C语言入门 – 2023年12月11日

前言

断断续续写了四天的教程暂时也画上了句号。希望你可以在这个教程里学习到C语言的一二，由于是个人赶制，所以可能会有不少错误纰漏，还有不严谨和做得不好的地方，敬请原谅。

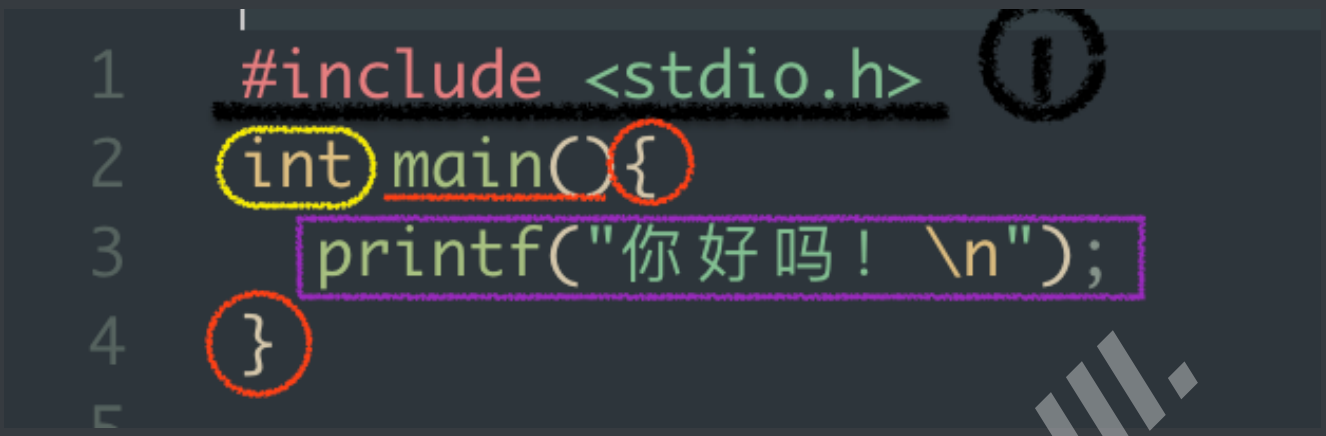
1.认识C语言

什么是C语言？它的前世今生是什么？

C语言前身其实是B语言（不严谨地说），因为作者为了致敬B语言开发者的牛逼，所以就引用了C语言这个单词（取用了 B语言 - BCPL 的第二个英文字母）（20世纪70年代，肯·汤姆森为了使其设计的Unix系统更加高效，使用B语言的变种（即C语言）在DEC PDP-7计算机上重写了Unix。C语言中许多重要概念来源于BCPL语言，其对C语言的影响也间接地来源于B语言。）

因为在计算机早期，计算机非常不普及，能用到的也绝非泛泛之辈，而后面就衍生了操作系统这种概念，之前的操作系统非常笨，而且使用很复杂，很无趣。直到那位男人，为了玩游戏，而移植了一个大型计算机的操作系统。而最后，一点点铸就了UNIX系统的诞生！既然系统有了，我总不能像个机器人一样天天说那机器人的话（汇编语言），人类说机器人的话，那效率也太低了，这不满足日益增长的计算速度和人类半天说不完一句机器话的需要。但是又不能让计算机屈尊跟我们讲人话，所以大佬就依照汇编语言（低级语言）发明了更简单，快速，牛逼哄哄的B语言（高级语言），用于高速地向那傲慢的计算机发布任务用（其实是为了方便人们针对该系统可以更快速、高速、高效地开发一些软件工具使用）

C语言的基础语法构成



第一行：#include是引入库的意思，比如在Python的时候，我们需要使用到print函数时，不需要额外引入其他库。但是当我们需要用到类似Re正则表达式、Time时间、Random随机生成数字之类的额外功能时（函数），就需要引入库。

那为什么C第一行就需要引入库呢？这段代码明明那么简单。是的，很简单，但是C这b，把东西都塞库里了，导致我们想使用Printf函数打印点东西都会报错：C说 - 你这Printf啥啊？我咋没见过啊，我不会啊！

就是如此生草，所以你还得引入一个称为标准库，或者说初始库的东西 —— stdio.h

当你使用命令告诉C，#include（我要引入了喔）<stdio.h>（看到<>里面的东西没，那就是我要引入的库，C语言，你要是看不懂我说什么，就在我引入的字典里找，你就会知道我说的printf是啥了。什么？！你还是不知道我说的是啥？！）

看了字典的我：好吧，是我的问题...我忘记这个函数在另外一个字典里了，这本小学生用的，只有一百多页厚的字典确实找不到这种生僻字...（也有可能是你写错字了，没有感情的机器人可不会帮你纠正这种低等错误！🤡）

第二行：在C语言中，main函数（main()）是必须的，C语言编译器不会知道你 ~~爸叫马云，你妈叫董明珠~~😂，它只知道看准main就执行里面的代码。~~很狂，属于是目中无人的狂~~💢。

所以我们就要创建一个名为main的函数，实际上这是必须的，C语言会找到它，找到这个名为mian的入口，如果一个代码中没有main函数，C就会报错，因为它找不到入口，没办法进入你冰冷的内心💔

而你告诉了它，这里使用的方法是 —— 「 int 」。int正中翻译是 —— 「 整形 」

所以当代码结束后，就会返回一个整形，就是0或1（这里后面作解释）

第三行：这里其实是第二行的大括号「 { } 」里面的东西，其实main( )后的「 { 」符，可以放到第四行，第四行放到第五行，第六行....

这样会使得代码更易看，而我的写法可以让代码更加简洁。~~可以提高逼格。~~

这被称为一个代码块，而代码就是这样一块块执行的，当我们回首这个代码块，就会发现里面只有短短的一行代码，一个来自标准库的函数 —— Printf （打印函数），它会依照你所输入/给出 来 打印出来内容。

Printf：「我阐述你的梦」

好了，此时我们执行代码，它就会返回一个 「“你好吗！”」，它向来说一不二。~~是你往后余生里的很好伴侣~~❤️ 不过，那个碍眼又黄黄的 「 \n 」为何许人也？绿色的【你好吗】- 黄色的【\n】是否就代表着它们有着不可告人的伦理故事？

事实上，C语言大量运用这些奇奇怪怪的符号，~~来让自己看起来很独特~~，其实这个【\n】是换行的意思。而上面的代码事实上会在执行后的终端输出以下内容：

```
你好吗!
（由\n带来的换了个行）
```

如果阁下拿出这段代码C语言又该如何应对？

```
printf("I'm");
printf("sb");
```

它会给你打印出下面的内容

```
I'msb
```

诶不是，老子的换行呢？？？。事实上无论你写多少行都不会换行，因为它没看到【\n】这位小可爱，它这个说一不二的性格不敢给你换行，怕你家暴它❤️。

所以，下面两段代码其实是等价的：

```
print("I'm\n");
print(" sb");

// 或者

print("I'm\n sb");
```

上面都会输出

```
I'm
  sb
```

但是，它们虽输出都一样，但不代表长头发就会是你妈，所以呢

第一段事实上是 先输出 - I'm(换行)，再输出 - (藏了个小空格)sb

第二段的代码事实上是 直接输出 - I'm(换行) - 接上后面的 (空格)sb

这两段代码旨在告诉你“我是傻逼！”，因为每段代码其实后面都必须加上「；」符，这样C才知道，哦，这一段结束了啊。不然这小兔崽子就会无视你，继续执行代码，直到编译器给你抛出个红红的 【你错了，哥。😂】

第四行：而第四行，就是为了闭合上上面那个 梦开始的「{」符了。结束这个代码块。

但我总感觉代码少了点什么...哦！int什么来头还没介绍，它返回的0/1干啥用的，告诉你这段main函数是0还是1？是，但不完全是，计算机不搞人类基佬那一套。

C语言的0/1是啥？C语言不会是基佬用的语言吧？

如果你对电脑有所了解，你就会知道电脑它唯一懂的就是那堆名为二进制的东西，就是一群0和1，当然，这里不是指基佬里面的0/1，而是很单纯的数字0/1，但单纯得来，又不太单纯，啊！当然，我不是说它是基佬。请看二进制的维基百科解答：

> **二进制**（英语：binary）在数学和数字电路中指以2为底数的记数系统，以2为基数代表系统是二进位制的。这一系统中，通常用两个不同的数字0和1来表示。数字电子电路中，逻辑门直接采用了二进制，因此现代的计算机和依赖计算机的设备里都用到二进制。每个数字称为一个比特（二进制位）或比特（Bit，Binary digit 的缩写）。
>
> ## 历史
>
> 现代的二进制记数系统由戈特弗里德·莱布尼茨于1679年设计，在他1703年发表的文章《论只使用符号0和1的二进制算术，兼论其用途及它赋予伏羲所使用的古老图形的意义》[1]出现。与二进制数相关的系统在一些更早的文化中也有出现，包括古埃及、古代中国、古印度以及太平洋岛原住民文明。其中，古代中国的《易经》尤其引起了莱布尼茨的联想。

历史总教导人们，看待事情不要非黑即白，非对即错。但是C语言的世界（或者说编程的世界）都是「不是对，就是错」。

你说你挺喜欢我的，但又爱他是几个意思？渣男，你一点都不爱C语言。0/1的精髓一点都没学进去，有他没我！有我没他！

所以其实，「0/1」是用来表示「假/真」的（0代表假，非0代表真。非0的意思是，不是0就是真！但是0就代表假。）

```c
#include <stdio.h>
int main() {
    printf("你好吗！ \n");
    return 0;
}
```

多么优美标准的写法啊，但是别忘了这操蛋的「;」结束符

于是，int所返回的0/1，此时就对应了return的0（记得不要落下「；」结束符。这里是否代表返回一个错误呢？

并非如此，在int和对应的return0;中，0事实上在这里代表成功，而不是假！

而代码执行到如此时，（int）程序就会返回对比0/1，如果int返回0则代表成功，而1却代表失败！！这是和真假不同的

> `int` 表示 `main` 函数将返回一个整数值，通常用于表示程序的执行状态。`return 0` 语句表示 `main` 函数执行成功并返回了一个值为0的状态码，这通常被视为程序成功执行的标志。如果程序遇到错误或异常情况，`main` 函数可以使用 `return` 语句返回其他整数值，以表示不同的错误状态或异常情况。因此，`int main` 和 `return 0` 在C语言中用于表示程序的起点和执行状态。
>
> 在C语言中，一般约定使用整数返回值来表示程序执行状态，其中非零值通常被视为表示"真"或"失败"，而值为0通常被视为表示"假"或"成功"。因此，当 `main` 函数返回0时，通常表示程序成功执行完毕。
>
> `return 0`：这是 `main` 函数内部的一条语句，它表示函数执行成功并返回了一个整数值为0的状态码。在C语言中，约定以0表示程序成功执行完毕。这样的约定使得其他程序（如shell脚本等）可以根据 `main` 函数的返回值来判断程序的执行状态，并作出相应的处理。

ok啊，这边也是说得差不多，是时候搬点教科书等等上古代码出来害害人了。

```
1  #include <stdio.h>
2  void main() {
3      printf("你好吗！\n");
4      return 0;
5  }
```

*main "的返回类型不是 "int"（可修复）

Return type of 'main' is not 'int' (fix available)

void 函数 "main "不应返回值

Void function 'main' should not return a value

（上面代码，划掉return0；或者将void改成int可解决报错）

书上喜欢用void，而不带return0;，事实上这种写法会导致很多错误，我下面引用一个说明

> 在 C 和 C++ 中，**不接收任何参数也不返回任何信息的主函数原型为*“void main(void)”***。可能正是因为这个，所以很多人都误认为如果不需要程序返回任何信息时可以把 main 函数定义成 *void main(void)* 。**然而这是错误的！** main 函数的返回值应该定义为 int 类型，C 和 C++ 标准中都是这样规定的。虽然在一些编译器中，*void main *可以通过编译（如 vc6），但并非所有编译器都支持 *void main* ，因为标准中从来没有定义过 *void main* 。g++3.2 中如果 main 函数的返回值不是 int 类型，就根本通不过编译。而 gcc3.2 则会发出警告。**所以，如果你想你的程序拥有很好的**

> **可移植性，请一定要用 *int main* 。**

ok啊，对于C的基本语法和概念现在就告一段落了！

2.C语言的变量常量和数据类型

C语言中有几种基本的数据类型，包括整型、浮点型、字符型和空类型。这些数据类型可以通过关键字来定义和使用：

数据类型	C语言内	使用：
整型	int	一般定义**整数**用：int xxx = 10;
浮点型	float 或 double	一般定义**浮点数用(其实就是小数啦)**：float xxx = 3.14;
字符型	char	一般用来定义**字符串(其实就是定义文字用的)**：char xxx = 'A';
空类型	void	**表示为空**，可以通过 void func()来定义（当函数无返回值，或者给指针类型使用）

或许你可以很好理解int，但是浮点型（其实就是小数）里面的float和double又是什么呢?

float翻译过来很简单，就是浮动的意思，就是浮点数的英文啦（小数），而double又是什么呢，这个翻译个来是两倍的意思，双的意思。

就是说，double要比float要精准，那么double，代价是什么。“d比f” 占用的内存条空间（下面用内存代称）要大！ 但是 “d” 这厮可以表达更精准的数字，啊！但也不是说 “f” 就不精准了，而是 “d” 可以存储大约 15-16 位有效数字，表示范围约为 10^308。

而 “f” 呢只能存储大约 6-7 位有效数字，表示范围约为 10^38，但是一般情况下是完全够用的。

> **float 和 double 都是C语言中用于存储浮点数的数据类型。它们之间的主要区别在于存储范围和精度：**
>
> 1. ***float***：*单精度浮点数，通常占用内存4个字节，用于存储大约 6-7 位有效数字，表示范围约为 10^38。*
> 2. ***double***：*双精度浮点数，通常占用内存8个字节，用于存储大约 15-16 位有效数字，表示范围约为 10^308。*
>
> *因此，**double 类型** 比 float 类型具有更高的精度和存储范围，适合需要更高精度计算的场景，比如科学计算、工程应用等。* 而**float 类型** 则更适合于内存有限的环境或者对精度要求不是特别高的场景。**

这里需要注意的就是Char（字符型）了，由于过去，作者并没想到这个C语言会如此伟大，全球范围内都在大规模使用，所以在函数定义时，是没有针对 *中文，日文，韩文* 进行优化。~~因为亚洲人实在太大了~~❤️ 平常英语一个字符（比如说A），只占一个字符。但是中文它们不一样，太屌了，以至于一个汉字就占两个字符，洋带人属于是太看得起我们天朝上国了❤️

但是Char（字符型）的定义，本身就有很多坑在里面，那我们先一一来讲解，下面先讲如何定义一个int类型和浮点类型，并打印。

C语言中常见的格式化占位符如下：我们需要用到下面的占位符给该死的变量占个位置，不然它还不想出来了。md

占位符	类型	描述
%d	int	输出十进制有符号整数
%u	unsigned int	输出十进制无符号整数
%o	unsigned int	输出八进制数
%x	unsigned int	输出小写十六进制数
%X	unsigned int	输出大写十六进制数
%f	float 或 double	输出浮点数
%e	float 或 double	输出用科学计数法表示的浮点数
%g	float 或 double	自动选择 %f 或 %e 格式
%c	char	输出单个字符
%s	char*	输出字符串
%p	void*	输出指针地址
%lu	unsigned long int	输出无符号长整形
%lld	long long int	输出长长整形

还有其他几个类型说明符，但不是非常常用，如 `%lu`、`%lld` 等。不同的类型说明符需要和相应的数据类型一起使用，否则程序可能会出现运行时错误。

```c
#include <stdio.h>
int main() {
    int age = 21;
    float credit = 3.4;

    printf("我的年龄是%d，学分是%f", age,credit);

    return 0;
}
```

红线第一行（第三行）：告诉C语言，有一个age（*自己瞎写的，但是这个英文单词的中文是年龄的意思，但是这个age你可以随意写，但是不建议，因为我怕你明天看着自己定义的a1=21，挠破脑袋也想不起来原来自己定义的这个a1是年龄，当然啊！你别直接 年龄=21 啊，C看不懂*），而这个age是int（整数类型），而且是等于21的。

红线第二行（第四行）：告诉C语言，有一个credit（*学分*），他的类型是浮点类型（小数），而且它等于3.4

红框第三行（第六行）：（*无视那个黄色的框框，是代码编辑器告诉我，我用了中文符号*）我用了一个%d（*int整数类型占位符*），%f（*float 或 double 浮点类型占位符*）。给age（年龄）和credit（学分）这两个变量占好位置后，下面就得发请柬请它俩b东西出来了。于是我用「""」把这段话框起来后，加了个逗号，就跑去嬉皮笑脸请它俩出来了，记得啊！记得啊！请完一个出来，记得加上个「，」逗号再请另一个，否则它们会觉得你不给它们脸，就不出来了。还有，记得类型占位符别搞错，就像美国有59种性别，你怎么就敢说他不是男的就是女的！搞错的话，它就会当场跟你翻脸，就算你把它请过来了，它也不会露脸，不然就露个不明所以的“数字中指（*意义不明的数据输出*）”鄙视你

执行返回内容：

```
我的年龄是21，学分是3.400000
```

此时可能你会惊呼，啥玩意，咋3.4后面还带了那么多崽子？？

哈哈，这是因为%f精准到小数6位的，你可以不严谨地理解，即便我的屁股只能坐一个凳子，我也得抢6个回来，不为别的，只为告诉你，我做得到😈，那么好，我们该如何制裁它？只允许它输出精准到两位小数点就好了？

我们可以用这一段代码:

```c
#include <stdio.h>
int main() {
  int age = 21;
  float credit = 3.4;

  printf("我的年龄是%d，学分是%.2f",age,credit);

  return 0;
}
```

此时的输出返回内容就会是：

```
我的年龄是21，学分是3.40
```

小样，我还治不了你了？😄。Oh Shit💩. 我手抖输入成了

```c
printf("我的年龄是%d，学分是%.20f",age,credit);
```

此时它是否会输出20位小数？是，又不完全是，打开了，但又没完全打开。它会输出匪夷所思的：

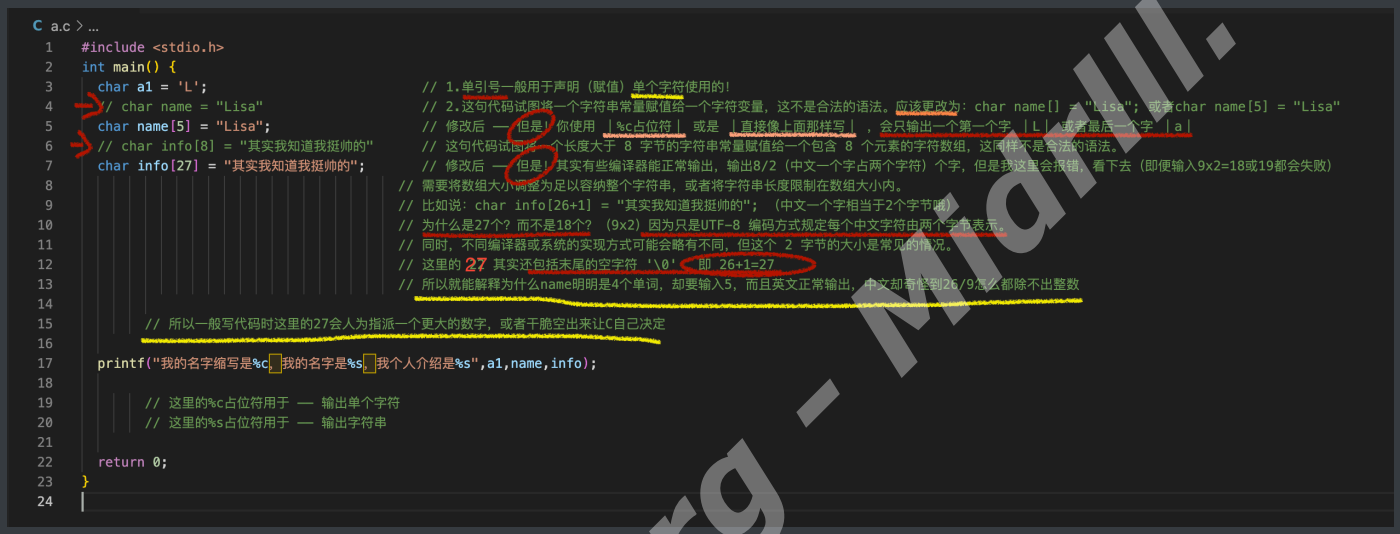
```
我的年龄是21，学分是3.40000009536743164062
```

尼玛，这是为何？？看下面

> %.20f 表示输出 credit 小数精准到20位的值。然而，对于一些浮点数，二进制表示法不能准确地表示小数部分，因此浮点数的存储值可能不是你期望的值，这也将影响最终的输出结果。
>
> 在 Python 中，有一个比较常见的类似问题，即使用 float 类型时，将 0.1 乘以 10 再乘以 10 可能会得到 1.0000000000000001 而不是我们期望的 1.0。这是因为浮点类型无法准确表示十进制数字，只能近似表示，因此在某些情况下可能会出现精度丢失的问题。

> 可以使用 double 类型来解决该问题，因为 double 类型提供了更多的存储空间来表示更精确的数字。
>
> 如果只是输出小数部分小于20位精度的浮点数，可以省略精度说明符 %.20f（*直接写%f*），这将在默认的小数点后输出六位浮点数，其结果常常已经足够了。

ok啊，这边对于占位，常量声明和屁股大等等概念就说到这了，下面我们来说说Char（*字符类型*）😡



```c
#include <stdio.h>
int main() {
  char a1 = 'L';                    // 1.单引号一般用于声明（赋值）单个字符使用的!
  // char name = "Lisa"             // 2.这句代码试图将一个字符串常量赋值给一个字符变量，这不是合法的语法。应该更改为: char name[] = "Lisa"; 或者char name[5] = "Lisa"
  char name[5] = "Lisa";            // 修改后 —— 但是！你使用 | %c占位符 | 或是 | 直接像上面那样写 | ，会只输出一个第一个字 | L | 或者最后一个字 | a |
  // char info[8] = "其实我知道我挺帅的"  // 这句代码试图将一个长度大于 8 字节的字符串常量赋值给一个包含 8 个元素的字符数组，这同样不是合法的语法。
  char info[27] = "其实我知道我挺帅的";  // 修改后 —— 但是！其实有些编译器能正常输出，输出8/2（中文一个字占两个字符）个字，但是我这里会报错，看下去（即便输入9x2=18或19都会失败）
                                    // 需要将数组大小调整为足以容纳整个字符串，或者将字符串长度限制在数组大小内。
                                    // 比如说：char info[26+1] = "其实我知道我挺帅的";  （中文一个字相当于2个字节哦）
                                    // 为什么是27个？而不是18个？（9x2）因为只是UTF-8 编码方式规定每个中文字符由两个字节表示。
                                    // 同时，不同编译器或系统的实现方式可能会略有不同，但这个 2 字节的大小是常见的情况。
                                    // 这里的 27 其实还包括末尾的空字符 '\0'，即 26+1=27
                                    // 所以就能解释为什么name明明是4个单词，却要输入5，而且英文正常输出，中文却奇怪到26/9怎么都除不出整数

  // 所以一般写代码时这里的27会人为指派一个更大的数字，或者干脆空出来让C自己决定

  printf("我的名字缩写是%c，我的名字是%s，我个人介绍是%s",a1,name,info);

  // 这里的%c占位符用于 —— 输出单个字符
  // 这里的%s占位符用于 —— 输出字符串

  return 0;
}
```

注意事项和奇奇怪怪的设定都写在了上方代码中，C语言坑很多～你需要做到的记住正确的写法，而不是记住错误的写法，因为你会发现，换了台电脑，另一个工作环境，结果错误语法它又能跑起来了，真是见鬼。

这段代码运行后会输出：

```
我的名字缩写是L，我的名字是Lisa，我个人介绍是其实我知道我挺帅的
```

贴出个源代码给你瞻仰一下C的神奇💥：

```c
#include <stdio.h>
int main() {
  char a1 = 'L';                    // 1.单引号一般用于声明（赋值）单个字符使用的!
  // char name = "Lisa"             // 2.这句代码试图将一个字符串常量赋值给一个字符变量，这不是合法的语法。应该更改为: char name[] = "Lisa"; 或者char name[5] = "Lisa"
```

```
  char name[5] = "Lisa";                    // 修改后 —— 但是！你使用 |%c占位符| 或是 |直接像上面那样写| ，会只输出一个第一个字 |L| 或者最后一个字 |a|
  // char info[8] = "其实我知道我挺帅的"     // 这句代码试图将一个长度大于 8 字节的字符串常量赋值给一个包含 8 个元素的字符数组，这同样不是合法的语法。
  char info[27] = "其实我知道我挺帅的";     // 修改后 —— 但是！其实有些编译器能正常输出，输出8/2（中文一个字占两个字符）个字，但是我这里会报错，看下去（即便输入9x2=18或19都会失败）
                                            // 需要将数组大小调整为足以容纳整个字符串，或者将字符串长度限制在数组大小内。
                                            // 比如说：char info[26+1] = "其实我知道我挺帅的"; （中文一个字相当于2个字节哦）
                                            // 为什么是27个？而不是18个？（9x2）因为只是UTF-8 编码方式规定每个中文字符由两个字节表示。
                                            // 同时，不同编译器或系统的实现方式可能会略有不同，但这个 2 字节的大小是常见的情况。
                                            // 这里的 22 其实还包括末尾的空字符 '\0' ，即 26+1=27
                                            // 所以就能解释为什么name明明是4个单词，却要输入5，而且英文正常输出，中文却奇怪到26/9怎么都除不出整数

        // 所以一般写代码时这里的27会人为指派一个更大的数字，或者干脆空出来让C自己决定

  printf("我的名字缩写是%c，我的名字是%s，我个人介绍是%s",a1,name,info);

        // 这里的%c占位符用于 —— 输出单个字符
        // 这里的%s占位符用于 —— 输出字符串

  return 0;
}
```

好，不愧是我们的C语言，还是那么会玩🤪

其实我们还可以这样玩：

```
#include <stdio.h>
int main() {
  int age,lengths;
  char* name;

  age = 20;
  lengths = 18;
  name = "Lisa";

  printf("我的名字叫：%s，我年龄%d，长度%d",name,age,lengths);

  return 0;
}
```

上面的代码将输出：

```
我的名字叫：Lisa，我年龄20，长度18
```

啊不是，我能看出来先声明后赋值这种先斩后奏的戏码，但是这个「char*」的「*」是个嘛玩意？

> 如果将 `char* name` 改为 `char name[]` 或 `char* name = "Lisa"` 是可以正常编译和运行的，这是因为这两种定义方式都为 `name` 分配了存储字符串的空间（而不是只分配了一个指针变量），这样指针变量 `name` 就可以指向该数组或字符串常量了。如果你使用的是 `char* name`，则需要在使用 `name` 实际之前指向一个有效的字符串常量（后面的name = "Lisa";），否则会引发错误。
>
> *为什么char name就不可以？*

> 在 C 语言中，`char name` 定义的是一个 `char` 类型的变量，存储一个字符的数据。如果您将其用于存储字符串，会导致以下问题：
>
> 1. *`char name` 只能存储一个字符，无法存储多个字符构成的字符串。*
> 2. *C 语言中的字符串实际上是以 `char` 类型的字符数组的形式存在的，需要分配足够的存储空间来存储整个字符串。如果您将 `char name` 用于存储字符串，需要在定义时就预先分配好存储空间，否则会导致缓冲区溢出等内存错误。*
>
> 因此，如果需要定义一个字符数组来存储字符串，您可以使用 `char name[]` 或 `char* name` 。
>
> 对于 `char* name` ，它是一个指向字符的指针变量，指向一个字符串常量或者字符数组的首字符，这样的定义方式更加灵活，可以适应不同长度的字符串。同时，在使用完毕后也可以通过 `free()` 函数释放动态分配的空间。
>
> 总之，`char name` 适合用于单个字符的存储，而 `char name[]` 或 `char* name` 则更适合用于字符串的存储

ok，那么常量变量和数据类型，操蛋的Char类型也就告一段落了。

下面我们将进入对加减乘除运算和真假美猴王（真或假）的逻辑研究！

3.C语言的运算和真真假假，假假真真的逻辑运算

1+1=2？真还是假啊。我不信（🤪）

C 语言中常用的运算符如下：

1.算术运算符 用于进行基本的数学运算，包括加减乘除和求余等。常用的算术运算符包括：`+`、`-`、`*`、`/`、`%` 等。*(加减乘除求余)*

```
int a = 5, b = 2;

int sum = a + b;    // 加法运算，结果为 7
int diff = a - b;   // 减法运算，结果为 3
int prod = a * b;   // 乘法运算，结果为 10
int quo = a / b;    // 整除运算，结果为 2
int mod = a % b;    // 求余运算，结果为 1（ 5/2=1（余））
```

2.赋值运算符 将右侧的数值或表达式的值赋值给左侧的变量。常用的赋值运算符包括：`=`，`+=`，`-=`，`*=`，`/=` 和 `%=` 等。*（赋值，加某个数再赋值，减某个数再赋值，乘某.，除某.，取某数余...）*

```
int a = 5;

a += 2;  // 将 a 的值加 2 并重新赋值给 a，相当于 a = a + 2，最终结果为 7
a -= 3;  // 将 a 的值减 3 并重新赋值给 a，相当于 a = a - 3，最终结果为 4
a *= 2;  // 将 a 的值乘 2 并重新赋值给 a，相当于 a = a * 2，最终结果为 8
a /= 2;  // 将 a 的值除以 2 并重新赋值给 a，相当于 a = a / 2，最终结果为 4
a %= 3;  // 将 a 的值对 3 取模并重新赋值给 a，相当于 a = a % 3，最终结果为 2（取余数）
```

3.关系运算符 用于比较两个变量或表达式之间的大小关系，返回布尔值（`0` 或 `1`|假 或 `真`）。常用的关系运算符包括：`==`，`!=`，`<`，`>`，`<=` 和 `>=` 等。*（全等于判断是否一样，不等于，小于，大于，小于等于，大于等于。）*

```
int a = 5, b = 2;

//下面左边的变量被赋予了右边运算后的0/1（假/真）
int result1 = (a == b);   // 判断 a 是否等于 b，结果为 0      |假   （两个等于叫全等，是比较用的，而一个的话，是赋值用的）
int result2 = (a != b);   // 判断 a 是否不等于 b，结果为 1    |真
int result3 = (a < b);    // 判断 a 是否小于 b，结果为 0      |假
int result4 = (a > b);    // 判断 a 是否大于 b，结果为 1      |真
int result5 = (a <= b);   // 判断 a 是否小于等于 b，结果为 0   |假
int result6 = (a >= b);   // 判断 a 是否大于等于 b，结果为 1   |真
```

4.逻辑运算符 用于结合两个或多个布尔类型的表达式，返回布尔值（0 或 1 | 假 或 真）。常用的逻辑运算符包括：`&&`，`||` 和 `!` 等。

（「&&」：1与2的意思，必须有一个为真的。「||」：1或2的意思，有一个为真即可。「!」：取一个反的值，颠倒黑白用）

```
int a = 5, b = 2;

int result1 = (a > 2 && b < 1);  // 判断 a 是否大于 2 且 b 是否小于 1，结果为 0        |假（且 - 必须两个都为真）
int result2 = (a > 2 || b < 1);  // 判断 a 是否大于 2 或 b 是否小于 1，结果为 1        |真（或 - 一个为真即可）
int result3 = !(a > 2);          // 取反 a 是否大于 2 的结果为布尔型的相反值，结果为 0   |假（真都能给你「!」成假的）
```

5.位运算符 用于对数据的二进制位进行操作，包括按位与、按位或、按位异或等。常用的位运算符包括：`&`，`|`，`^`，`~`，`<<` 和 `>>` 等。

```c
unsigned int a = 0x5, b = 0x3;

unsigned int result1 = a & b;   // 位与运算，结果为 1
unsigned int result2 = a | b;   // 位或运算，结果为 7
unsigned int result3 = a ^ b;   // 位异或运算，结果为 6
unsigned int result4 = ~a;      // 按位取反运算，结果为 0xFFFFFFFA
unsigned int result5 = a << 1;  // 左移一位，相当于乘以 2，结果为 0xA
unsigned int result6 = b >> 1;  // 右移一位，相当于除以 2，结果为 0x1
```

6.条件运算符 用于在两个表达式之间进行选择，根据判定条件的真假返回其中之一的值。常用的条件运算符为：`?` 和 `:`。

```c
int a = 5, b = 2;

int max = (a > b) ? a : b;  // 判断 a 是否大于 b，如果为真则返回 a，否则返回 b。该处结果为 5
```

7.其他运算符 包括地址运算符 `&`、指针运算符 `*`、sizeof 运算符等。 *(&是用于将右边a的内存地址值赋给左边的指针变量的，`*ptr` 是用于解除上面那个赋值的，因为打印出来人类也看不懂内存地址里面有什么，所以解除后，结果就会恢复成5啦。最后一个经常用到，可以用来判断某个数组，变量等到底占用了多少字节数。)*

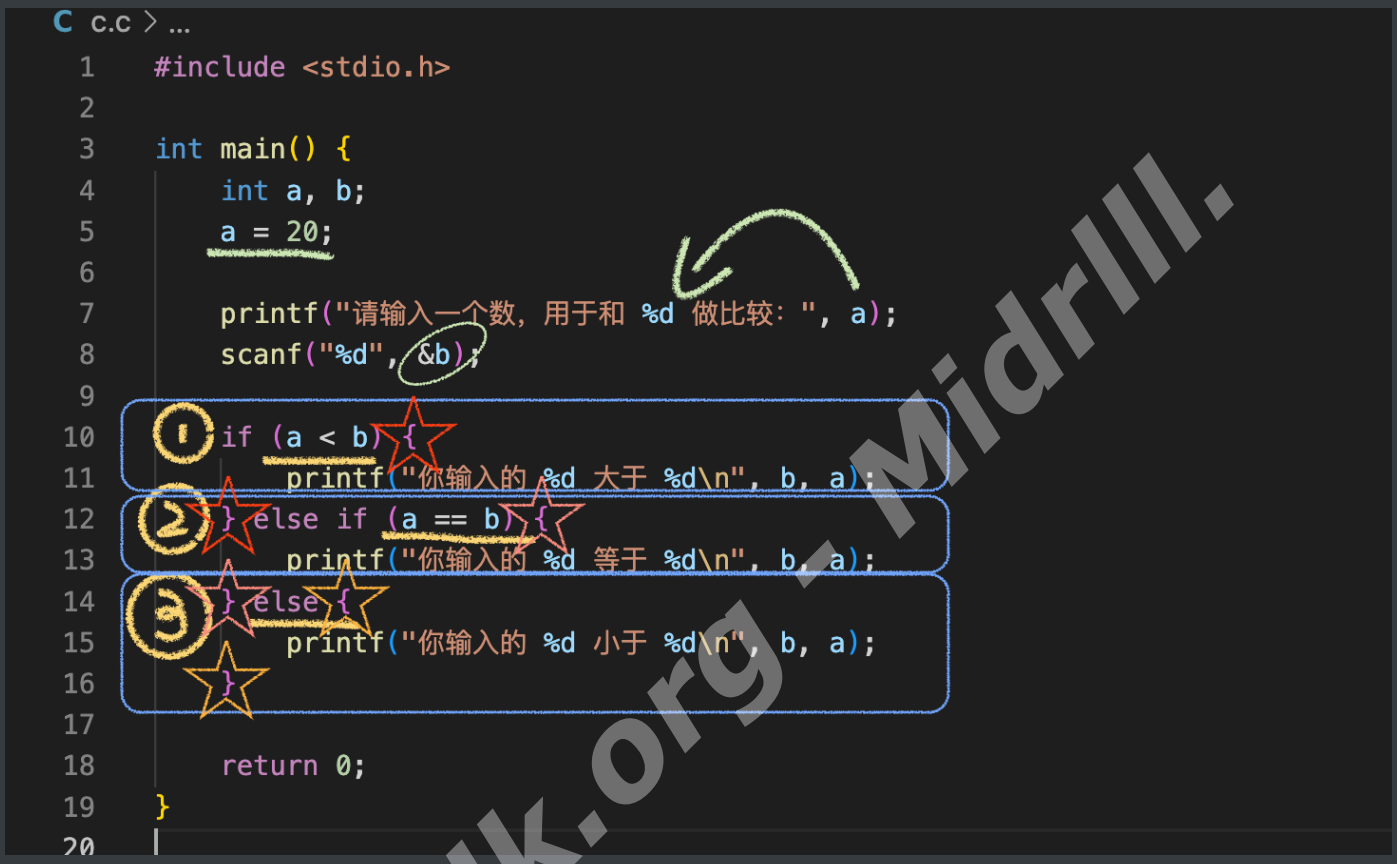
```c
int a = 5;
int* ptr = &a;              // 取得 a 的地址并赋值给指针变量 ptr
                            // &a 取得的是变量 a 的内存地址，即指向变量 a 在内存中的位置的地址
int val = *ptr;             // 解除指针变量 ptr 并获取其指向的值，结果为 5

int size = sizeof(int);     // 获取 int 类型占用的字节数，结果为 4
```

这里只列举了 C 语言中的一些常用运算符，还有一些其他运算符和运算符的组合用法，需要根据实际情况进行理解和使用。

到这里，我们对C语言或者是大部分语言的运算方法和原理都知道得差不多了，下面我们就来说说，真假到底有何作用

4.万千世界，真真假假，假假真真，狠活都是假的，只有平平淡淡才是真的

IF 判断



我们看上面的代码，其中星星对应的是一个「{}」代码块，我这样写是为了简洁，事实上，上面的代码等价于下面的代码，而有时候可能下面的代码更简单易看。🤓

```c
#include <stdio.h>

int main() {
    int a, b;
    a = 20;

    printf("请输入一个数，用于和 %d 做比较：", a);      // 这里是做一次打印，告诉你要做什么，不然打开黑漆漆一片，根本不知道代
// 码有没有跑起来，也不知道该做什么。这里直接将a变量填充到占位符里
// 面。int类型用%d，很合理好吧。ok啊
```

```
                                                    // 如果你手动写用于与20做
比较，那你想改20，你就得改a=xx;这个赋值代
                                                    // 码。但是，不还得手动改
下面打印那个20？所以直接填充，免得b事多多。
    scanf("%d", &b);                            // 等待陛下您给他一个数，「%d」限定了
int，上面我们定义的也是int类型
                                                    // 这里的b，记得要加上「&」哦，因为
scanf它需要给贵宾【b】在内存开个房间。
                                                    // 等待【b】的入住，入住的【b】是个黑
鬼（不是）（我的意思是，是，，是int类型）
                                                    // 至于是哪位黑鬼，不是，至于是多少，
就得由殿下您决定了

    if (a < b)                                      // if(条件)，这里就是 -
如果a变量<(小于)你请来的黑鬼【b】(bushi)
                                                    // (条件为真的话)则执行下
面的代码
                                                    // (a上面定义为20，如果你
输入大于20，表达式就会为真)
    {
        printf("你输入的 %d 大于 %d\n", b, a);      // 条件为真，执行这个代码
块，所以就会输出
                                                    // “你输入的 xx 大于 20
（换行）”
    }
    else if (a == b)                                // 这里是「否则 如果」的意
思，否则如果 - 20与你输入的数字完全相等
    {                                               // 你输入了20，所以条件为
真，上面的当然就不是真的啦（上面变成假的了）
        printf("你输入的 %d 等于 %d\n", b, a);      // 所以这里会打印出：“你输
入的【20】(b)等于【20】(a)(换行)”
    }
    else                                            // 这里是「否则」的意思，
因为这里没有任何判断的表达式
    {                                               // 所以当上面都不成立，都
为假的时候，就会执行这个代码块
```

```
        printf("你输入的 %d 小于 %d\n", b, a);
    }
                    // if语句判断完毕，执行完一次后就会跳出，继续执行下面的代码哦
    return 0;
}
```

Switch 判断

我们已经知道 if 语句判断的使用方法，下面我们来看看Switch判断的方法，Switch判断一般用于判断多个结果。

Switch 一般只支持【整形】和【字符类型】，【枚举类型】比较，我们可以看看下面的例子：

```
#include <stdio.h>

int main() {
    int a;

    printf("我喜欢1月和2月，你呢？\n告诉我：");
    scanf("%d", &a);

    switch (a)
    {
        case 1:
            printf("不错，一月份，万物复苏");
            break;
        case 2:
            printf("很好，二月份，生机勃勃");
            break;

        default:
            printf("你在胡说什么，我发誓我会用山姆叔叔的臭皮靴狠狠踢你的屁股");
            break;
    }
```

```
    return 0;
}
```

上面的例子，当你运行时，它会打印出：

```
我喜欢1月和2月，你呢？（换行）
告诉我：
```

之后scanf函数会等待你的输入并回车，之后将值赋给「 a 」变量，此时下面的Switch循环将会起作用。

它获得了你输入的「 a 」的内容后，会交给它的特工【case】，此时，特工就会查看「 a 」的身份证件，透过变量本身看到它内心到底是红的还是黑的。如果你输入的是「 1 」，那么它就会被特工抓到第一个审讯室里面，而那个「 : 」里面就是一个审讯室，但我们很良心，只是打印了一句 *“不错，一月份，万物复苏”* 就把它放了出来(break)，如果你不放人，很有可能会导致“代码叛乱”，哪里有压迫，哪里就有起来反了nm的，所以记得哦，记得在必要的地方放人哦 *（否则会一直卡在循环里出不来，我们得到了我们想要的，警告了它，或者其他事情完成后，记得加上一个“放人代码”(break)）*

好的，你可以试试上面的代码！😋

如果你输入的是1，就会输出 - *不错，一月份，万物复苏 （break 跳出 Switch）*

如果你输入的是2，就会输出 - *很好，二月份，生机勃勃 （break 跳出 Switch）*

如果你输入的不是1也不是2，就会输出 - *你在胡说什么，我发誓我会用山姆叔叔的臭皮靴狠狠踢你的屁股（break 跳出 Switch）*

至于字符类型判断呢 😨❓

记得哦， `char a[] = "Lisa"` 是字符串！用的还是双引号（用的占位符是%s）

而字符类型则是： `char a = 'L'` 这个才是字符类型（单个字母，用的占位符是%c）

5.循环！生活日复一日的循环..

Ok啊，这边也是说到了循环结构体好吧，我说，从0数到100很费劲吧，至少得花个50秒吧？🤔

再或者是困扰你小学初中的 1+2+3+....+99+100 的数学题？在循环结构里，机器的速度能给你快出火花来🔥

大概就是，你给它设置一个机械的重复的条件（你也可以多加几个，在循环里面套一个循环，就像if判断里面依旧可以套个if判断，相当于是你先过滤了一次米饭（大部分东西），再把里面的稀粥（小部分东西）也过滤掉，之后你就得到了一碗纯净的米汤！🥣

所以说呢，要是吴1凡在监狱里有这种循环神器，缝纫机都能踩出火🔥来咯。

While 循环（*别搞错，不是Switch。Switch是判断，这个是循环*🤣）

我们看下面的代码：

```
#include <stdio.h>

int main() {
    int a;
    a = 1;

    while (a<100)
    {
        printf("%d \n",a);
        a++;
    }

    return 0;
}
```

这段代码就是一个很标准的While循环，我们给出一个条件，a<100，真吗？那必定是真啊！我的老baby

因为我在上面给a赋了个a=1的值，所以这个While后面的表达式肯定为True（真）咯

所以呢这个循环就会执行，代码块里有什么呢？打印「 a 」这个变量，还有一个换行，ok

之后a++，还记得吧，加它本身，就是 a = a + 1

这下代码块执行完了，回到While这里，a++了，此时a就等于2了嘛，但是不还是 a<100 吗？？？

所以呢，这个循环又会再执行，a++也将继续执行，直到 a不再小于100

尼玛，此时你会发现打印的最后一个数字是99

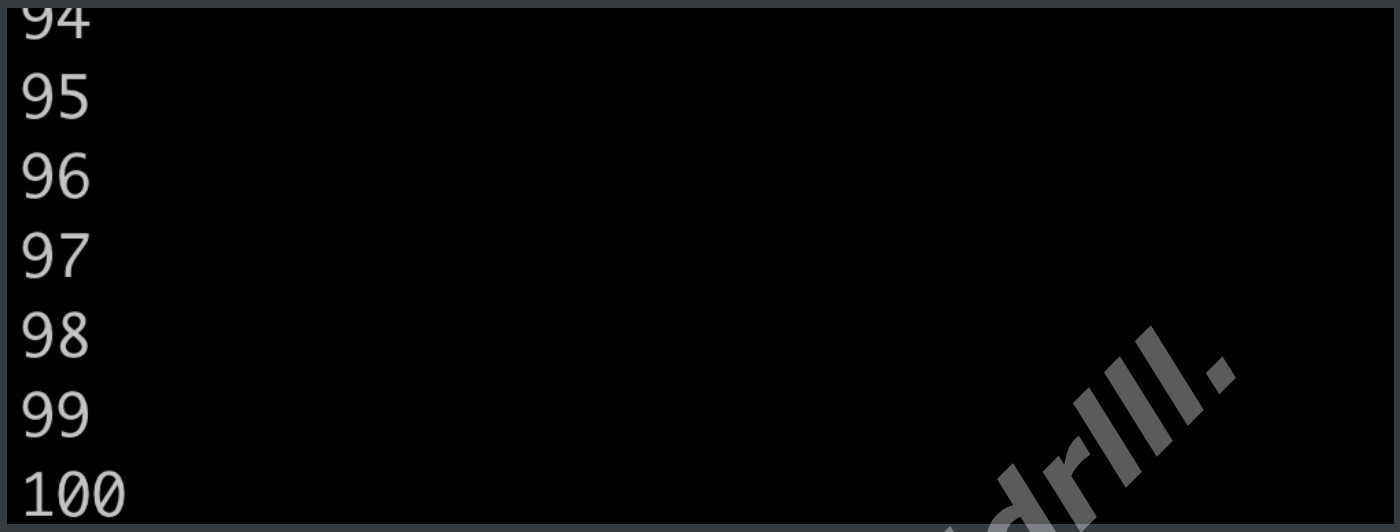
这就得考考你逻辑啦，当你值达到99时，依旧不为假，也就是说，a<100依旧成立，依旧为真，循环依旧生效

那就给你打印个「 99 」出来咯，但是呢！之后a++，这a不就成100了嘛？此时循环条件就不满足了，因为a不再少于100，a已经是100

那怎么破？你把a++放到打印之前试试？再或者猪b一点，把条件调成 `a<101`

这里我就不告诉你啦，你自己想想为什么把a++放打印前面就没问题了！

（但是其实最开始的「1」就不见了，你会发现会从「2」开始，这个时候你可以将a改成 - a=0，让a从0开始，思考一下为什么）



for循环

ok，我们这里也是说说for循环，看看它是怎么个事😡

for循环牛批很多了，因为你可以在for后面那个括号里面写完你的所有需求！

我们看一下下面的代码，事实上是等价于上面的代码的，多简洁有力

```
#include <stdio.h>

int main() {

    for (int a = 1; a < 100; a++)
    {
        printf("%d \n",a);
    }

    return 0;
}
```

但是！很不幸，上面的代码执行后依旧会输出到99后就停止哈哈哈，想想为什么

首先for的用法是这样的

`for（初始化你的对象(变量之类的)； 开出你的循环条件； 执行完一次循环你想让我做什么）`
（记得用「;」隔开三个表达式哦）

当上面For循环第二条表达式为真时，循环当然会执行，执行代码块里面的打印，之后完成代码块，它会执行For循环第三条表达式：对a这个变量进行++（也就是a=a+1）

是不是很简单粗暴？那我们最后只输出到99这个怎么破哈哈哈，那就把第二条表达式 `a<100` 改成 `a<101` 呗😂

嵌套循环 – 九九乘法表

九九乘法表也算是中国学生的传统艺能了，那我们如何使用While循环/For循环做一个出来呢?

这时，就得搬出我们的嵌套循环了！但是这b东西并没有多神秘，就是循环里面又套了个循环哈哈哈哈

那我们就先用While循环试试，弄一个出来

```
#include <stdio.h>

int main() {
    int x,y;
    x = 1;
    while (x<10)
    {
        y = 1;                                          // 这里重新赋值用哦，为了第二次循环，第二个循环的
                                                        // 条件表达式会重新变成真
        while (y<10)                                    // 该循环执行完，条件为假后，会返回到上面的循环执行
        {                                               // 直到上面的条件也变假，所以我才需要在上面弄个y=1
            printf("%d x %d = %d \n", x, y, x*y);      // 我直接在这里计算后再填充回去占位那里😋
```

```
            y++;                                                    // 不加这个
    会导致条件永远为真会死循环哦
        }
        printf("\n");                                               // 这个肯定
    不是多余的，你想想C语言多么说一不二❤️
        x++;
    }

    return 0;
}
```

我一个执行甩过去！

```
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63

8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72

9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
```

那么其实我们怎么理解嵌套循环里重新赋值y，循环第一层，又循环第二层，第二层循环完又循环第一层，第一层重新赋值，所以第二层又变成真，又循环呢？？？是不是耳朵都听肿了，我到底说了什么猪话

那我就简单跟你唠唠，大概就是「体育老师A」它让你跑【5】圈，实在太不是人了，之后跑到半路，「体育老师B」 又出来了，这小逼崽子让你跑【3】圈，但他或许不知道「体育老师A」已经让你跑【5】圈了，反正..谁在乎呢，没办法，那你只能先跑个【3】圈，给「体育老师B」交差，之后继续返回去跑你的「5」圈，之后你跑完那半圈，到第【4】圈了，结果第【4】圈跑了一半，byd「体育老师B」又舔着逼脸出来了，让你继续跑【3】。emm为什么这样呢，那大概就是【体育老师A】和【体育老师B】串通好的(第一个循环y=1重新赋值)，每次都让「体育老师B」出来骚扰你，直到「体育老师A」那【5】圈都跑完了，这下就没办法迫害你了，因为你连最开始那【5】圈都跑完了。🤔

试试我们的for循环

```c
#include <stdio.h>

int main() {

    for (int x = 1; x < 10; x++)
    {
        for (int y = 1; y < 10; y++)
        {
            printf("%d x %d = %d \n",x ,y ,x*y);
        }
        printf("\n");
    }

    return 0;
}
```

ok啊，害得是我们for循环：

```
6 x 9 = 54

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63

8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72

9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
```

这里的代码就不作注释了，你看看能看懂否？😄

do...while 循环

差点忘记我们冷门的do...while 循环了

那，do...while 循环何许人也？为何被殿下打入冷宫，今日才忽然想起来？

“do...while 循环事实上是个极度自我的家伙，特别喜欢先斩后奏 —— 就是不管条件是否成立，都会先执行一次。”

但..事实上，他真的是这样的人吗？

不是的，我们先看他的语法：

```
do
{
   要执行的代码;

}while( 条件表达式 );
```

比如下面的代码：

```
#include <stdio.h>

int main() {
    int a;
    a = 99;

    do
    {
        printf("当前的a的值是：%d 哦\n",a);
    }
    while( a > 999 );

    return 0;
}
```

上面的等价于我下面的代码，我认为下面的代码更加简洁：

```c
C d.c > ...
#include <stdio.h>

int main() {
    int a;
    a = 99;

    do{
        printf("当前的a的值是: %d 哦\n",a);
    }while( a > 999 );

    return 0;
}

```

是不是更简洁有力，但有时候脑子没转过来也挺蛋疼的，ok，那我们先解释一下这段代码。

为什么大家都说他先斩后奏呢，这真的不能怪他，因为你说「 do 」，让他先做，做完后，md，他才看到你后面给出的「 While 」条件，这时候无论后面的「 While 」是否成立，他都先执行了一次了。

我给「 a 」的定义是「 99 」，所以后面的 While 表达式 `a > 999` 肯定是不成立的，肯定是假的啦，但是无论是否真假，他都已经执行了一次，并打印出了：

```
当前的a的值是: 99 哦
```

这就是我们 do...while 循环 啦，所以他真的不是你想的那样的，有时候真的得留点时间给别人，好好读读别人的内心。🍂

6.函数

下面我们讲一下C语言的函数，C语言也差不多讲完了

或许你看到函数就已经开始瑟瑟发抖，初高中没少被函数支配吧，那种痛苦和恐惧，我懂

那么我们下面讲一下，函数到底是什么。其实函数只是个概念，非常非常简单，千万别害怕。使用它甚至可以让复杂的程序变得简单。

> 在 C 语言中，函数是一组可重复使用的代码块，提供了模块化、结构化的编程方式。函数的使用可以将程序的复杂度降低，同时也便于代码的重用和维护。

我们如何定义一个函数呢？函数又长什么b样？ 如下：

```
int multiply(int a, int b) {      // 说明是int类型，这个函数叫「multiply」，其中需要传入两个变量（「 , 」隔开）
    return a * b;                 // 返回「 a 乘以 b 」的值
}                                 // 闭合大括号，形成一个代码块
```

我们前面说过，C语言只会认准「 main( ) 」函数后执行里面的内容，是的，如果你没有在「 main( ) 」函数里面调用「 multiply( ) 」这个函数，那么「 multiply( ) 」函数就不会执行哦，记得，你说明了「 multiply(int a, int b) 」得传入一个int类型的「 a 」变量，和一个int类型的「 b 」变量。你可以传入分别名为「 x 」，「 y 」的int变量，而这个「 a 」，「 b 」是你传入后，告诉C，在这个函数代码块就叫「 a 」和「 b 」了。 ~~*大概是什么意思呢，就是你把你的孩子给别人照顾，吃人家的睡人家的，不把孩子改个名改个姓借人家玩两天说不过去吧？*~~

~~**那么如何把孩子借给别人玩两天呢？**~~ **那么我们如何调用函数呢？** 如下：

```
#include <stdio.h>

int multiply(int a, int b) {      // 说明是int类型，这个函数叫「multiply」，其中需要传入两个变量（「 , 」隔开）
    return a * b;                 // 返回「 a 乘以 b 」的值
}

int main() {
    int x = 3, y = 4, c;
    c = multiply(x, y);
    printf("x 和 y 的乘积是 %d\n", c);
    return 0;
}
```

Ok啊，上面的代码也是直接把 main( ) 的孩子「 x 」和「 y 」借给 multiply( ) 玩了一下，把他们乘起来后就还给我们的 main( ) 了。

下面上点强度：

函数参数传递

C 语言中的函数参数传递有两种方式：值传递和指针传递。值传递是指将函数调用中的实际参数值复制一份给形式参数，即在函数内部对形式参数的修改不会影响实际参数。指针传递是将实参的地址传递给形参，即在函数内部对形参的修改会影响实际参数。

例如：

```
void swap(int *p, int *q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

int main() {
    int a = 5, b = 3;
    printf("a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```


在这个例子中，我们定义了一个名为 `swap()` 的函数，它接受两个指向整型变量的指针作为参数。在 `swap()` 函数内，我们使用指针来交换两个整数的值。在 `main()` 函数中，我们声明了两个整数变量 `a` 和 `b`，并初始化它们的值为 5 和 3，然后我们调用 `swap()` 函数来交换它们的值，最后输出交换后的结果。

7.概念篇 - 本篇源自 ChatGPT 3.5-Turbo 数据 作修改和引用（仅本篇）

作用域 - 代码变量在某些区块有作用

是不是听起来很高大上？其实简单得一批

比如说下面的代码：

```
#include <stdio.h>

/* 全局变量声明 */
int g = 20;

int main ()
{
  /* 局部变量声明 */
  int g = 10;

  printf ("value of g = %d\n",  g);

    while( g == 10 ){
        /* While循环内局部变量声明 */
        int e = 12;
        printf("%d",e);
    }

  return 0;
}
```

这里全局定义了（在 main() 外面） `int g = 20` ，但是我们我们又在 main() 里面定义了 `int g = 10` ，所以其实a最终是「 =10 」的，之后我们这里引入一个叫 “生命周期” 的东西，于 main() 里面的 while 循环里，我们定义了 `int e = 12` 所以，当循环结束后，变量 `e` 的生命周期也就结束了，不再能在程序中被访问。 *（因为在 `while` 循环内部定义的变量 `e` 的作用域仅限于 `while` 循环的代码块 - 大括号内部。）*

是不是，其实作用域这种概念简单得一批，理解后能省下很多事。😎

一维数组

一维数组是最简单的数组类型，它只有一个维度，可以存储一组相同类型的数据。在 C 语言中，定义一维数组的语法格式为：

```
数据类型 数组名[数组长度];
```

例如，下面的代码定义了一个长度为 5 的整型数组：

```
int array[5];
```

一维数组在程序中的作用是方便处理同种数据类型的一组数据，可以根据数组下标来访问数组中的元素。数组下标从 0 开始，因此在访问第 n 个元素时，下标应该为 n-1。

例如，下面的代码给数组赋值并输出数组元素：

```
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    int i;

    for(i = 0; i < 5; i++) {
        printf("array[%d] = %d\n", i, array[i]);
    }

    return 0;
}
```

输出为：（几乎所有编程语言，都是从0数起的哦，而不是1）

```
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

二维数组

二维数组是由相同类型的元素组成的表格，它有两个维度，可以看做是一个由行和列组成的矩阵。在 C 语言中，定义二维数组的语法格式为：

```
数据类型 数组名[行数][列数];
```

例如，下面的代码定义了一个 2 行 3 列的整型数组：

```
int array[2][3];
```

二维数组在程序中的作用是方便处理二维数据或多维数据，可以根据行列下标来访问数组中的元素。访问二维数组时，需要用两个下标来确定所访问的元素位置，第一个下标表示行号，第二个下标表示列号，下标从 0 开始。

例如，下面的代码给数组赋值并输出数组元素：

```
#include <stdio.h>

int main() {
    int array[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int i, j;

    for(i = 0; i < 2; i++) {
        for(j = 0; j < 3; j++) {
            printf("array[%d][%d] = %d\n", i, j, array[i][j]);
        }
    }
```

```
    return 0;
}
```

输出为：

```
array[0][0] = 1
array[0][1] = 2
array[0][2] = 3
array[1][0] = 4
array[1][1] = 5
array[1][2] = 6
```

多维数组

多维数组是由相同类型的元素组成的表格，它有三个或以上的维度，可以看做是由二维或一维数组组成的数组数组。在 C 语言中，定义多维数组的语法格式为：

```
数据类型 数组名[第一维长度][第二维长度]...[第 n 维长度];
```

例如，下面的代码定义了一个 2 行 3 列 4 层的整型数组：

```
int array[2][3][4];
```

多维数组在程序中的作用是方便处理多维数据，可以根据任意个下标来访问数组中的元素。

例如，下面的代码给数组赋值并输出数组元素：

```
#include <stdio.h>

int main() {
    int array[2][3][4];
    int i, j, k;

    for(i = 0; i < 2; i++) {
```

```
            for(j = 0; j < 3; j++) {
                for(k = 0; k < 4; k++) {
                    array[i][j][k] = i * j * k;
                }
            }
        }

        for(i = 0; i < 2; i++) {
            for(j = 0; j < 3; j++) {
                for(k = 0; k < 4; k++) {
                    printf("array[%d][%d][%d] = %d\n", i, j, k, array[i][j]
[k]);
                }
            }
        }

        return 0;
    }
```

输出为：

```
    array[0][0][0] = 0
    array[0][0][1] = 0
    array[0][0][2] = 0
    array[0][0][3] = 0
    array[0][1][0] = 0
    array[0][1][1] = 0
    array[0][1][2] = 0
    array[0][1][3] = 0
    array[0][2][0] = 0
    array[0][2][1] = 0
    array[0][2][2] = 0
    array[0][2][3] = 0
    array[1][0][0] = 0
    array[1][0][1] = 0
    array[1][0][2] = 0
```

```
array[1][0][3] = 0
array[1][1][0] = 0
array[1][1][1] = 1
array[1][1][2] = 2
array[1][1][3] = 3
array[1][2][0] = 0
array[1][2][1] = 2
array[1][2][2] = 4
array[1][2][3] = 6
```

enum(枚举)

> 在 C 语言中，枚举（Enumerations）是一种数据类型，用于定义一组命名常量。枚举通过列举出所有可能的取值来限制变量的取值范围。

在 C 语言中，定义枚举的语法格式为：

```
enum 枚举名 {
    标识符1 = 常量1,
    标识符2 = 常量2,
    ...
    标识符n = 常量n
};
```

其中，枚举名是一种自定义的数据类型名称，标识符是枚举常量名，常量可以是整型或字符型，如果没有设置常量值，则默认从 0 开始自增。例如：

```
enum Color {
    RED,
    YELLOW,
    GREEN
};
```

这里定义了一个名为 `Color` 的枚举类型，它有三个取值：`RED`、`YELLOW`、`GREEN`。如果不设置取值，则它们的默认值为 `0`、`1`、`2`。

枚举类型变量的定义语法格式为：

```
枚举名 变量名;
```

例如：

```
enum Color c = RED;
```

这里定义了一个名为 `c` 的枚举类型变量，它的值被赋为 `RED`，即 `c` 的值为 `0`。

枚举类型的作用在于：它可以用于提高代码的可读性和可维护性，同时也有助于限制变量的取值范围。

例如，下面的代码定义了一个表示季节的枚举类型，用它来定义和存储季节变量：

```
#include <stdio.h>

enum Season {
    SPRING,
    SUMMER,
    AUTUMN,
    WINTER
};

int main() {
    enum Season currentSeason = AUTUMN;

    if(currentSeason == SPRING) {
        printf("It's spring now.\n");
    } else if(currentSeason == SUMMER) {
        printf("It's summer now.\n");
    } else if(currentSeason == AUTUMN) {
```

```
            printf("It's autumn now.\n");
        } else {
            printf("It's winter now.\n");
        }

        return 0;
    }
```

输出为：

```
    It's autumn now.
```

在这个例子中，我们定义了一个名为 `Season` 的枚举类型，它有四个取值：`SPRING`、`SUMMER`、`AUTUMN`、`WINTER`，这些取值对应春季、夏季、秋季、冬季四个季节。在 `main()` 函数中，我们定义了一个名为 `currentSeason` 的 `Season` 类型变量，将其值设置为 `AUTUMN`，最后使用一系列的 `if-else` 语句来判断当前季节，输出相应的结果。

8.指针

指针一直被誉为C语言的灵魂，也被大家认为是C语言里面最难懂最莫名其妙的存在。

那么我们就用指针来做这断断续续写了四天的教程的最终章！

> C 语言中，指针（Pointer）是一种变量，它存储了一个变量的地址，而不是变量的值。指针使用通常会涉及到以下几个部分:指针的定义语法格式为：
>
> - 指针声明：指针变量的类型声明，用于定义指针变量。
> - 指针初始化：将指针变量初始化为一个有效的地址。
> - 指针使用：通过指针来访问内存中的值，或者进行指针运算。

指针的定义语法格式为：

```
    数据类型 *指针变量名;         // 比如说 :  int *a;
```

那个菊花一样的「 * 」就是用来告诉C，这个是个指针类型变量哦，int类型的指针变量～～

初始化指针的两种方法

如果我们已经定义了一个 `int a = “10”` ，那么我们该如何把它变成一个指针变量呢?

1.只需要这样：

```
int a = 10;
int *z = &a;
```

这个时候， `a = 10` 就已经存入「 *z 」变量里面，雨天配咖啡，而**「 *记得配上& 」**哦。

> *2.指针变量需要动态分配内存，使用malloc等函数申请。这种方式的指针变量需要另外调用free等函数释放内存。*

比如：

```
int *z;
z = (int *)malloc(sizeof(int));
```

指针的作用主要也有两个

> 1. 通过指针可以直接访问内存中存储的数据，可以实现函数间的数据共享、数据传递、动态数据存储等操作。
> 2. 指针使得数据结构和算法等复杂程序实现变得更容易。

比如说我们定义一个指针可以这样：

```
*z = 20;                        // 记得菊花「 * 」
```

在比如说我们取用一个指针（内存地址）可以这样：

```
printf("z 的地址：%p\n", &z);    // 记得 「 & 」
```

我们也可以对指针进行算术运算，比如加法、减法等。例如：

```
p = p + 1;                          //将指针变量 p 加 1，即指向下一个整型变量的地址。
```

指针的使用需要注意有效性哦，如果访问已经释放了的指针，就会引起程序崩溃之类的问题。

同时要记得，指针变量的名称不会给变量分配内存，所以在定义指针变量后，还要给它赋值。

上面这句话的意思是

> 定义指针变量时，编译器只会为指针本身分配一个内存地址，但是不会为指针所指向的变量分配内存空间。因此，需要在定义指针变量之后，为指针所指向的变量分配内存空间，并把指针变量的值设置为这个变量的地址，否则指针变量将指向一个不确定的内存地址，访问其所指向的变量，可能会导致程序崩溃或导致诡异的行为。

比如，对于下面的代码：

```
int *p;
int num = 10;
*p = &num;
```

这里定义了一个整型指针 `p`，但是没有为其所指向的变量分配内存空间，然后把 `p` 的值设置为变量 `num` 的地址。这个赋值语句是不正确的，应该写作：

```
int *p;
int num = 10;
p = &num;
```

这里首先定义了一个整型指针 `p`，然后将 `p` 的值设置为变量 `num` 的地址，这样变量 `p` 就指向了变量 `num`，可以通过指针 `p` 来访问变量 `num` 的值。

所以说，指针变量的本质是一个内存地址，它的作用是指向某个变量的位置，而不是为这个变量分配内存空间。

最后：

目前教程就到此结束，断断续续写了四天的教程暂时也画上了句号。希望你可以在这个教程里学习到C语言的一二，由于是个人赶制，所以可能会有不少错误纰漏，还有不严谨和做得不好的地方，敬请原谅。