关于C考试，xdx该注意些什么

MC的大虾 申雄全

xdx们要期中考辣！那么（对于大多人）初学的C语言有什么需要注意的咧？

基础篇

1.对于数组的处理

```c
//有些同学可能会这样开一个数值
#include<stdio.h>

int main(){
    int m;
    scanf("%d",&m);
    int a[m];
    return 0;
}
```

- 这样开一个数组后有些同学在本地的编译器可以通过，但是提交时评测机**有一定可能性**给出CE的错误，这种错误一般是代表编译不通过。
- 这是因为这样定义数组是不安全的，因为我们在 `int a[]` 开辟一个数组时，系统需要在编译的时候给数组分配相应的内存空间；而m实际上是一个变量，这在编译的时候就有可能出现错误。
- 所以在使用数组的时候，我们一定要根据题目意思开一个**大小合适**的定长数组。
- 怎么开？例如我需要对100个数进行升序排序
  我们就可以开一个大小为a[105]的数组，比给定的最大数100还要大一些，防止出现数组越界导致REG的问题。 **（本质上是要做好边界处理）**

2.对于标准输入和输出函数

这个可以详见菜鸟教程，相信大家学了 `gets` 和 `puts` 后还会来看这里的

我简单提一句吧，

```c
#pragma region

//对于一些常用的类型
scanf("%d",&a);
//一定要记得加上&符号

printf("%d",a);
//不要加&号
//等学到指针就会明白为什么了
#pragma endregion
```

3.对于自增和自减

自增和自减的规则是一样的，这里我以一段代码举例说明吧。

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

int main(){
    int a=0,b=1;
    printf("%d %d\n",a++,a+b);
    printf("%d %d\n",a-b,++a);
    printf("%d %d\n",b++,++b);
    a+=b;
    printf("%d",a);
    return 0;
}
//请问程序输出是多少呢？
//答案如下
//0,2
//0,2
//1,3
//5

#pragma endregion
```

请记住以下规则：

如果a初始为m，

- 执行a++后，会返回a自增之前的值，即m，而a自增已经变成了m+1。（如果b=a++，a返回值m然后赋给b，b为m，而a此时变为了m+1）
- 对于++a,会返回a自增之后的值，即m+1，同时a因为自增也变成了m+1。

3.对于位运算的理解

位运算就是对位进行操作，1个位就只能存**0**和**1**。

同时在计算机里， `1` 代表 `true` ， `0` 代表 `false` ,这也是判断语句 `if(){}` 中 `()` 内执行的逻辑。

对于c语言，只要小括号内的值不为0，就执行 `{}` 中的语句，反之不执行。

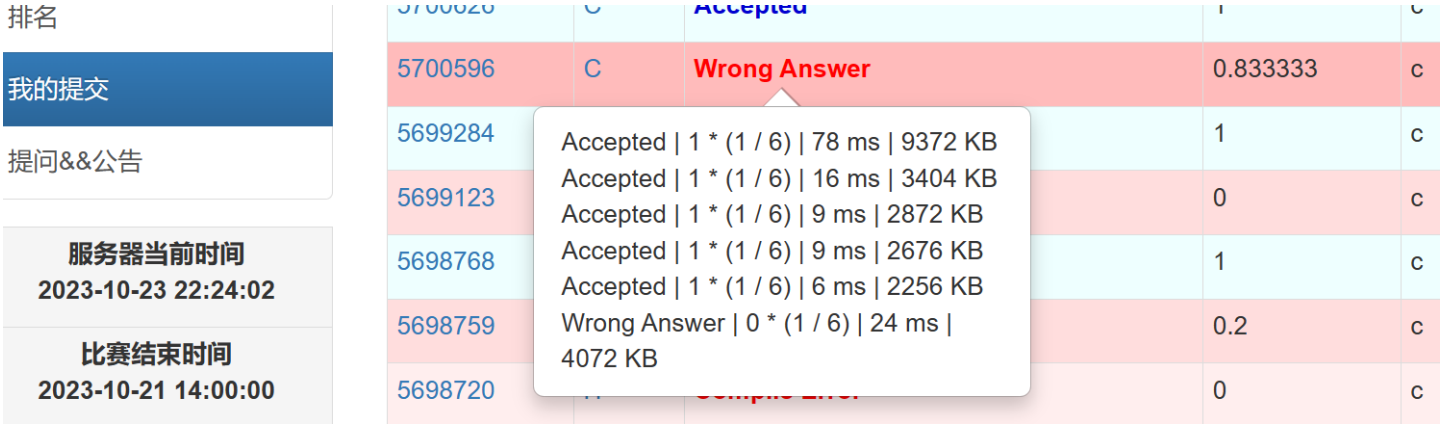
我们常见的 `int` 类型是4个字节，故32位，考虑到第一位是符号位，所以后面31位全部取1，符号位取0.就是int所能存的最大数据，把2进制数化成10进制，即$\sum_{i=0}^{31} 2^i$，得到大概是21亿多。

那么所谓的 << 和 >> 实际上是在把一个数乘以2，除以2。
从10进制来理解，我们把110右移一位，得到11，不就是除以10了吗，那对于二进制数0110(6)，右移一位变成0011(3)，这本质上是除以2。

其余的一些参照书上的规则进行即可。

4.不开long long一场空

我们在某些题目里面，明明觉得自己的代码十分完美，却依旧得到了wa的结果。
这时我们不放先看看数据点的通过情况。



只有最后一个数据点没过，有一种可能就是爆int了！

比如求斐波拉契数列的 `A[100]` ,结果是$2189229958345551690 26$,相信这已经远大于2.1×10^{10}了。

这种异常溢出情况是能够计算的。大家学习了补码的计算，用补码去计算就能很好找出规律。

所以时刻警惕数据范围，因为不开long long而丢分很可惜。

进阶篇

对于递归

给出下列一个题目吧，请输出$1-n$的全排序，按**字典序**输出，$1 \leq n \leq 10$

用排列组合简单计算一下，有A_n^n种情况，暴力循环是怎么样的呢?

```
#include<stdio.h>
int main(){
   int n;
   scanf("%d",&n);
   for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
        //且j不等于i;
    }
   }
    return 0;
}
```

我们要嵌套n个for循环，并且每一个内存循环的值不能与外层循环的相等，即1，1，1，1，1……是无效排序，1，2，3，4，5……或者2，1，3，4，5……才是合理的。

这样无脑嵌套for循环肯定是不合理的，因为n是不确定的。

我们就要思考一下用递归了。

我们通常是从小往大去思考一个问题，先把简单情况求出来，然后一个个合并。

但是递归往往是自顶而下的思考问题，从大问题入手，把大问题拆成**相同解法，规模不同**的子问题。

对于$n=10$，那么对于这道题目我们要做的是**排后面10个数**，就分解成先把第一个数确定好，再排**后面9个数**。

排10个数和9个数的规则都是一样的，按字典序排序，这两个问题的唯一区别就是问题规模大小不同。

那么我们不妨就用一个函数 `solve(i)` 来解决这个问题。函数的作用就代表排前i个数的字典序。我们调用 `solve(n)` 不就解决问题了吗。

```
#include<stdio.h>
int n;
int a[12];
int flag[12];//用来标记哪个数已经用了，如果最后一个数放了1，那么前面的数就不能排1了

void solve(int t){
    if(t==0){
        //递归底边，排到前0个数的时候，代表已经排完了
        for(int i=0;i<n;i++)
            printf("%d ",a[i]);//一个个输出即可
         printf("\n");
         return;//返回到上一层去。
    }
     for(int i=1;i<=n;i++){
          if(!flag[i])
          {
            a[n-t]=i;//第一个数排好了
            flag[i]=1;//标记已经使用了这个数
            solve(t-1);//排后面t-1个数
            flag[i]=0;//这一次排完了，重新开始下排。
            //所以flag[i]要置为0，可以重新用了
        }

     }
}//排后面t个数

int main(){

    scanf("%d",&n);
    solve(n);
    return 0;
}
```

有余力的可以看看分治法，递归就是其中的一种思想。

To be continue

规范篇

- 除非你是C语言大佬，手敲汇编扒王者源代码，不要命名一堆abcd变量!
  对于很多初学程序的来说，往往喜欢用 `abcd` 这样的名称来命名，这并不是不可理解的（因为毕竟题目描述里面大多也是这样的名字），不过尽量都要用**意义明确的，或者和题目紧密相关的，或者缩写较为普遍认可**的变量名。
  for example:

```
int main(){
    char s[] = "Hello, world!";
    int l = strlen(s);
    char c = 'e';
    int r = 0;
    for(int i = 0; i < l; i++){
        if(s[i] == c){
            r++;
        }
    }
    printf("%d", r);
}
```

这其实是一个非常简单的示例，统计了 `s` 字符串当中出现了多少次 `c` 。如果改成这样：

```
int main(){
    char string[] = "Hello, world!";
    int stringLength = strlen(s);
    char findingChar = 'e';
    int appearCount = 0;
    for(int i = 0; i < stringLength; i++){
        if(string[i] == findingChar){
            appearCount++;
        }
    }
    printf("%d", appearCount);
}
```

能更好地在编写与调试当中理解各变量的意义，并且让你的代码看起来**更像自然语言**。不说别的，拿这样的代码去给别人帮你debug也要比前者要好的多。

当然这段程序比较简单，可能看不出来胡乱命名可怕的程度。当你最终需要写比较复杂逻辑的程序的时候，就知道合理的命名有多么重要了。

- 避免过于复杂的 `if-else` 片段，避免过于深的缩进
  有的xdx的代码，看起来就有种拾级而上，蜀道难的美。

```
//...
if(con1){

}else if(con2){
    if(con3){
        if(con4){
            if(con5){
                if(con6){
                    //...
                }
            }
        }
    }
}
//...
```

像这样过于的缩进，使得代码总体变得混乱，可读性极差。
如果你真的需要这么多的条件同时判断，为什么不使用 `&&` ？

```
if(con1 && con2 && con3 /*&& ...*/){

}
```

- 如果你要用 `else` ，判断一下**真的需要if-else吗**?

```
while(true){
    if(breakCon){
        break;
    }else{
        //do something in loop
        //在这里，如果上面的if判断为真，那么break直接执行
        //就跳出循环了，根本不会执行else里面。
    }
}
```

不如修改成例外判断：

```
while(true){
    if(breakCon){
        break;
    }
    //do something in loop
}
```

- 如果必须要用一堆 `if-else` ，能不能使用 `switch` ？

```
int a = 1;
if(a == 1){
    //...
}else if(a == 2){
    //...
}else if(a == 3){
    //...
}
```

可以优化结构成

```
switch(a){
case 1:
    //...
    break;
case 2:
    //...
    break;
case 3:
    //...
    break;
}
```

同样的，这也是比较简单的示例，当你需要判断很多条件的时候，这样的方法能避免你把代码写成天梯。

- 合理的**封装函数**，让你的代码更易使用和调试。
  很多人可能认为，函数就是把一大块代码隐藏转移了嘛，还不如放在原位置让我看清运行细节。实际上，合理的函数封装，**能让你的代码更易于理解**，并且在找到bug的时候更好地专注于某个功能的错误。

例如这个函数，它实现了把一个字符串逆序

```
void Reverse(char source[], char buffer[]){
    strcpy(buffer, source);
    int len = strlen(source);
    for(int i = len - 1; i >= 0; i--){
        source[i] = buffer[len - 1 - i];
    }
    return;
}
```

如果我要经常用到它，那么就是一件很舒服的事情：

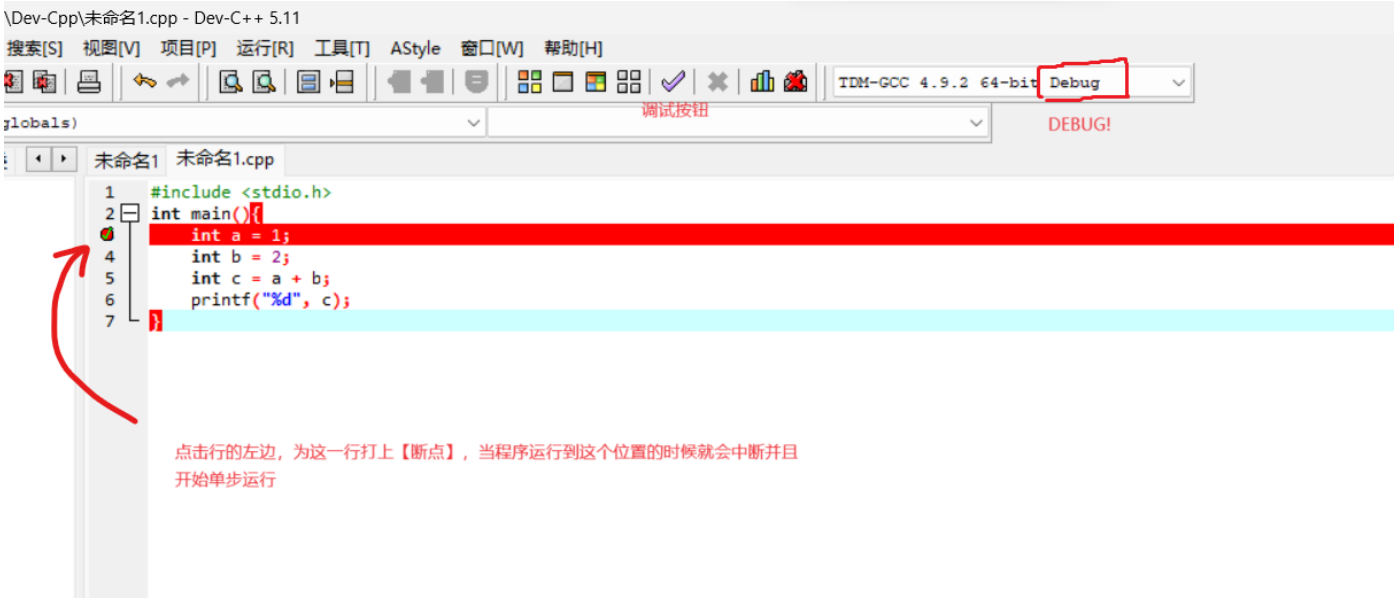
```
Reverse(string1, buffer);
Reverse(string2, buffer);
```

相比起来，你把源代码完全放在主函数里，会让函数更复杂，并且容易出现**一处改了另一处没改的错误**，如果我出现bug，调试 `Reverse` 函数如果实现有错，我就只需要关注 `Reverse` 而不需要去关注主函数复杂的逻辑。而且由于多了 `Reverse` 这个函数名字，也让自己和别人更好地理解你这一段代码到底是想做什么。所以同样的，函数的名字也要有意义，不要用 `f()` ， `func()` 这样模棱两可的名字去命名一堆函数啦！

技巧篇

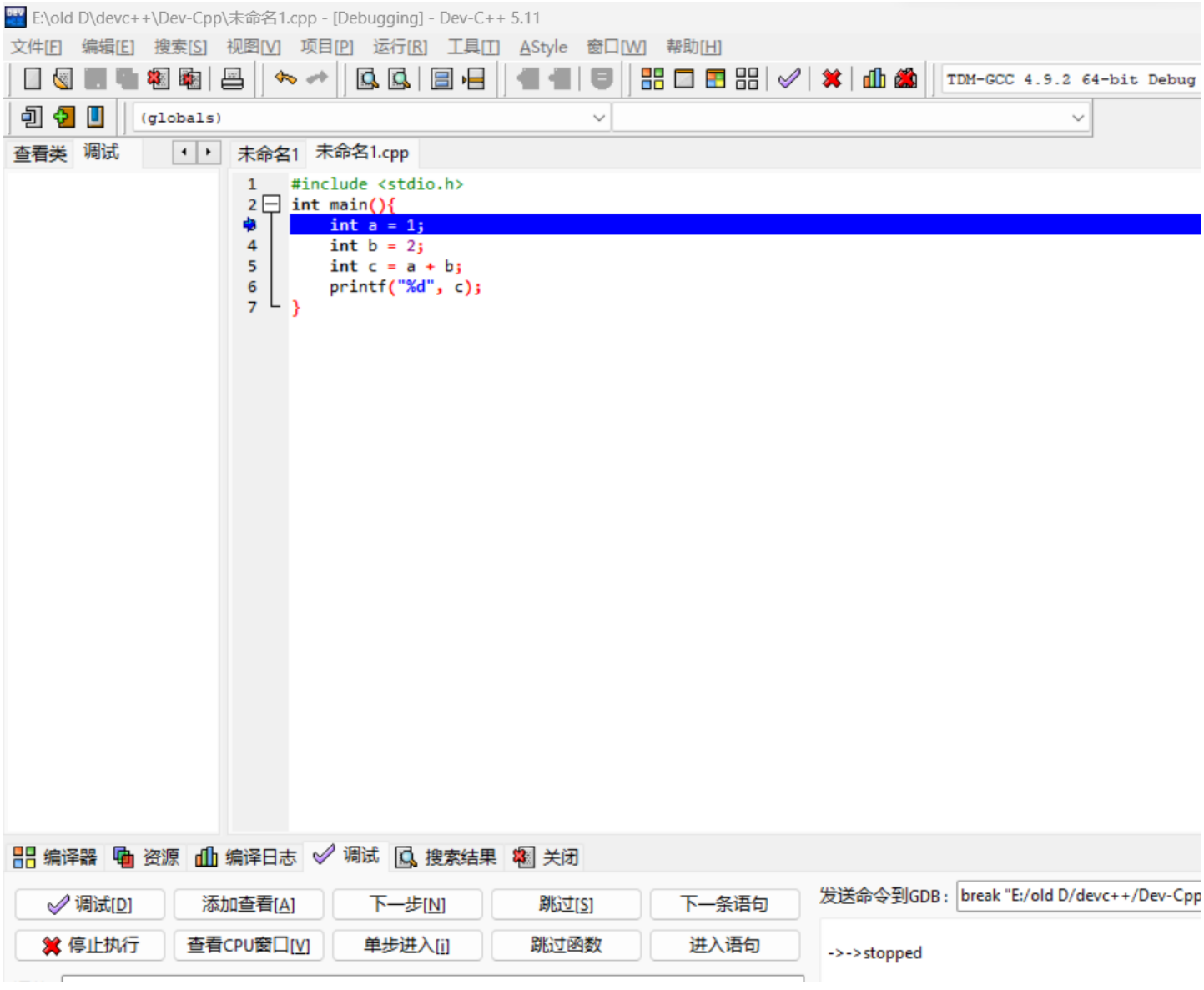
- **怎么调试**

  调试其实是写程序非常重要的技能，单步实现监视过程与变量，相信应该有很多同学用 `printf()` 打log这样的方式去debug自己的代码，其实很多代码你学会单步调试能节省很多时间，下面以大家最爱的 `DEV CPP` 举例。

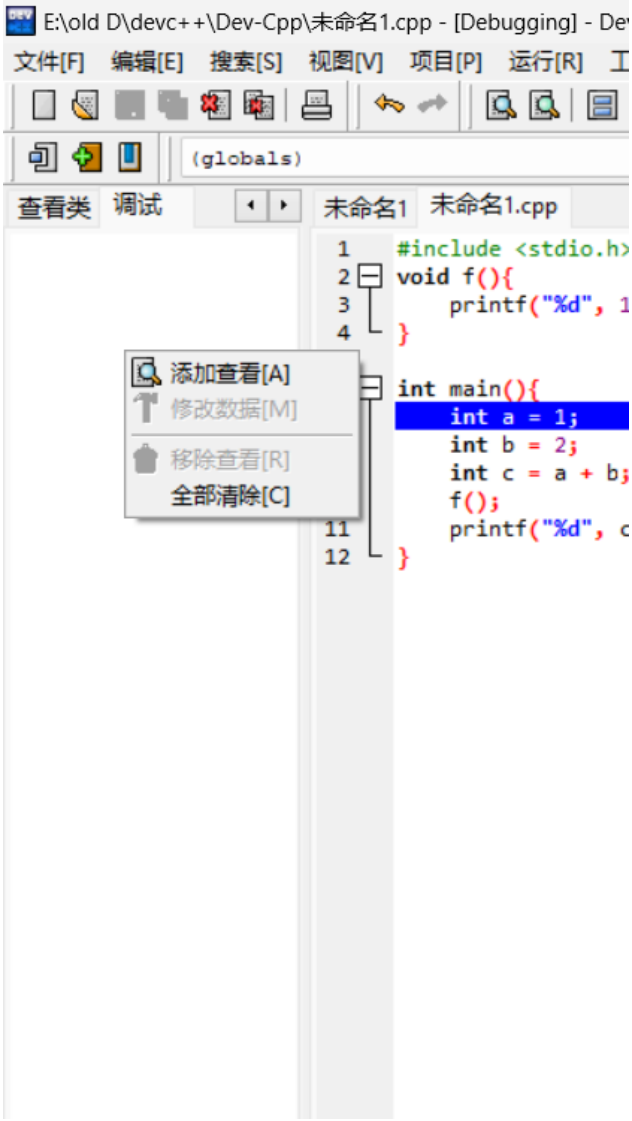


点击**调试**

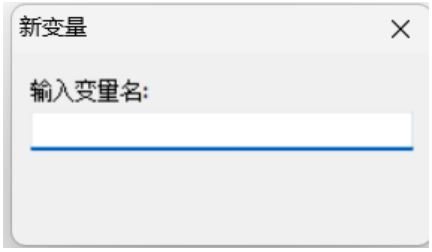
，你会发现程序运行到断点的位置就中止了，并且在代码编辑处出现了一条蓝行。



现在就看下面的位置，主要用到的就是**下一步**，**单步进入**，**停止调试。**最后一个的意义显然的，前两个的区别是**下一步会忽略具体函数内部，单步进入会直接进入函数内部（如果函数在此文件/项目）**。当然这样的操作还看不出来调试有多么牛逼，现在看编辑器的左边



右键调试框，点击**添加查看。**



输入变量名称或者表达式，就会直接在左侧列表看到对应的值，这比使用 `printf` 输出要快捷方便多了。**如果函数对变量有值的影响（例如 `scanf()` 等乱七八糟的函数），最好不要查看而是单步进入**

调试允许程序员实时修改某个值变量，但是要谨慎。

- 其他IDE,比如VSCode可能调试对于新手会相对困难，不过最傻瓜式的调试还是Visual Studio。*VS赛高!*