

# C2-E2答疑要点整理

## C2-E2答疑要点整理

- 格式符和数据类型要匹配
- 学会使用字符面值
- 循环中变量的处理与重置
- 运算过程中的数据类型问题
- 数组的定义使用与全局数组
- 循环结构的使用错误

## 格式符和数据类型要匹配

使用 `scanf` 读入或使用 `printf` 输出的时候一定要注意**变量类型与格式符匹配**：`int` 类型对应 `%d`，`long long int` 类型对应 `%lld`，`double` 类型对应 `%lf`（注意 `double` 类型的输出需要使用 `%f`），`char` 类型对应 `%c`，`unsigned int` 对应 `%u`，`unsigned long long` 对应 `%llu`

如果不匹配，在C99标准中属于“未定义行为”，编译器可以任意实现未定义的行为（是的，按理来说，如果出现未定义行为，编译器给你弹出一个俄罗斯方块游戏，那也是合理的！）也就是说很有可能在你的电脑上，它按照你的想法运行，但是换一个环境，它就不按照你的想法运行了。

我以前见到很多人这样写代码：

```
1 //例题：输入一个字符，输出它的ascii码值
2 char c;
3 scanf("%c",&c);
4 printf("%d",c);
```

这就是一种典型的“输出变量类型和格式符不匹配”，所以严格来说，这样的写法是不好的，应该写成：

```
1 char c;
2 scanf("%c",&c);
3 printf("%d",(int)c);
```

## 学会使用字符面值

在编写代码时，如果需要使用字符的 `ASCII` 值，不必查表记忆，可以**直接使用字符面值**，字符面值为由单引号括起来的单个字符。这样做一来可以减少大家记忆的工作量，二来可以使得你的程序更易读。

例如：

```
1 if ('0'<=c && c<='9')//判断 c 是不是数字字符
2 if ('a'<=c && c<='z')//判断 c 是不是小写字母
3 if ('A'<=c && c<='Z')//判断 c 是不是大写字母
```

再次强调：**由单引号括起来的单个字符才是字符面值**，没有单引号括起来的字符是**变量名**，若该变量在此前未被声明，则编译时会报错（变量未定义），若该变量在此前已被声明和使用（但存入的值不是它对应的字符面值），则编译时可能不会报错，但代码运行结果可能与期望的相差很远

```
1 //char类型变量c与字符字面量'A'进行比较
2 char c;
3 scanf("%c", &c);
4 if (c == 'A'){
5     ...
6 }
7
8 //char类型变量c与变量A进行比较
9 //若变量A未定义则会编译报错
10 //若已定义使用则编译可能不会报错，但结果可能非预期
11 char c;
12 scanf("%c", &c);
13 if (c == A){
14     ...
15 }
```

## 循环中变量的处理与重置

在进入循环前定义的变量，如果在每趟循环开始或结束时不进行重置值操作，那么在进入下一趟循环后会保持原有值不变，从而导致存在多组数据的题目中产生输出出错的问题。**此问题尤见于多组输入输出问题中。**

下面是一道例题，以及一种典型的**错误**写法：

```
1 /*
2 例题：给定n,m，分别计算n组输入中每组的m个数之和
3 */
4
5 //一种比较典型的错误写法如下
6 int n, m, sum = 0, tmp;
7 scanf("%d%d", &n, &m);
8 for (int i = 0; i < n; ++i){
9     for (int j = 0; j < m; ++j){
10         scanf("%d", &tmp);
11         sum = sum + tmp;
12     }
13     printf("%d\n", sum);
14 }
```

错误的原因：`sum` 变量虽然在声明时被初始化为 0，但在内层循环结束、输出 `sum` 的值后，`sum` **并没有被重置为 0**，而是在进入下一次外层循环时保持原值，这就导致当外层循环跑后面的  $n - 1$  趟时，`sum` 变量的初始值不是 0，从而每次累加 `tmp` 值都会出错

要避免这样的错误，就需要在每次跑  $m$  趟循环累加之前，先将 `sum` 变量置为 0，或者在输出 `sum` 变量后，立刻将 `sum` 的值置 0，如下：

```
1 /*
2 例题：给定n,m，分别计算n组输入中每组的m个数之和
```

```

3  */
4
5  //正确写法之一
6  int n, m, sum = 0, tmp;
7  scanf("%d%d", &n, &m);
8  for (int i = 0; i < n; ++i){
9      sum = 0;    //可以在内层循环开始前，置sum为0
10     for (int j = 0; j < m; ++j){
11         scanf("%d", &tmp);
12         sum = sum + tmp;
13     }
14     printf("%d\n", sum);
15     //也可以在输出sum后，在这里将sum置为0
16 }

```

另外，由于 C99 不要求所有的变量定义声明都写在 `main` 函数中的最上面，也可以在进入内层循环之前定义计数变量 `sum`。这样，新定义的变量作用域就是本次外层循环，当外层循环的一趟结束时，在循环内定义的变量 `sum` 会被清理掉，在循环外无法使用该变量，进入下一趟外层循环时，又会重新定义声明一个新的 `sum` 变量：

```

1  /*
2  例题：给定n,m，分别计算n组输入中每组的m个数之和
3  */
4
5  //正确写法之二
6  int n, m;
7  scanf("%d%d", &n, &m);
8  for (int i = 0; i < n; ++i){
9      int sum = 0;    //循环内定义并初始化sum变量
10     for (int j = 0; j < m; ++j){
11         int tmp;
12         scanf("%d", &tmp);
13         sum = sum + tmp;
14     }
15     printf("%d\n", sum);
16 }

```

## 运算过程中的数据类型问题

`int` 类型变量所能存入的整数是有大小范围的。当两个 `int` 类型变量执行运算时，如果结果超出了 `int` 变量所能存入的范围，就会**立即发生溢出**，即使将结果赋值给 `long long int` 变量也是将错误的值存入了变量。因此，当题给数据范围存在溢出风险时，要么**直接使用** `long long int` 类型变量存储和运算，要么在运算前将其中一个运算数使用**强制类型转换**先转为 `long long int`，这样才能得到正确的结果。

举个例子，请大家猜一猜下面这个代码会输出什么：

```

1 #include <stdio.h>
2 int main(){
3     long long int a=2000000000*2;
4     printf("%lld\n",a);
5     return 0;
6 }

```

它会输出正确答案 4000000000 吗？错了！它的运行结果是：

```

1 #include <stdio.h>
2 int main(){
3     long long int a=2000000000*2;
4     printf("%lld\n",a);
5     return 0;
6 }

```

Output: -294967296

可以看到结果和我们预料到的不一样，因为出现了有符号整型数溢出的未定义行为。如果你看了上一期答疑要点，在编译选项里加入了 `-Wall`，那么这段代码会有一个警告：

```

1 [警告] integer overflow in expression [-Woverflow]

```

为了得到正确的结果，我们可以进行类型转换，在两个乘数后面加个 `LL`，来表示它是 `long long int` 型：


```

1 #include <stdio.h>
2 int main(){
3     long long int a=2000000000LL*2LL;
4     printf("%lld\n",a);
5     return 0;
6 }

```

这样就对了：

```
1 #include <stdio.h>
2 int main(){
3     long long int a=2000000000LL*2LL;
4     printf("%lld\n",a);
5     return 0;
6 }
```



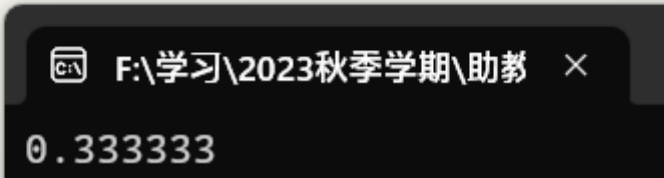
同样的道理，当除号的两个运算数都是整型时，本次执行的除法是整除，即使将结果赋给 `double` 类型变量也只是存入了错误的值。例如：

```
1 #include <stdio.h>
2 int main(){
3     double a=1/3;
4     printf("%f",a);
5     return 0;
6 }
```



如果想要让除法成为浮点数除法，需要在执行除法前，先使用强制类型转换将运算数转为 `double` 类型。

```
1 #include <stdio.h>
2 int main(){
3     double a=1.0/3.0;
4     printf("%f",a);
5     return 0;
6 }
```



我们的建议是：如果一个表达式，你希望它最终的结果是浮点数，那么其中的每个量都要是浮点数。变量，使用强制类型转换 (`double`) 或使用乘以 `1.0` 的方式转为浮点数；常量，不要写作 `2` 而是写作 `2.0` 来声明其为浮点数。

## 数组的定义使用与全局数组

数组的定义方式为：

```
1  数组中元素的类型  数组名[数组大小];
```

注意**数组的下标范围**是从 0 ~ 数组大小-1，而不是 1 ~ 数组大小，使用时一定要注意不要访问超出数组下标范围的位置

全局数组定义的位置在 `main` 函数的前面，和在 `main` 函数内定义的数组相比，一般可以有更大的容量。以经验来看，当所使用的 `int` 类型数组大小超过 100000 时，就需要改为在 `main` 函数前面定义全局数组，而不是在 `main` 函数里定义数组。`main` 函数中过大的数组会导致栈溢出错误，进而直接使程序终止。

## 循环结构的使用错误

在 `for` 循环和 `while` 循环的圆括号后面，**不能**输入分号，否则会使这个循环在遇到分号后就会结束，无法“管住”后面花括号包裹的代码块，进而导致循环体在循环结束后才去执行（且只执行一次），或因为循环条件无法被打破而无尽循环、卡住代码执行的问题。

错误：

```
1  for(int i=1;i<=n;++i);
2  {
3      //需要循环执行的代码
4  }
```

正确：

```
1  for(int i=1;i<=n;++i)
2  {
3      //需要循环执行的代码
4  }
```

---

作者：梁秋月，逐月的流星

审核：王君臣