小数点后几位，为整型时为在数字前面添加0之所需位数位置.

如`printf("%4.3d",10)`的输出结果为" 010"，而`printf("%4.3lf",10)`的输出结果为"10.00".