※main 是一个函数名不是关键字，这个是可以改的，但是一般没人改，因为main作为程序的入口已经成为一种共识。

所以变量命名可以是main

※定义的变量或常量最好一定要用，有些编译器会有warning。

※Warning: value computed is not used.

Computed :计算（computer）

计算的结果没有使用,比如单写一行3+4;

**输入：**

※getchar()一定要关注会不会导致漏项

※scanf()模式匹配的原理

参考：<https://www.cnblogs.com/shuaishuaidefeizhu/p/5886899.html>

一、理论说明

1、用户**输入**的字符，会以ASCII码形式存储在键盘缓冲区；//如果字符接收到换行符，输出%c || %d 得到10（换行符的ASCII码） 见程序实例2

2、每调用一次**scanf函数**

①就从键盘缓冲区读走一个字符，相当于清除缓冲区；

①’调用scanf()函数时，用户最后输入的回车也会储存在键盘缓冲区；//见程序示例2

②若用户一次输入n个字符，则前n次调用scanf函数都不需要用户再次输入，直到把缓冲区的数据全部读取(清除)干净

二、代码示例

三、清除缓冲区的几种方法

1、为什么要清理？

我们使用多个scanf()的时候,如果输入缓冲区还有数据的话,那么scanf()就不会询问用户输入,而是直接就将输入缓冲区的内容拿出来用了,这就导致了前面的错误影响到后面的内容(比如scanf()输入完后会有换行符残留，会影响下一个scanf(()、getchar()、gets()的输入!————标志：多次(2+)输入、多次取缓冲区

为了隔离这种问题,需要通过各种方法将输入缓冲区的内容读出来（清除）

1、fflush(stdin)

//程序示例1错误！在试验区进行了验证与更改 见 scanf()原理程序实例2.c

1. 此种方法对vc可以，但对xcode和linux不适用

② 缓冲区的内容会全部清空！

2、while+getchar

while (ch=getchar() != '\n' && ch != 'EOF'),直到读取到缓冲区的换行或者空值

由于getchar()取到了有效的一个字符，所以要用ch保存，（while的出口是其条件的否命题）

3、利用模式匹配

scanf(“ %d”, &a);

在要输入的东西前多加一个空格以匹配空符！

//详细代码见 试验区有关缓冲区的实验.c

int a, c; char b;

scanf("%d %c", &a, &b); //若直接输入一个数字 和一个长度超过一的 数字 或 一个字符加一个数字

printf("1\n"); //会直接跳过第二个scanf()的步骤

scanf("%d",&c); //因为缓冲区里已存在符合模式的内容

printf("%d %c %d", a, b, c); // 这就是模式匹配的原理

缓冲区在实践区的实验.c

标志：多次(2+)输入

利用模式匹配的方法解决缓冲区的问题

float x1, x2, y1, y2;

float d;

x1=x2=y1=y2=d=0.00;

printf("Please input the coordinators of two points: \n");

printf("point1(x1, y1): ");

scanf("(%f, %f)",&x1, &y1);

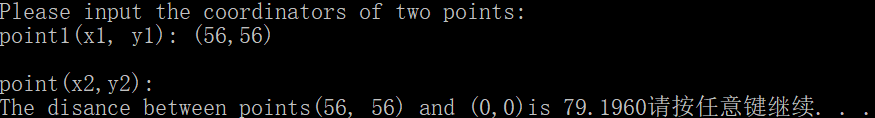
printf("\npoint(x2,y2):");

scanf(" (%f, %f)", &x2, &y2);

d= sqrtf((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));

printf("\nThe disance between points(%g, %g) and (%g,%g)is %.4f", x1, y1, x2, y2, d);

//如果第二个scanf的格式串里没有前面的若干（至少一个）个空格，会出现下面的状况



p.s. 如果在终端输入的时候用中文输入法也会出现此问题

※对于格式要求多变（好多未知的空格）的编程题，善用scanf()的模式匹配解决问题

当然仅仅如此是不够的，还要利用好scanf（）的返回值才能更好地保证输入

scanf(" %c%lf", &ch, &f) == 2

就能很好的解决问题

printf("Enter an expression:\n");

scanf("%lf", &sum);

while(scanf(" %c%lf", &ch, &f) == 2)

{

if( (ch != '+')&&(ch != '-')&&(ch != '\*')&&(ch != '/'))

break;

switch(ch)

{

case'+':

sum = sum + f;

break;

case'-':

sum = sum - f;

break;

case'\*':

sum = sum \* f;

break;

case'/':

sum = sum / f;

break;

}

}

printf("Value of expression: %g\n",sum);

在逻辑用语里注意 || 与 && 也涉及到否命题中

当初写此代码时 写成 || 一直没有发现

※

出现的标志：

1. 程序运行卡壳
2. 返回值



可能的原因：

1、当格式串中对应的转换说明没有对应 e.g. %d %c %s对应错误也会返回这样的值

2、Scanf()表达式中缺少地址符：

※warning: the `gets' function is dangerous and should not be used.

该函数在读取字符串时不检查边界，可能会造成内存访问越界

解决方法：用fgets替换

#include<string.h>

char str1[40];

e.g. fgets (str1,40, stdin);//stdin表示从键盘输入

·补充：函数原型

char \*fgets(char \*buf, int bufsize, FILE \*stream);

参数

\*buf: 字符型指针，指向用来存储所得数据的地址。

bufsize: 整型数据，指明存储数据的大小。

\*stream: 文件结构体指针，将要读取的文件流。

同理有scanf() 与 scanf\_s() 的区别

scanf\_s() 的功能虽然与scanf() 相同，但却比 scanf() 安全

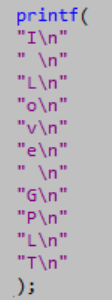
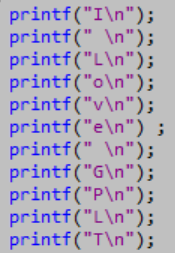
必须提供一个数字以表明最多读取多少位字符。

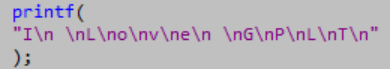
e.g. scanf\_s("%s",buf,5);

p.s. ANSI C中没有scanf\_s()

**输出：**

※printf()

1. 一定要关注 \n
2. 不要习惯性地使用%d，有时候会导致编译通过，但是结果错误。
3. 简化，一次函数调用即输出多行:

等价于  等价于

但是不要分行就忘了“;”，结构的定义也是同理

第二种形式用一个函数调用即输出多行，但不利于阅读

第三种形式依然是一个语句（；），是格式串的集合，但更利于阅读，是二的加强版

※当输出结果一直是5.0 6.0这样的浮点数的时候一定要想是不是整数与整数之间的除法赋给了浮点数

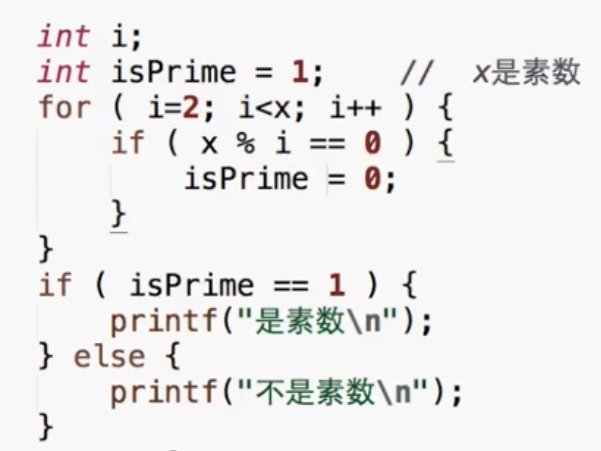
**程序设计：**

※初始值的设定

1. 累加的sum初始值为0；累乘product的初始值为1。
2. Flag 在证实的时候设定为0,；在证伪的时候设定为1.

(0一般默认表示错误，1表示正确，有利于可读性)

下面是在判断是否为整数的证伪代码：



1. 于累加累乘的值一定要初始化！！

int x, n;

int sum=1;

int i=1;

scanf("%d %d",&x, &n);

if(n==0)

sum = 1; //一个函数的定义 1、阶乘、指数、斐波那契 首几项要弄好

for (i=1; i<=n;i++)

{

sum \*= x;

}

printf("%d",sum);

没有初始化会直接导致基本所有测试数据错误

※在涉及表达式计算的时候关注变量隐形转化的问题

当输出结果一直是5.0 6.0这样的浮点数的时候一定要想是不是整数与整数之间的除法赋给了浮点数

结构：

※if 改成while

if while 的本质区别只在于执行的次数！

所以使用时根据所需执行的次数选择

※if 一定要加花括号！！！switch一定记得加花括号

不要偷懒，最省力的方法就是不偷懒

写多个语句的时候一定要考虑有没有复合语句的关系（除了case）

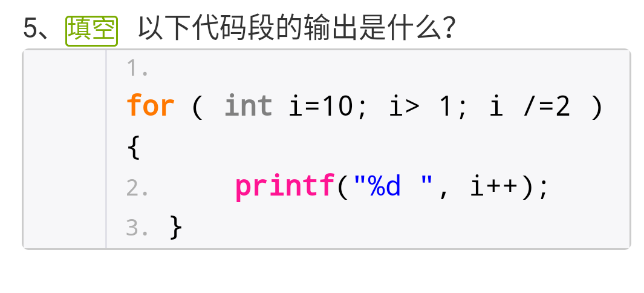
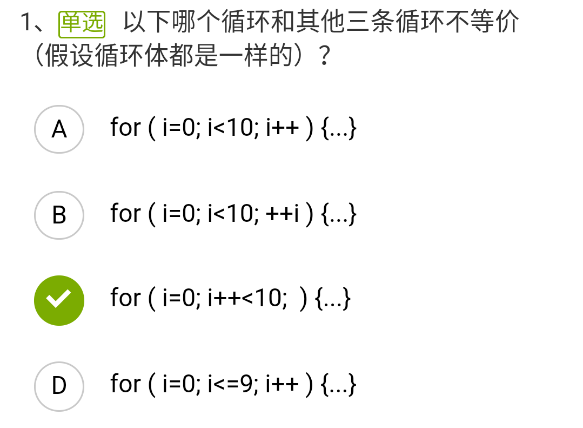
在圆括号后尽量加花括号！！

※编程时 条件为多分支if结构

如果分支多，建议使用级联的结构，省力！

※对for语句本身的误解

(5的代码实现见试验区的 死循环？.c ; 图片也来自试验区的 for1 for2)



For语句中()中的三个语句不是分别按顺序执行完后再执行循环体的，而是先执行一，判断二，然后进行循环，最后在循环末执行三。这就是For语句的“奇怪”之处(没有while的顺畅)。

※草稿技术

// i a d sum a\*\*\*a在最后面再打一次草稿是有必要的

// 1 123 3 3 12

// 2 12 2 32 1

// 3 1 1 321 0

// ??循环变量是否会大一

※数据全面性检验的方法：

//对全部数据的覆盖度（数据类型、数据类型下对应的所有范围）

// 1）超出输入要求的数据 进行合法性检验（可使用一些库函数e.g. isnumeric），排除

// 2）对范围内的数据的通性，即对所有数据都符合题目输入输出要求

// 1>输入数据从最小到最大轮次思考

// 2>反过来利用输出数据进行检验

例子：

//题目内容：

//

//UTC是世界协调时，BJT是北京时间，UTC时间相当于BJT减去8。现在，你的程序要读入一个整数，表示BJT的时和分。整数的个位和十位表示分，百位和千位表示小时。如果小时小于10，则没有千位部分；如果小时是0，则没有百位部分；如果小时不是0而分小于10分，需要保留十位上的0；如果小时是0而分小于10分的，则不需要保留十位上的0。如1124表示11点24分，而905表示9点5分，36表示0点36分，7表示0点7分。

//

//有效的输入范围是0到2359，即你的程序不可能从测试服务器读到0到2359以外的输入数据。

//

//你的程序要输出这个时间对应的UTC时间，输出的格式和输入的相同，即输出一个整数，表示UTC的时和分。整数的个位和十位表示分，百位和千位表示小时。如果小时小于10，则没有千位部分；如果小时是0，则没有百位部分；如果小时不是0而分小于10分，需要保留十位上的0；如果小时是0而分小于10分的，则不需要保留十位上的0。

//

//提醒：要小心跨日的换算。

#include <stdio.h>

int main ()

{

int min1, hour1, min2, hour2;

int in;

int sum1, sum2;

scanf("%d",&in);

if(in > 2359 || in < 0)//数据合法性检验

{

return 0;

}

min1 = in % 100;

hour1 = in / 100;

sum1 = hour1 \*60 + min1;

if (sum1- 8\*60 > 0)

{

sum2 = sum1- 8\*60;

}

else

{

sum2 = 24\*60 + sum1- 8\*60; //跨日的计算

}

min2 = sum2 % 60;

hour2 = sum2 / 60;

if (hour2 != 0) //抓住问题产生分类的关键

{

printf("%d",hour2);

printf("%02d",min2);

}

else

{

printf("%d",min2);

}

return 0;

}

// 在此问题中， 1）有效的输入范围是0到2359

// 2） 1>1、12、102、1102

// 2>涉及到 跨日的换算