C语言

一、前言

1.1 数字电路简介

用数字信号完成对数字量进行算术运算和逻辑运算的电路称为数字电路，或数字系统。由于它具有逻辑运算和逻辑处理功能，所以又称数字逻辑电路。现代的数字电路由半导体工艺制成的若干数字集成器件构造而成。逻辑门是数字逻辑电路的基本单元。最基本的逻辑关系是与、或、非，而逻辑门就有相应的是与门、或门和非门，可以用电阻、电容、二极管、三极管等分立原件构成。

数字电路中，用电压的高低来区分出两种信号，低电压表示0，高电压表示1，由于只能通过这种方式表示出两种类型的信号，所以计算机采用的是二进制。

1.2 程序执行方式

计算机做的所有事情都叫做计算，包括看电影、画图等等，计算的步骤叫做算法

程序的两种执行方式：

• 解释：借助一个程序，那个程序能试图理解你的程序，然后按照你的要求执行。下次再执行还需要通过解释器
• 编译：借助一个程序，就像一个翻译，把你的程序翻译成计算机真正能懂的语言——机器语言——写的程序，然后，这个机器语言写的程序就能直接执行了。

解释型语言有特殊的计算能力，比如运行时修改源代码等；编译型语言有确定的运算性能

语言本身没有编译和解释之分，对于任何一种语言来说，既可以解释执行也可以编译执行。

比如说c语言是编译型语言，一般需要先编译再执行，但也有人做了c语言的解释器，习惯上来说的c语言是编译型语言只是它常用的执行方式而已

1.3 C语言简介

C语言诞生于美国的贝尔实验室，由丹尼斯·里奇（Dennis Ritch）以肯·汤普逊（Ken Thompson）在开发UNIX操作系统时设计了C语言，它是在B语言的基础上进行设计的。

C语言是一种工业语言，广泛应用于：

• 操作系统
• 嵌入式系统
• 驱动程序
• 底层驱动
• 图形引擎、图像处理、声音效果

第一个c语言程序：

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

1.4 关于GCC

C语言与UNIX系统关系密切，它通常会将C编译器作为软件包的一部分。安装Linux时，通常也会安装C编译器。后来**GNU开发了新的C/C++编译器并被广泛使用，即GCC**。另外，微软有自己的MS CC。后来发展为visual studio中的VC6，VS 20XX。

GCC原名为GNU C语言编译器（GNU C Compiler），只能处理C语言。但其很快扩展，变得可处理C++，后来又扩展为能够支持更多编程语言，如Fortran、Pascal、Objective -C、Java、Ada、Go以及各类处理器架构上的汇编语言等，所以改名GNU编译器套件（GNU Compiler Collection）

GCC移植到windows运行：需要将GCC和库都移植过去

方式1：cgwin为代表，经过unix library提供的接口将源码变成可以在windows上运行的unix程序。

方式2：mingw为代表，tdm-gcc等，minimal GNU for windows，GCC配合win lib生成win.exe。

推荐安装tdm-gcc。下载地址：https://jmeubank.github.io/tdm-gcc/，会自动添加到PATH中。

在命令行中输入gcc --version查看版本;

编译：

gcc 文件名1.c 文件名2.c... -o 可执行文件名（无后缀）即可编译，弊端是需要输入多个文件名

mingw32-make命令可以帮助编译多个.c文件;

mingw32-make -f Makefile.mingw则是使用makefile脚本进行编译;

运行：

最终命令行输入./文件名（不需要后缀）即可运行

二、变量

2.1引例：简单的输入程序

在终端中进行输入，以行为单位进行，行的结束标志就是按下回车键。未按下时程序不会读到任何东西。

```
#include <stdio.h>
int main(){
    int price=0;
    printf("请输入金额：");
    //出现在scanf格式字符串中的东西是它一定要输入的东西！！！
    scanf("%d",&price);//这里要求scanf函数读入下一个整数，读到的结果赋值给price。一定要加
&！！！！
    int change = 100 - price;
    printf("找您%d元", change);
    return 0;//编译器会自动添加，可以不写
}
```

2.2变量

`int price=0;` 在上述程序中这一行即定义了变量。变量名price，类型为int，初始值为0

变量是保存数据的地方，当它保存了数据才能参与到后续计算中。

变量定义的一般形式：

<类型名称> <变量名称>;，比如`int price`;或者`int price,amount;`

变量名是一种**标识符**，标识符只能由**字母、数字和下划线组成，数字不能放在第一位，C语言的关键字不能作标识符**

当变量定义时发生赋值，就是变量的初始化。C语言**不强制初始化，但所有变量在被使用前应该要赋一次值。**因为不赋值就使用变量的话，变量所在内存位置上原本的值就会被默认使用。

组合定义变量时，可以给他们单独赋值(**不能把一个值同时赋给所有变量！！！**)：

`int price = 10,amount = 100;`

2.3变量类型

C是一种有类型的语言，**所有的变量在使用前必须定义或声明，所有变量必须有确定的数据类型。**数据类型表示了在变量中可以存放什么类型的数据，变量中只能存放指定类型的数据，程序运行过程中也不能改变变量的类型。

2.4常量

固定不变的数是常数，直接写在程序中，称之为字面量（literal，字面文字）。更好的方式是定义一个常量：

`const int AMOUNT = 100;`上面的引例则可以改为`int change = AMOUNT-price;`

常量便于理解数字的实际意义，一般将常量放在程序前面，很容易找到它。常量一般全部字母大写

`const`（常数，不变的）为修饰符，加在int前面，用来给变量加上一个`const`属性，代表**这个变量的值一旦初始化后就不能修改了**

2.5浮点数

1、数据的底层表示和单位：

我们的程序离不开数据，比如我们需要保存一个数字或是字母，这时候这些东西就是作为数据进行保存，不过不同的数据他们的类型可能不同，比如1就是一个整数，0.5就是一个小数，A就是一个字符，C语言提供了多种数据类型供我们使用，我们就可以很轻松的使用这些数据了。

不同的数据类型占据的空间也会不同，这里我们需要先提一个概念，就是字、字节是什么？

计算机底层实际上只有0和1能够表示，这时如果我们要存储一个数据，比如十进制的3，那么就需要使用2个二进制位来保存，二进制格式为11，占用两个位置，再比如我们要表示十进制的15，这时转换为二进制就是1111占用四个位置（4个bit位）来保存。

一般占用8个bit位表示一个字节（B），它们是计量单位。

8 bit = 1 B，1024 B = 1KB，1024 KB = 1 MB，1024 MB = 1GB，1024 GB = 1TB，1024TB = 1PB

在不同位数的系统下基本数据类型的大小可能会不同，因为现在主流已经是64位系统，所以统一默认64位系统

2、原码、反码和补码

原码

实际上所有的数字都是使用0和1这样的二进制数来进行表示的，但是这样仅仅只能保存正数，那么负数怎么办呢?

比如现在一共有4个bit位来保存我们的数据，为了表示正负，我们可以让第一个bit位专门来保存符号，这样，我们这4个bit位能够表示的数据范围就是：（**0为正数，1为负数**）

- 最小：1111 => - (22+21+2^0) => -7
- 最大：0111 => + (22+21+2^0) => +7 => 7

虽然原码表示简单，但是原码在做加减法的时候，很麻烦！以4bit位为例：

1+(-1) = 0001 + 1001 = 怎么让计算机去计算？（虽然我们知道该去怎么算，但是计算机不知道，计算机顶多知道1+1需要进位！）

我们得创造一种更好的表示方式！于是我们引入了反码！！！

反码

正数的反码是其本身
负数的反码是在其原码的基础上, 符号位不变，其余各个位取反，即0变1；1变0
经过上面的定义，我们再来进行加减法：

1+(-1) = 0001 + 1110 = 1111 （这里反码再化为原码为1000）=> -0 （直接相加，这样就简单多了！）

思考：1111代表-0，0000代表+0，在我们实数的范围内，0有正负之分吗?

0既不是正数也不是负数，那么显然这样的表示依然不够合理！！！

补码

根据上面的问题，我们引入了最终的解决方案，那就是补码，定义如下：

正数的补码就是其本身 （不变！）
负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1。(即在反码的基础上+1)
其实现在就已经能够想通了，-0其实已经被消除了！我们再来看上面的运算：

1+(-1) = 0001 + 1111 = (1)0000 => +0 （只有4位，多进的1只能丢掉，现在无论你怎么算，也不会有-0了！）

补码1000即表示原码形式的“-0”，可以用来表示最小值！！！

所以现在，4bit位能够表示的范围是：-8~+7（C语言使用的就是补码存储！！！）

对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了**将符号位也参与运算的方法。** 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以**机器可以只有加法而没有减法**, 这样计算机运算的设计就更简单了。

计算机在存储时并不是直接按照其二进制表示（原码）来存储的，而是**按照其补码形式来进行存储的。**

因为计算机在运算时默认将原来的数表示成补码再参与运算，然后得到结果。注意此时的结果也是补码形式。

简而言之，当计算机**在输出时（在屏幕上打印时）会将其转换成原码，再转换成对应的十进制数进行输出。**

3、浮点数

```
//英尺英寸的转换
#include <stdio.h>
int main(){
    printf("请分别输入身高的英尺和英寸，"
        "如输入\"5 7\"表示5英尺7英寸：");

    int foot;
    int inch;
    scanf("%d %d", &foot, &inch); //内部改为"%lf %lf"
//方式1：可以将foot,inch变量改为double类型，同时scanf中改为 %lf
//方式2：将下面的inch/12改为inch/12.0即可
    printf("身高是%f米。\n", ((foot + inch / 12) * 0.3048));
}
```

上述程序在输入5 7或5 8等数字时发现一直是1.524米，因为**对两个整数做运算时它的结果也只能是整数！！上例中的7/12会直接丢弃小数部分，即是0。**

比如`10/3*3`的结果为9，但是`10.0/3*3`结果就是10了

浮点数是指带小数点的数值。浮点这个词的本意就是指小数点是浮动的，是计算机内部表达非整数（包含分数和无理数）的一种方式。

2.6表达式

一个表达式是一系列运算符和算子的组合，用来计算一个值

• 运算符（operator）是指进行运算的动作，比如加法运算符“+”，减法运算符“-”。
• 算子（operand，操作数）是指参与运算的值，这个值可能是常数，也可能是变量，还可能是一个方法的返回值

```
//计算时间差
#include <stdio.h>
int main() {
    int hour1, minute1;//先输入的时间
    int hour2, minute2;//后输入的时间
    scanf("%d %d", &hour1, &minute1);
    scanf("%d %d",&hour2,&minute2);
    int t1=hour1*60+minute1;
    int t2=hour2*60+minute2;
    int t=t2-t1;
    printf("相差%d时%d分",t/60,t%60);
}
```

运算符的优先级初步

优先级	运算符	运算	结合关系	举例
1	+	单目不变	自右向左	a*+b
1	–	单目取负	自右向左	a*-b
2	*	乘	自左向右	a*b
2	/	除	自左向右	a/b
2	%	取余	自左向右	a%b
3	+	加	自左向右	a+b
3	-	减	自左向右	a-b
4	=	赋值	自右向左	a=b

单目运算符：只有一个操作数

运算符的结合关系一般是自左向右；单目和赋值运算符是自右向左

赋值也是运算，也有结果；a = 6的结果是a被赋予的值6，a=b=6 ---->a=(b=6)

C语言没有幂运算

关于程序调试

设置断点的这一行代表还没执行，需要点击下一步才会执行

复合赋值符

5个算术运算符`+ - * / %`可以和赋值运算符“=”结合起来，形成一些复合赋值运算符：`“+=”、“-=”、“*=”、“/=”和“%=”`

比如说total += 5即代表total = total + 5；注意连个运算符中间不能有空格

total * = sum+12即代表total = total*(sum+12); 要先将右边的值算完

自增和自减运算符

“++”和“--”是两个很特殊的运算符，它们是单目运算符，这个算子还必须是变量。这两个运算符分别叫做自增和自减运算符，他们的**作用就是给这个变量+1或者-1。**

当它们位于变量前面叫做前缀形式，**++a的值是a加1以后的值。**

当它们位于变量后面则叫做后缀形式，**a++的值是a加1以前的值，即还是等于a。**

```
#include <stdio.h>
int main() {
    int a = 10;
    printf("a++=%d\n",a++);
    printf("a=%d\n",a);

    printf("++a=%d\n",++a);
    printf("a=%d\n",a);
}
输出结果：
a++=10
a=11
++a=12
a=12
```

对于一个四位数或三位数，比如1120,分割出后两位的办法：1120%100 即可！！！

例：怎么得到一个数字的十位数?

36/10 即可　360%100得到60，再60/10

三、判断和分支

3.1判断的一般形式

```
//还是计算时间差，如果想使用小时2-小时1，分钟2-分钟1的方式：
#include <stdio.h>
int main() {
    int hour1, minute1;//先输入的时间
    int hour2, minute2;//后输入的时间
    scanf("%d %d", &hour1, &minute1);
    scanf("%d %d",&hour2,&minute2);

    int ih = hour2 - hour1;
    int im = minute2 - minute1;
    if(im < 0){    //判断分钟数够不够减
        im = im + 60;
        ih --;    //分钟向小时借位
    }
    printf("相差%d时%d分",ih,im);
}

判断的一般形式： if只要求括号中的值为0或非0
if (条件成立) {
...
}

当if语句或else语句中只有一条代码时，可以不需要大括号。if语句结尾这行不能加分号！！！因为它结束的地方在下面一行。
if(total>amount)   //无分号！！！
    total +=amount+10;
else               //无分号！！！
    total +=10;
```

关系运算符	意义
==	相等
!=	不相等
>	大于
>=	大于或等于
<	小于
<=	小于或等于

当两个值的关系符合关系运算符的预期时，关系运算的结果为整数1，否则为整数0

```
printf("%d\n",5==3);  //结果为0
printf("%d\n",5>3);   //结果为1
```

3.2 关系运算符的优先级

所有的关系运算符的优先级**比算术运算符的低，但是比赋值运算的高**

在关系运算符内部当中，判断是否相等的==和!=的优先级比其他的低，而**连续的关系运算是从左到右进行的**

```
5 > 3 == 6 > 4
6 > 5 > 4
a == b == 6
a == b > 0
```

3.3注释(comment)

以两个斜杠`//`开头,用来向读者提供解释信息，对于程序功能无影响

延续多行的注释，要用多行注释的格式来写。多行注释由一对字符序列`/*`开始，而以`*/`结束。

3.4 else

```
if(条件判断){
...
}
else{ //这里是条件不成立则执行
...
}

//比较两个数的大小
printf("请输入两个整数进行比较：");
scanf("%d %d",&a,&b);
int max=0;
if(a>b){
    max=a;
}
else{
    max=b;
}
printf("max的值：%d", max);

//比较两个数的大小方法2：
int max = b;
if(a>b){
    max = a;
}
```

3.5 嵌套的if和级联的if-else if

当if的条件满足或者不满足的时候要执行的语句也可以是一条if或if-else语句，这就是嵌套的if语句

```
if(条件判断)    //这里内部嵌套可以不加大括号，也相当于一条语句
    if(条件判断2){
        语句1;
```

```
        语句2;
    }
    else{
        语句1;
        语句2;
    }
```

同一个括号内，else总是和最近的if匹配！！！　注意：缩进和对齐是不起作用的。

```
级联的if-else if-else
if(条件1){
    ...
}
else if(条件2){
    ...
}
else{
    ...
}
```

3.6多路分支switch-case

switch语句可以看作是一种**基于计算的跳转**，计算控制表达式的值后，**程序会跳转到相匹配的case（分支标号）处**，并执行case后面的语句，一直到break为止。

分支标号只是说明switch内部位置的路标，在执行完分支内部中的最后一条语句后，**如果后面没有break，就会顺序执行到下面的case里去**，直到遇到一个break，或者switch结束为止。

如果所有的case都不匹配，那么就执行default后面的语句；如果没有default，那么就什么都不做

```
switch(控制表达式){    //控制表达式只能是整数型的结果!!!
case 整数常量：    //整数常量可以是常数，也可以是常数计算的表达式，即表达式中所有数为常数，比如1+1
    ...;
    break;
case 整数常量2:
    ...;
    break;
...
default:
    ...
}

//月份的英文表示：
int main(){
    printf("请输入月份：");
    int month;
    scanf("%d", &month);
    switch ( month ){
        case 1: printf("January\n"); break;
        case 2: printf("February\n"); break;
        case 3: printf("March\n"); break;
        case 4: printf("April\n"); break;
        case 5: printf("May\n"); break;
        case 6: printf("June\n"); break;
        case 7: printf("July\n"); break;
```

```
        case 8: printf("August\n"); break;
        case 9: printf("September\n"); break;
        case 10:printf("October\n"); break;
        case 11:printf("November\n"); break;
        case 12:printf("December\n"); break;
    }
}
```

四、循环（loop）

4.1 while循环

```
//计算正整数x的位数
int x;
int n=0;// n代表位数
scanf("%d",&x);
n++;        //可以判断0是几位数
x /= 10;  //n每次加1时去掉最后一位即个位的数字
while(x>0){
    n++;
    x /=10;
}
printf("%d\n",n);
```

如果我们把while翻译作“当”，那么一个while循环的意思就是：当条件满足时，不断地重复循环体内的语句。

循环执行之前判断是否继续循环，所以有可能循环一次也没有被执行；

条件成立是循环继续的条件。

验证：测试程序常使用边界数据，如有效范围两端的数据、特殊的倍数等

调试：也可以在程序适当的地方插入`printf`来输出变量的内容

4.2 do-while循环

在进入循环的时候不做检查，而是在执行完一轮循环体的代码之后，再来检查循环的条件是否满足，如果满足则继续下一轮循环，不满足则结束循环。

```
do{
    ...
}while(循环条件); //注意有分号！！！
```

do-while循环和while循环很像，区别是在循环体执行结束的时候才来判断条件。也就是说，无论如何，循环都会执行至少一遍，然后再来判断条件。与while循环相同的是，条件满足时执行循环，条件不满足时结束循环。

4.3 while循环应用

```
//计算log2X中X是2的几次方
int x;
int ret = 0;  //ret=-1时，下面的变为while(x>-1)
scanf("%d",&x);  //小套路：可以令int t=x;保存x的原始值，最终打印t就行了。
```

```c
while(x>1){
    x /=2;
    ret++;
}
printf("log2of %d是2的%d次方",x,ret); //这里的x会随着循环而改变，不是原来的值

//计数循环
int count = 100;    //小套路：若模拟运行很大次数的循环，可以先模拟较少的次数再进行推断
while(count>=0){    //比如count为3时，一共执行了4次循环，推断等于100时一共101次循环
    count--;
    printf("%d\n",count);
}                           //一共执行101次循环。
printf("发射！！\n");


//猜数游戏  让计算机来想一个数，然后让用户来猜，用户每输入入一个数，就告诉它是大了还是小了，直到用户猜中为 //止，最后还要告诉用户它猜了多少次。
1. 计算机随机想一个数，记在变量number里；
2. 一个负责计次数的变量count初始化为0；
3. 让用户输入一个数字a；
4. count递增（加一）；
5. 判断a和number的大小关系，如果a大，就输出“大”；如果a小就输出“小”；
6. 如果a和number是不相等的（无论大还是小），程序转回到第3步；
7. 否则，程序输出“猜中”和次数，然后结束。
随机数：每次调用rand()就得到随机的整数
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    srand(time(0));          //这里会得到一个很大的随机数，
    int num = rand()%100+1; //rand()%n会得到0到n-1，则%100会得到0-99，再加1得到1-100之间的数字
    int count = 0;
    int a = 0;
    printf("我已经想好了1-100之间的一个数字\n");
    do{                         //需要先让用户进入循环输入
        printf("请猜测这个数字：");
        scanf("%d",&a);
        count++;
        if(a>num){
            printf("你猜大了\n");
        }
        else if(a<num){
            printf("你猜小了\n");
        }
    }while(a!=num);
    printf("恭喜！你用了%d次猜中了数为%d\n",count,a);
}
```

算平均数：让用户输入一系列的正整数，最后输入-1表示输入结束，然后程序计算出这些数字的平均数，输出输入的数字的个数和平均数

```c
#include <stdio.h>
int main(){
    int num;
    int sum = 0;
    int count = 0;
```

```
    scanf("%d",&num);
    while(num != -1){ //-1代表输入结束了
        sum +=num;
        count++;
        scanf("%d",&num);
    }

    printf("整数和d的平均数为%f\n",1.0*sum/count);
}
```

整数的分解：
一个整数是由1至多位数字组成的，如何分解出整数的各个位上的数字，然后加以计算？

- 对一个整数做%10的操作，就得到它的个位数；
- 对一个整数做/10的操作，就去掉了它的个位数；
- 然后再对第2步的结果做%10，就得到原来数的十十位数了
- 依此类推。

正整数的逆序：

```
#include <stdio.h>
int main()
{
    int x;
    scanf("%d", &x);
    int digit;
    int ret = 0;

    while ( x> 0 ) {
        digit = x%10;
        printf("%d", digit);  //700输出007
        ret = ret*10 + digit;
        // printf("x=%d,digit=%d,ret=%d\n", x, digit, ret);这种700输出7
        x /= 10;
    }
    // printf("%d", ret);

    return 0;
}
```

4.4 for循环

for循环像一个计数循环：设定一个计数器，初始化它，然后在计数器到达某值之前，重复执行循环体，而每执行一轮循环，计数器值以一定步长进行调整，比如加1或者减1。

```
for(初始值;判断条件;计数值变化){
    ...
}   //先满足判断条件后进入循环体内执行语句，然后再执行计数值变化的语句，再去判断是否满足条件，如此重复下去

//求阶乘
int main(){
    int x,res=1;
    scanf("%d",&x);
    int t=x;
    for(x;x>1;x--){ //满足条件后先进循环体中执行语句，然后再执行x--。
        res *= x;
    }
    printf("%d! 结果为%d",t,res);
```

```
}

for(i=0;i<n;i++)  它的循环次数是n，循环结束后i的值为n
```

for循环括号中的每个表达式都可以省略，分号不能省！！！ `for(;;)`

tips for loops:

- 如果有固定次数，选择for
- 如果必须要执行一次，选择do-while
- 其他情况选择while

4.5 break和continue循环控制

素数引例

素数指只能被1和本身整除的数，不包括1，1既不是素数也不是合数。

```
//判断输入的数是不是素数
#include <stdio.h>
int main(){
    int x;
    printf("输入大于等于2的整数：");
    scanf("%d",&x);

    int isPrime=1;  //isPrime为1代表是素数
    int i;
    for(i=2;i<x;i++){
        if(x%i==0){
            isPrime=0;
            break; //直接跳出循环
        }
    }
    if(isPrime==1){ //可以直接用if(i==x)来判断，不需要用isPrime，这里说明循环没有被
break,但可读性差
        printf("%d是素数",x);
    }
    else{
        printf("%d不是素数",x);
    }
}
当x能被i整除时，其实不需要走完循环。可以添加break直接跳出
```

break: 直接跳出循环

continue: 跳过循环中这一轮剩下的语句直接进入下一轮

```
//输出1-100中的素数
int main(){
    int x;
    for(x=2;x<=100;x++){
        for(i=2;i<x;i++){
            if(x%i==0){
```

```
                isPrime=0;
                break; //直接跳出循环
            }
        }
        if(isPrime==1){
            printf("%d\n",x);
        }
    }
}

//输出从2开始的前50个素数
int main(){
    int x=2;
    int count=0;
    while(count<=50){
        for(i=2;i<x;i++){
            if(x%i==0){
                isPrime=0;
                break; //直接跳出循环
            }
        }
        if(isPrime==1){
            count++;
            printf("%d\t",x);
            if(count%5==0){
                printf("\n"); //每5个数进行换行
            }
        }
        x++;
    }
}
```

注意：break和continue只能对它们所在的那一层循环进行操作！！

```
//凑硬币：用1毛、2毛和5毛的硬币凑出10元以下的金额，只想要一种结果：
int main(){
    int x;//x代表10元以下的金额
    int one,two,five;//代表1毛、2毛和5毛的个数
    int exit=0; //核实退出循环
    scanf("%d",&x);
    for(one=1;one<x*10;one++){
        for(two=1;two<x*10/2;two++){
            for(five=1;five<x*10/5;five++){
                if(one+two*2+five*5==10){
                    printf("1毛：%d张、2毛：%d张、5毛：%d张得到%d元",one,two,five,x);
                    exit=1;
                    break;
                }
            }
            if(exit) break; //要判断一下是否真的找到了一组满足条件的值，可能最里面一层没找到，这时第二层还不能退出，需要继续改变第二层的值
        }
        if(exit) break;//三个break跳出所有循环
    }
}
//这种情况叫作接力break
```

goto语句用法

在上述凑硬币的例子中，可以不使用接力break的方式来退出循环。

可以使用goto语句直接从多重嵌套循环处跳到最外层的地方。

```
//程序直接从goto语句执行处跳转到下面的"标号:"处执行。
goto 标号;
...
...
标号:
    ...


int main(){
    int x;//x代表10元以下的金额
    int one,two,five;//代表1毛、2毛和5毛的个数
    scanf("%d",&x);
    for(one=1;one<x*10;one++){
        for(two=1;two<x*10/2;two++){
            for(five=1;five<x*10/5;five++){
                if(one+two*2+five*5==10){
                    printf("1毛：%d张、2毛：%d张、5毛：%d张得到%d元",one,two,five,x);
                    goto out;
                }
            }
        }
    }
    out:
        return 0;
}
```

注：goto语句可能会破坏程序的结构性，谨慎使用

4.6 循环例题

分数求和

```
//分数求和: 1+1/2+1/3...+1/n
int main(){
    int n;
    scanf("%d",&n);
    int i;
    double res=0.0;
    for(i=1;i<=n;i++){
        res += 1.0/i;
    }
    printf("sum=%f",res);
}
//若计算改为1-1/2+1/3-1/4+...+1/n
int main(){
    int n;
    scanf("%d",&n);
    int i;
    double res=0.0;
    double sign=1.0; //代表分子的正负
    for(i=1;i<=n;i++){
```

```
        res += sign/i;
        sign=-sign;
    }
    printf("sum=%f",res);
}
```

正序分解整数

```
//正序分解整数，并用空格隔开每一位
int main(){
    int x;
    scanf("%d",&x);
    int count=0;
    int t=x;
    do{
        x /=10; //计算x的位数，每去掉个位数时count就加1
        count++;
    }while(x>0);
    int mask=pow(10,count-1)//pow函数这里可以计算出10的count-1次方
    x=t;
    do{
        int d = x/mask; //d为每一位数字，这里从最高位开始取。
        printf("%d",d);
        if(mask>9){
            printf(" ");//最后一位数字之后不想要空格,最后一位时mask的值为1，不满足if条件。
        }
        x %=mask;
        mask /=10;
    }while(mask>0);
}

//在计算整数位数时候，其实可以将mask融入到循环中，并且mask只需要从x>9开始，从x>0开始mask会多一个0
int mask=1;
while(x>9){ //这里改成while是考虑到个位数的问题，个位数直接就是mask等于1
    x /=10;
    mask *=10;
}
```

求最大公约数

```
//枚举法
int a,b;
int min;
scanf("%d %d",&a,&b);
if(a>b){
    min=b;
}
else
    min=a;
int i,res=0;
for(i=1;i<min;i++){
    if(a%i==0){
        if(b%i==0){
            res=i;
        }
```

```
    }
}
printf("最大公约数为%d",res);

//辗转相除法
1. 如果b等于0，计算结束，a就是最大公约数；
2. 否则，计算a除以b的余数，让a等于b，而b等于那个余数；
3. 回到第一步。
int main(){
    int a,b;
    int t;
    scanf("%d %d",&a,&b);
    while(b!=0){
        t=a%b;
        a=b;
        b=t;
    }
    printf("最大公约数%d",a);
}

//输出整数集：给定不超过6的正整数A，考虑从A开始的连续4个数字，输出所有由它们组成的无重复3位数
int main(){
    int a;
    scanf("%d",&a);
    int i,j,k;
    int count=0;
    i=a;
    while(i<=a+3){
        j=a;
        while(j<=a+3){
            while(k<=a+3){
                if(i!=j){
                    if(i!=k){
                        if(j!=k){
                            count++;
                            printf("%d%d%d",i,j,k);
                            if(count == 6){
                                printf("\n");
                                count=0;
                            }
                            else{
                                printf(" ");
                            }
                        }
                    }
                }
                k++;
            }
            j++;
        }
        i++;
    }
}
```

水仙花数

水仙花数是指一个N位正整数（N>=3），它的每位上的数字的N次幂之和等于它本身，比如153=1^3+5^3+3^3

```
//计算所有n位水仙花数 3<=n<=7
int main(){
    int n;
    scanf("%d",&n);
    int first=1;
    int i=1;
    while(i<n){    //计算n位数的起始值
        first *= 10;
        i++;
    }
    i=first;
    while(i<first*10){  //遍历所有n位数
        int t=i;
        int sum=0;
        while(t>0){  //得到数字的每一位
            int d=t%10;
            t /=10;
            int p=1;
            int j=0;
            while(j<n){    //求幂
                p *=d;
                j++;
            }
            sum +=p;
        }
        if(sum==i){
            printf("%d\n",i);
        }
        i++;
    }
}
```

五、数据类型

C语言是有类型的语言，变量必须在使用前定义并且确定类型

C以后的语言向两个方向发展：
• C++/Java更强调类型，对类型的检查更严格
• JavaScript、Python、PHP不看重类型，甚至不需要事先定义

• 支持强类型的观点认为明确的类型有助于尽早发现程序中的简单错误
• 反对强类型的观点认为过于强调类型迫使程序员面对底层、实现而非事务逻辑

总的来说，早期语言强调类型，面向底层的语言强调类型。C语言需要类型，但是对类型的安全检查并不足够

C语言中的类型分类：

• 整数
  • char、short、int、long、long long
• 浮点数
  • float、double、long double

• 逻辑
  • bool
• 指针
• 自定义类型

类型之间的不同点：

• 类型名称：int、long、double
• 输入输出时的格式化：%d、%ld、%hd、%lf
• 所表达的数的范围：char < short < int < float <double
• 内存中所占据的大小：1个字节到16个字节
• 内存中的表现形式：二进制数（补码，比如整型）、编码（不是自然二进制数，不能直接做运算，比如浮点数）

sizeof运算符：

给出某个类型或变量在内存中所占据的字节数。

静态运算符，它的结果在编译时刻就决定了；不能在sizeof的括号中做运算，这些运算不会执行！！

`sizeof(int);  sizeof(i);`

`sizeof(4+2)` 结果为4(int)，`sizeof(1+1.0)` 结果为8(double)，根据表达式的结果类型在编译时就确定了它的结果，但是表达式不会执行。

5.1 整数类型

整数类型	占用大小	范围
char	1字节	-128~127
short	2字节	-32768~32767
int	取决于编译器和CPU，4字节	
long	取决于编译器和CPU，4字节（CPU是64位，但编译器数据模型为LLP64）	
long long	8字节	

字长代表寄存器的大小，CPU和内存之间的数据传送单位通常也是1字长。

int就是用来表达寄存器的大小，比如寄存器字长为32bit或64bit，int就是这个不同的值

整数的内部表达：计算机内部一切都是二进制

数的范围：

1个字节可以表达的数：2^8=256 ，去掉符号位之后为-2^7~2^7-1

• 00000000 —> 0
• 11111111 ~ 10000000 —> -1 ~ -128
• 00000001 ~ 01111111 —> 1 ~ 127

```
char c = 255;  //255的二进制表示为11111111，char只有8位，按照补码为-1
int i = 255;
printf("c=%d,i=%d",c,i);
//结果为c=-1,i=255
```

unsigned

在整型前加上unsigned使得它们成为无符号的整数

不以补码形式来看待一个数，看成是纯二进制数，没有负号了，即只有0和正整数

可以扩大正数的表示范围

```
unsigned char c = 255;
printf("c=%d",c)//输出结果为255
```

- 如果一个字面量常数想要表达自己是unsigned，**可以在后面加u或U**
  - 255U
- 用l或L表示long(long)
  - 255l或255L

unsigned的初衷并非扩展一个数能表达的范围，而是为了做纯二进制运算，主要是为了移位。

整数越界

整数是以**纯二进制方式**进行计算的，所以：

- 11111111 + 1 —> (1)00000000 —> 0　　(超出8位的部分会被丢掉！！！)
- 01111111 + 1 —> 10000000 —> -128　(**最大的正数127加1之后就变为最小的负数！！**)
- 10000000 - 1 —> 01111111 —> 127　　(-128减1之后变为最大的正数127)

```
char c = 127;
c = c+1;
printf("c=%d",c); //结果为-128，类似一个圆，127过去就是-128

char c = -128;
c = c-1;
printf("c=%d",c);//结果为127

unsigned char c = 255;//无符号类型
c = c+1;
printf("c=%d",c);  //结果为0，也是类似圆，255过去就是0

//求int类型最大值
int main(){
   int a=1;
   while(a>0){ //类似一个圆，超过边界就会变为负值
       a++;
   }
   printf("int类型的最大值为%d",a-1); //int类型的最大值为2147483647
}
```

整数的输入和输出

其实只有两种形式：int或long （long）

• %d： int　比int小的类型都以int类型输出
• %u： unsigned int
• %ld： long
• %lu： unsigned long

•%lld： long long

```
int main(){
    char c=-1;
    int i=-1;
    printf("c=%u  i=%u",c,i); //c=4294967295  i=4294967295
}
```

八进制(octal)和十六进制(hexadecimal)

• 一个以`0`开始的数字字面量是8进制
• 一个以`0x`开始的数字字面量是16进制

%o用于8进制（这里是**字母o**）

%x和%X用于16进制，%X是输出的十以上的字母大写，%x是输出十以上的字母小写。

16进制很适合表达二进制数据，因为**4位**二进制正好是一个16进制位，可以表达0~15所有的16进制数

2进制、十进制、16进制转换：

例：2进制的1111是4位，所以我们必须直接记住它每一位的权值，并且是从高位往低位记：8、4、2、1。记住8421，对于任意一个4位的二进制数，我们都可以很快算出它对应的10进制值。

二进制(binary)	十进制(decimal)	十六进制(hex)
1111	15	F
1110	14	E
1101	13	D
...	...	...
0001	1	1
0000	0	0

比如看到F，我们需知道它是15（慢慢熟悉A ~ F这六个数），然后15如何用8421凑呢？应该是8 + 4 + 2 + 1，所以四位全是1即代表二进制1111。

5.2 浮点类型

浮点类型	占用大小	范围（第一个±代表接近0的数字，nan代表不是有效数字）	有效数字
float	4字节	±(1.20x10^-38~3.4x10^38)，0，±inf，nan	7位
double	8字节	±(2.2x10^-308~1.79x10^308)，0，±inf，nan	15位

注：有效数字代表着有多少位有效数字。

浮点类型	scanf	printf
float	%f	%f，%e（科学计数法的形式输出）
double	%lf	%f，%e

科学计数法：-5.12e+16，可以是正或负，整个词不能有空格

输出精度：

在%和f之间加上.n可以指定输出小数点后几位，这样的输出是做4舍5入的

```
float a,b,c;
a=1.345f;
b=1.123f;
c=a+b;
if(c==2.468){
    printf("相等");
}
else
    printf("不相等 c=%.10f或者%f",c,c);//不相等 c=2.4679999352或者2.468000

float是7位有效数字，四舍五入之后是2.468，实际在内部不等于2.468

//两个浮点数不能直接f1==f2来比大小
fabs(f1-f2)<1e-8 （float比它小就行）  fabs是求绝对值
```

带小数点的字面量默认是double类型！！！，因此float类型数字需加上f或F，比如2.0f

超出范围的浮点数：

```
printf("%f",12.0/0.0) //inf
printf("%f",-12.0/0.0)//-inf
printf("%f",0.0/0.0) //nan
```

浮点数是一种编码形式，在计算时是由专用的硬件部件实现的,计算double和float所用的部件是一样的

5.3 字符类型

char是一种整数，也是一种特殊的类型：字符。

用单引号表示的字符字面量：'a'， '1'

`printf` 和 `scanf` 里用 `%c` 来输入输出字符

如何输入'1'这个字符给char c呢?

`scanf("%c", &c);  ->1`

`scanf("%d", &i); c=i;  ->49`

'1'的ASCII编码是49，所以当c==49时，它代表'1'

```
printf("c='%c'或%d",c);   //结果为c='1'或49
```

5.4 转义字符\

用来表达无法印出来的控制字符或特殊字符，它由一个反斜杠“\”开头，后面跟上另一个字符，**这两个字符合起来，组成了一个字符。**

字符	意义	字符	意义
\b	退回一格但不删除（再输入会覆盖之前的数据）	\"	双引号
\t	制表位（它是在行当中的固定位置，不随数据变化）	\'	单引号
\n	换行	\\	反斜杠本身
\r	回车，光标退回最左边		

5.5类型转换

自动类型转换：

当运算符的两边出现不一致的类型时，**会自动转换成较大的类型**，这里大的意思是能表达的数的范围更大。

char —> short —> int —> long —> long long

int —> float —> double

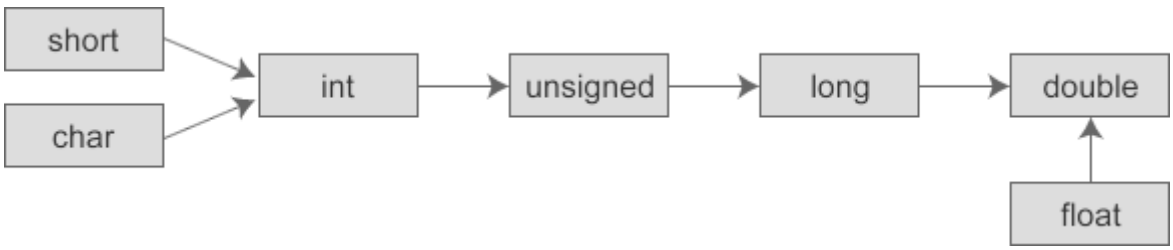
对于printf，任何小于int的类型会被转换成int；float会被转换成double。

但是scanf不会转换，比如要输入short，需要**%hd**

注意：在赋值表达式语句中，两侧类型不一致但都是算术类型的，计算的结果会被转换成被赋值变量的类型。这个过程可能导致类型升级或降级！！

在不同类型的混合运算中，编译器也会自动地转换数据类型，将参与运算的所有数据先转换为同一种类型，然后再进行计算。转换的规则如下：

- 转换按数据长度增加的方向进行，以保证数值不失真，或者精度不降低。例如，int 和 long 参与运算时，先把 int 类型的数据转成 long 类型后再进行运算。
- 所有的浮点运算都是以双精度进行的，**即使运算中只有 float 类型，也要先转换为 double 类型**，才能进行运算。
- **char 和 short 参与运算时，必须先转换成 int 类型。**



注意：图中unsigned代表unsigned int

强制类型转换：

要把一个量强制转换成另一个类型（通常是较小的类型），需要：`(类型)值`。比如：(int)10.0

这里只是从那个变量计算出了一个新的类型的值，它并不改变原来那个变量，无论是值还是类型都不改变

需要注意这时候的安全性，小的变量不能总表达大的量，比如(short)32768，结果是`-32768`

强制类型转换的优先级高于四则运算！！！ `(int)(a/b)`

5.6 逻辑类型

bool类型

`#include <stdbool.h>`之后就可以使用bool类型，它的值为true和false

```
#include <stdbool.h>

int main(){
    bool b=6>5;
    bool t= true;//bool实际上还是以int的手段实现的，所以可以当作int来计算
    printf("%d",b)//结果为1，无法输出true和false
    t=2;
}
```

逻辑运算

逻辑运算是对逻辑量进行的运算，结果只有0或1

逻辑量是关系运算或逻辑运算的结果

运算符	描述	示例
!	逻辑非	!a
&&	逻辑与	a&&b
||	逻辑或	a||b

例：如何判断一个字符c是否是大写字母?
• c >= ‘A’ && c<= ‘Z’

运算符优先级all

优先级	运算符	结合性
1	()	从左到右
2	!、+、 -、++、 --	(单目的+和-)从右到左
3	*、/、%	从左到右
4	+、-	从左到右
5	<、<=、>、>=	从左到右
6	==、!=	从左到右
7	&&	从左到右
8	||	从左到右
9	=、+=、-=、*=、/=、%=	从右到左

短路现象

逻辑运算是自左向右进行的，如果左边的结果已经能够决定结果了，就不会做右边的计算

对于&&，左边是false时就不做右边了；对于||，左边是true时就不做右边了。

不要把赋值，包括复合赋值组合进表达式！！！ 比如a == 6&&b += 1这种就不要写，可能右边不执行

条件运算符（三目）

条件? 条件满足的值 : 条件不满足的值；自右向左结合

```
count= count>20?count-10:count+10;
```

条件运算符的优先级高于赋值运算符，但是低于其他所有运算符

逗号运算--自左向右结合

逗号用来连接两个表达式，并以其**右边的表达式的值作为它的结果**

逗号的优先级是所有的运算符中最低的，所以它两边的表达式会先计算

左边的表达式会先计算，而右边的表达式的值就留下来作为逗号运算的结果。

逗号表达式主要用于for中

```
i=3+4,5+6; //i为7
i=(3+4,5+6); //i为11

for(i=0,j=10;i<j;i++,j--)  //放多个计算
```

六、函数

6.1 函数初步

代码在程序内部复制使用是质量不良的表现。

函数是一块代码，接收零个或多个参数，做一件事情，并返回零个或一个值

```
定义函数：
返回值类型 函数名（参数1，参数2...）{   //函数头
    函数体...
}
调用函数：函数名(参数值)； //没有参数也需要括号;如果有参数，则需要给出正确的数量和顺序
()起到了表示函数调用的重要作用。这些值会被按照顺序依次用来初始化函数中的参数
```

函数返回：

函数知道每一次是哪里调用它，会返回到正确的地方。

return停止函数的执行，并可以返回一个值。

• return;
• return 表达式;

一个函数里可以出现多个return语句

•返回值可以再赋值给变量
• 可以再传递给函数
• 甚至可以丢弃,有的时候要的是它的其他副作用

```
c=max(12,10);
c=max(max(2,3),10);
```

无返回值的函数：

void 函数名(参数表)，不能使用带值的return

6.2函数参数

C编译器会自上而下分析代码，需要把自定义的函数放到main函数之前！！！，即预先知到函数的基本信息，这样才能检查你对函数的调用是否正确。

函数原型

为了可以首先看到main函数，将自定义函数放到下面去，可以将函数头以分号结尾放到前面来，这就构成了函数原型。

函数原型的目的是告诉编译器这个函数长什么样，即它的基本信息。

```
int max(int a,int b); //函数原型的声明。 int max(int,int );这样也可以，只需要知道参数的类型

int main(){
    int a=1,b=2,c;
    c = max(a,b);
    printf("%d",c);
}

int max(int a,int b){
```

```
    if(a>b){
        return a;
    }
    else{
        return b;
    }
}
```

注：函数原型里可以不写参数的名字或者可以不同名，但是一般仍然写上

参数传递

调用函数时给的值与参数的类型不匹配是C语言传统上最大的漏洞

编译器总是**悄悄替你把类型转换好，但是这很可能不是你所期望的**

后续的语言，C++/Java在这方面很严格。

C语言在调用函数时，永远只能传值给函数

每个函数有自己的变量空间，参数也位于这个独立的空间中，和其他函数没有关系。

过去，对于函数参数表中的参数，叫做“形式参数”，调用函数时给的值，叫做“实际参数”。由于容易让初学者误会实际参数就是实际在函数中进行计算的参数，会误认为调用函数的时候把变量而不是值传进去了，所以不建议继续用这种古老的方式来称呼它们。

它们是参数和值的关系。

本地变量（局部变量或自动变量）

函数的每次运行，就产生了一个独立的变量空间，在这个空间中的变量，是函数在**这次运行**所独有的，称
作本地变量

定义在函数内部的变量就是本地变量，参数表中的参数也是本地变量

变量的生存期和作用域：

• 生存期：什么时候这个变量开始出现了，直到什么时候它消亡了
• 作用域：在（代码的）什么范围内可以访问这个变量（这个变量可以起作用）
• 对于本地变量，这两个问题的答案是统一的：**大括号内——也叫作块**

本地变量**是定义在块内的**
• 它可以是定义在函数的块内
• 也可以定义在语句的块内
• 甚至可以随便拉一对大括号来定义变量

程序运行进入这个块之前，其中的变量不存在，离开这个块，其中的变量就消失了

在块外面定义的变量在里面仍然有效

块里面定义了和外面同名的变量则会掩盖了外面的

不能在一个块内定义同名的变量。

本地变量不会被默认初始化，而**参数在进入函数的时候被初始化了（即赋值）**

其他细节

函数原型书写：

当函数没有参数时，建议函数原型写成`void f(void);`明确代表不接受任何参数；

`void f();`在传统C中，它表示f函数的参数表未知，并不表示没有参数。编译器**会猜测需要int类型，而实际上函数如果不需要int，就会发生错误。**

关于逗号：

调用函数时的圆括号里的逗号是**标点符号**，不是运算符。如果`f((a,b))`再加一层括号，那么就是逗号运算符了。

C语言不允许嵌套定义函数！！

6.3 递归

化为问题规模更小的情况，注意递归出口

```
//递归化为循环：
有一头母牛，它每年年初生一头小母牛。每头小母牛从第四个年头开始，每年年初也生一头小母牛。请编程实现在第n年的时候，共有多少头母牛？
1--->1头
2--->2头
3--->3头
4--->4头
cows[i] = cows[i-1] + cows[i-3];//第五年开始，其实第四年就可以开始。

#include <stdio.h>
int main()
{
    int cows[55] = {0, 1, 2, 3, 4};//cows[0]忽略不管，从1开始。
    int n, i;
    for(i = 5; i < 55; i++)
        cows[i] = cows[i-1] + cows[i-3];
    scanf("%d",&n);
    while(n)
    {
        printf("%d\n", cows[n]);
        scanf("%d",&n);
    }
    return 0;
}
```

七、数组

```
//引例程序：
int x;
 double sum=0;
 int cnt=0;
 int number[100];  //定义数组
 scanf("%d",&x);
```

```
while(x!=-1){
    number[cnt]=x;    //对数组中的元素赋值
    sum +=x;
    cnt++;
    scanf("%d",&x);
}
if(cnt>0){
    printf("%f",sum/cnt);
    int i;
    for(i=0;i<cnt;i++){
        if(number[i]>sum/cnt){
            printf("%d\n",number[i]); //使用数组中的元素，遍历数组
        }
    }
}
```

安全隐患！！：没有判断cnt是否会超过定义的100

7.1 数组的定义和使用

定义数组：

`<类型> 变量名称[元素数量];` 比如`int grades[100];`

元素数量必须是整数，代表的是元素的个数。C99之前元素数量必须是编译时刻确定的字面量

数组是一种容器（放数据的地方），特点是：

- 其中所有的元素具有**相同的数据类型；**
- **一旦创建，不能改变大小**
- 数组中的元素在内存中是**连续依次排列的**

比如一个int类型的数组`int a[10]`

- 10个单元：a[0],a[1],...,a[9]
- 每个单元就是一个int类型的变量
- 可以出现在赋值的左边或右边：
- a[2] = a[1]+6;
- 在赋值左边的叫做左值

数组的每个单元就是数组类型的一个变量，使用数组时放在[ ]中的数字叫做下标或索引，下标从0开始计数。

有效的下标范围

编译器和运行环境都**不会检查数组下标是否越界，无论是对数组单元做读还是写。**

但是一旦程序运行，越界的数组访问可能造成问题，导致程序崩溃。

- segmentation fault! 分段错误

也可能运气好，没造成严重的后果。所以这是程序员的责任来保证程序只使用有效的下标值：

[0,数组的大小 - 1]

注：可以创建长度为0的数组，比如int a[0];但是没有什么意义。

```
//投票统计
//写一个程序，输入数量不确定的[0,9]范围内的整数，统计每一种数字出现的次数，输入-1表示结束
#include <stdio.h>
int main(){
    const int num = 10; //代表数组大小
```

```
    int x;
    int i;
    int count[num]; //count[0]就代表0的个数，以此类推。相当于计数器组成的数组

    for(i=0;i<num;i++){    //数组的初始化，也可以使用int count[num]={0};后面的值会默认全为0
        count[i]=0;
    }

    scanf("%d",&x);
    while(x!=-1){
        if(x>=0&&x<=9){
            count[x]++;
        }
        scanf("%d",&x);
    }
    for(i=0;i<num;i++){  //遍历输出数组
        printf("%d:出现了%d次",i,count[i]);
    }
}
```

7.2 数组的运算

数组的集成初始化

直接用大括号给出数组的所有元素的初始值

- 不给出数组的大小，编译器会计算有多少个数:

`int a[] = {1,2,3,4,};`

- 如果给出了数组的大小，但是后面的初始值数量不足，则其后的元素默认被初始化为0:

`int a[10]={1,2,}; //这里后面的值就会默认为0`

C99还可以定位，即在大括号内部给指定下标处赋值：使用[n]的方式

`int a[10]={ [3]=12,[4]=7,10, };  //紧跟着[4]=7后面的值默认赋给下标5，其余下标处默认赋值为0`

这种用法特别适合于初始数据稀疏的数组。

数组的大小

sizeof给出**整个数组或某一位**所占据的内容的大小，单位是字节。

`sizeof(a)`或`sizeof(a[0])`

sizeof(a[0])给出数组中单个元素的大小，于是它们相除就得到了数组的单元个数:

`sizeof(a)/sizeof(a[0])`

这样的代码，就算修改数组中初始的数据，也不需要修改遍历的代码。即可扩展性

数组的赋值

数组变量本身不能被赋值！！！

要把一个数组的所有元素交给另一个数组，必须采用遍历的方式。

```
int a[]={2,1,4,6,};
//int b[]=a; //这是错误写法！！！！！
for(int i=0;i<length;i++){
    b[i]=a[i];  //遍历赋值
}
```

遍历数组通常都是使用for循环，让循环变量i从**0到<数组的长度**，这样循环体内最大的i正好是数组最大的有效下标。

常见的错误是：

- 循环结束条件是**<=数组长度**；
- 离开循环后，继续用i的值来做数组元素的下标！

数组作为函数参数

数组作为函数参数时，往往必须再用另一个参数来传入数组的大小,因为：

- 不能在[ ]中给出数组的大小
- 不能再利用sizeof来计算数组的元素个数！

```
int search(int key,int a[],int length);
int main(){
    int a[]={2,4,7,10,11,100,};
    int x;
    printf("请输入一个数字：");
    scanf("%d",&x);
    int index = search(x,a,sizeof(a)/sizeof(a[0]));  //传入数组a
    if(index != -1){
        printf("%d在数组a中的位置为%d",x,index);
    }
    else{
        printf("数组a中不存在%d",x);
    }
}
/**
找出key在数组a中的位置
@param key 要寻找的数字
@param a 要寻找的数组
@param length 数组a的长度
@return 若找到了则返回其在数组中的位置，若未找到则返回-1
*/
int search(int key,int a[],int length){   //数组作为参数传入
    int ret = -1;
    for(int i=0;i<length;i++){
        if(key==a[i]){
            ret=i;
            break;
        }
    }
    return ret;
}
```

例：数组判断素数

```
//之前的程序判断素数需要循环n次，下面看几种时间复杂度低的方法

//1、去掉偶数后，从3到x-1，步长变为2
int isPrime(int x){
    int ret = 1;
    int i;
    if(x==1||x%2==0&&x!=2){
        ret=0;
    }
    for(i=3;i<x;i+=2){
        if(x%i==0){
            ret=0;
            break;
        }
    }
    return ret; //循环次数为n/2
}

//2、只需要遍历到sqrt(x)就行，即根号x,计算平方根
#include <math.h>
int isPrime(int x){
    int ret = 1;
    int i;
    if(x==1||x%2==0&&x!=2){
        ret=0;
    }
    for(i=3;i<sqrt(x);i+=2){
        if(x%i==0){
            ret=0;
            break;
        }
    }
    return ret; //循环次数为sqrt(x)/2
}

//3、判断是否能被已知的且小于x的素数整除，因为如果一个数是合数，就一定会被一个比它小的质数整除。
//将从2开始的前100个素数放入数组中
int main(){
    const int num = 100;
    int prime[num]={2};
    int count=1;//数组从第1位开始，第0位已经是2了
    int i=3; //从3开始判断是不是素数
    while(count<num){
        if(isPrime(i,prime,count)){ //判断i如果是素数则加入数组中
            prime[count++]=i;  //小套路：i加入数组中后count加1
        }
        i++;
    }
    for(i=0;i<num;i++){
        printf("%d",prime[i]);
        if((i+1)%5)
            printf("\t"); //间隔制表符
        else
            printf("\n");  //每5个数字之间换行
    }
```

```
}

//判断是否能被已知的且小于x的素数整除
int isPrime(int x,int primes[],int numberOfPrimes){
    int ret=1;
    for(int i=0;i<numberOfPrimes;i++){
        if(x%primes[i]==0){
            ret=0;
            break;
        }
    }
    return ret;
}

//4、构造素数表:n以内的素数
//令x为2
//将2x、3x、4x直至ax<n的数标记为非素数
//令x为下一个没有被标记为非素数的数，重复2；直到所有的数都已经尝试完毕
//伪代码步骤：
//1. 开辟prime[n]，初始化其所有元素为1，prime[x]为1表示x是素数，放弃前两位0和1
//2. 令x=2
//3. 如果x是素数，则对于(i=2;x*i<n;i++)令prime[i*x]=0
//4. 令x++，如果x<n，重复3，否则结束
int main(){
    const int maxNum = 25;
    int isPrime[maxNum];  //使用数组判断是不是素数,0代表不是素数，1代表是素数
    int i;
    int x;
    for(i=0;i<maxNum;i++){ //初始化，其实放弃了前两位0和1
        isPrime[i]=1;
    }

    for(x=2;x<maxNum;x++){
        if(isPrime[x]){   //依次将素数的倍数设为0
            for(i=2;i*x<maxNum;i++){
                isPrime[i*x]=0;
            }
        }
    }

    for(i=2;i<maxNum;i++){  //输出素数
        if(isPrime[i]){
            printf("%d\t",i);
        }
    }
}
```

7.3 二维数组

`例：int a[3][5];` 通常理解为a是一个3行5列的矩阵

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
```

二维数组的遍历：

```
for(i=0;i<3;i++){
    for(j=0;j<5;j++){
        a[i][j]=i*j;  //a[i][j]代表第i行第j列上的单元
    }
}

//如果写成a[i,j]则代表a[j]，内部是逗号表达式
```

二维数组的初始化：

- **列数是必须给出的**，行数可以由编译器来数
- 每行一个{}，逗号分隔
- 最后的逗号可以存在，有古老的传统
- 如果省略，表示补零
- 也可以用定位（C99 ONLY）

```
int [][5]={
    {0,1,2,3,4},  //内部的大括号其实可以省略
    {2,3,4,5,6},
          };
```

```
//小游戏：判断谁获胜了       3个棋子连成线就算赢，棋子双方为x和o
//读入一个3X3的矩阵，矩阵中的数字为1表示该位置上有一个x，为0表示为o
//• 程序判断这个矩阵中是否有获胜的一方，输出表示获胜一方的字符x或o，或输出无人获胜
const int size = 3;
int board = [size][size]; //棋盘
int i,j;
int numOfx;  //棋子x的个数
int numOfo;  //棋子o的个数
int res=-1; //-1：没人赢    0：o获胜      1：x获胜

//读入矩阵
for(i=0;i<size;i++){
    for(j=0;j<size;j++){
        scanf("%d",&board[i][j]);
    }
}

//检查每行
for(i=0;i<size&&res==-1;i++){
    numOfx=numOfo=0;  //起始都为0
    for(j=0;j<size;j++){
        if(board[i][j]==1){
            numOfx++;
        }
        else{
            numOfo++;
        }
    }
    if(numOfx==size)  //x棋子一行有三个则赢了
        res=1;
    else if(numOfo==size)
        res=0;
}

//检查每列
```

```
if(res==-1){
    for(j=0;j<size;j++){
        numOfx=numOfo=0;
        for(i=0;i<size;i++){
            if(board[i][j]==1){
                numOfx++;
            }
            else{
                numOfo++;
            }
        }
        if(numOfx==size)  //x棋子一列有三个则赢了
            res=1;
        else if(numOfo==size)
            res=0;
    }
}

//检查主对角线
if(res==-1){
numOfo=numOfx=o;
for(i=0;i<size;i++){
    if(board[i][i]==1){  //副对角线这里写成board[i][size-i-1]==1
        numOfx++;
    }
    else
        numOfo++;
}
if(numOfo==size)
    res=0;
else if(numOfx==size)
    res=1;
}
```

八、指针

8.1 取地址运算

运算符 &

`scanf("%d", &i);`里面的&:

• **获得变量的地址**，它的操作数必须是**变量**

•例：`int i; printf("0x%x",&i);  //以16进制的形式输出`

•使用&取出的地址大小是否与int相等取决于编译器和CPU。(比如32位编译器就可能相等)

```
int i = 3;
printf("%p",&i);  //这里%p就是以16进制的地址形式输出
printf("%#p",&i); //#就是前面带上进制说明格式，比如十六进制就是0x
```

```
//在本机上地址不和int类型大小相等
int i = 2;
int p = &i;
printf("%#x\n",p);   //0xa2bff658
printf("%#p\n",&i);   //0x57a2bff658
printf("%ld\n",sizeof(int));  //4
printf("%ld",sizeof(&i));     //8

int i = 2;
long long p = &i;    //这样才相同
printf("%#llx\n",p); //0xd393dff784
printf("%#p\n",&i);  //0xd393dff784
```

&不能对没有地址的东西取地址，比如&(i++)等，不可以。

```
//相邻变量取地址
int i;
int p;
printf("%#p\n",&i);  //0x9fe4fffc6c  先定义的变量地址更高（大），说明系统自顶向下分配内存
printf("%#p",&p);    //0x9fe4fffc68  相邻两个int变量的地址相差4，说明int占据4个字节

//数组取地址
int a[10];
printf("%#p",&a);   //0x14437ffa70
printf("%#p",a);    //0x14437ffa70
printf("%#p",&a[0]);//0x14437ffa70
printf("%#p",&a[1]);//0x14437ffa74
```

8.2 指针

指针变量

如果能够将取得的变量的地址传递给一个函数，能否通过这个地址在那个函数内访问这个变量？

我们需要一个参数能保存别的变量的地址，如何表达能够保存地址的变量？

指针变量就是保存地址的变量

```
int i;
int* p = &i;//p是指针，指向int类型，是这个指针指向的地址中存放的int类型数据。

//这两行的意思是一样的，p是一个指针，指向int，而q只是普通的int类型变量。
//没有int*这种类型！！！
int* p,q;
int *p,q;
```

指针变量的值是内存的地址

- 普通变量的值是实际的值
- 指针变量的值是具有实际值的变量的地址

作为参数的指针

```
//定义一个参数需要指针的函数
void f(int *p);
//在被调用的时候得到了某个变量的地址：
int i=0;
f(&i);
//在函数里面就可以通过这个指针变量访问外面的这个i

int main(){
    int i=6;
    printf("%#p",&i); //0xf4cdbffafc
    f(&i);
    g(i);
}
void f(int *p){
    printf("%#p\n",p);  //0xf4cdbffafc，与上面相同
    printf("*p=%d",*p); //*p=6
    *p=26;
}
void g(int k){
    printf("k=%d",k);//结果为k=26
}
```

访问地址上的变量--*

'*'是一个单目运算符，用来访问指针的值所表示的地址上的变量

可以做右值也可以做左值：

- int k = *p;
- *p = k+1;

左值叫做左值**是因为出现在赋值号左边的不是变量，而是值，是表达式计算的结果：**

- a[0] = 2;
- *p = 3;
- 是特殊的值，所以叫做左值

&和*是互相反作用的

- *&yptr -> * (&yptr) -> * (yptr的地址)-> 得到那个地址上的变量 -> yptr
- `&*yptr -> &(*yptr) -> &(yptr) ->` 得到yptr的地址

指针应用场景

1、指针带回值

函数返回多个值，某些值就只能通过指针返回。
传入的参数实际上是需要保存带回结果的变量

```
#include <stdio.h>
//交换
void swap(int *pa,int *pb);
int main(){
    int a=5;
    int b=6;
    swap(&a,&b);
    printf("a=%d b=%d",a,b);
```

```
    return 0;
}
void swap(int *pa,int *pb){
    int t=*pa;
    *pa=*pb;
    *pb=t;
}

//找出数组最大和最小值
void minmax(int a[],int len,int *max,int *min);
int main(){
    int a[]={2,3,7,10,4,12,5};
    int min,max;
    minmax(a,sizeof(a)/sizeof(a[0]),&max,&min);
    printf("max=%d min=%d",max,min);
}
void minmax(int a[],int len,int *max,int *min){
    int i;
    *min=*max=a[0];
    for(i=1;i<len;i++){
        if(a[i]<*min){
            *min=a[i];
        }
        if(a[i]>*max){ //这里不能else if，不然会不执行的
            *max=a[i];
        }
    }
}
```

2、函数返回运算的状态(成功或失败），结果通过指针返回

常用的套路是让函数**返回特殊的不属于有效范围内的值**来表示出错：
• -1或0（在文件操作会看到大量的例子）

但是当任何数值都是有效的可能结果时，就得分开返回了，状态通过return返回，实际的值使用指针返回。

```
#include <stdio.h>
/**
    @return 若除法成功，则返回1；失败则返回0
*/
int divide(int a,int b,int *res);

int main(){
    int a=5;
    int b=2;
    int c;
    if(divide(a,b,&c)){
        printf("%d/%d=%d",a,b,c);
    }
    return 0;
}
int divide(int a,int b,int *res){
    int ret=1;
    if(b==0)
        ret=0;
    else
```

```
        *res=a/b;
    return ret;
}
```

后续的语言（C++,Java）采用了**异常机制**来解决这个问题

指针最常见错误：

定义了指针变量，还没有指向任何变量，就开始使用指针

这样如果是本地变量的话没有默认初始值，就会可能有内存上残留的一个值，从而不知道指向了哪里，碰巧这个地方不能写数据，程序就会崩溃。

```
//错误的例子：
int *p;
int k=12;
*p=12;   //错误！！！
```

8.3 指针与数组

```
//引例：找出数组最大和最小值
void minmax(int a[],int len,int *max,int *min); //第一个参数写成int *a
int main(){
    int a[]={2,3,7,10,4,12,5};
    int min,max;
    printf("main sizeof(a)=%d\n",sizeof(a));  //main sizeof(a)=28
    printf("main %#p",a);  //main 0x4969fff8f0
    minmax(a,sizeof(a)/sizeof(a[0]),&max,&min);
    printf("max=%d min=%d",max,min);
}
void minmax(int a[],int len,int *max,int *min){  //这里第一个参数可以写成int *a！！
    int i;
    printf("minmax sizeof(a)=%d\n",sizeof(a));  //minmax sizeof(a)=8
    printf("minmax %#p",a);  //minmax 0x4969fff8f0  地址和main函数中定义的是一样
的！！！
    *min=*max=a[0];
    for(i=1;i<len;i++){
        if(a[i]<*min){
            *min=a[i];
        }
        if(a[i]>*max){
            *max=a[i];
        }
    }
}
```

传入函数的数组成了什么？

- 函数**参数表中的数组实际上是指针**
- sizeof(a) == sizeof(int*)
- 可以用数组的运算符[ ]进行运算

以下四种函数原型是等价的：
- `int sum(int *ar, int n);`
- `int sum(int *, int);`
- `int sum(int ar[], int n);`
- `int sum(int [], int);`

数组变量是特殊的指针

数组变量本身表达地址，所以：
- `int a[10]; int *p = a;` // 无需用&取地址

但是**数组的一个单元表达的是变量，需要用&取地址**
- `int a[10]; a == &a[0]` //数组a的地址就等于a[0]的地址，即**首地址**

[ ]运算符可以对数组做，也可以对指针做：

- p[0] <==> *p <==>a[0] //相当于长度为1的数组

'*'运算符可以对指针做，也可以对数组做：

- *a == a[0];

数组变量是 `const` 的指针，所以不能被赋值！！！

- int a[ ] <==> int * const a=.... //代表a是常量，不能改变

指针和数组的区别：

1、它们作为本地变量时，指针需要初始化，不然它会指向一个不确定的地址，导致程序崩溃。数组就算不初始化，它也不会导致程序崩溃，但会使它的元素有不确定的值

2、指针的值可以指向别处的地址。而数组变量的值是指向自己的地址，指向首地址位置。

```
int *p;   //需要初始化
int a[10];


int k=10;
int *p=&k;
int a[10];
printf("p的值=%#p\n",p);      //p的值=0xfb78fff6fc
printf("p自己的地址=%#p\n",&p); //p自己的地址=0xfb78fff6f0     p自己的地址和值中保存地址是不同的

printf("a的值=%#p\n",a);       //a的值=0xfb78fff6c0
printf("a自己的地址=%#p",&a);  //a自己的地址=0xfb78fff6c0     数组变量a的值中保存的是自己的地址
```

8.4 指针与const

指针可以是const，指向地址的值也可以是const。

指针是const：

表示一旦得到了某个变量的地址，**指针不能被修改，不能再指向其他变量。**

- `int *const q = &i;` //q 是 const
- *q = 26; // 可以
- q++; // ERROR错误！！！

指针指向地址的值是const:

表示**不能通过这个指针去修改那个变量**（变量本身仍然可以修改，指针也能修改）

- `const int *p = &i;`
- *p = 26; // ERROR错误!！！(*p) 是 const
- i = 26; //OK
- p = &j; //OK

```
这两种是一个意思：不能通过指针修改指向地址的值，const在*前面
const int* p1 = &i;
int const* p2 = &i;

这是指针不能被修改，const在*后面
int *const p3 = &i;
判断哪个被const了的标志是const在*的前面还是后面。
```

总是可以把一个非const的值转换成const的

当要传递的参数的类型比地址还大的时候，这是常用的手段：

既能用比较少的字节数传递值给参数，又能避免函数对外面的变量进行修改。

```
void f(const int* x);//函数
int a = 15;
f(&a);  // 可以传入非const，在函数内部保证不会去动指针所指的值！！！

const int b = a;
f(&b); // 可以传入
b = a + 1; // Error!
```

const数组

- const int a[ ] = {1,2,3,4,5,6,};

- 数组变量本身就是const的指针，这里int前面的const表明数组的每个单元都是const int

- 所以必须通过**初始化方式进行赋值**

因为把数组传入函数时传递的是地址，所以在那个函数内部可以修改数组的值。

为了**保护数组不被函数修改值，造成破坏，可以设置参数为const**

- `int sum(const int a[], int length);`

8.5 指针运算

加减运算和关系运算

给一个指针变量加1表示要让指针指向下一个变量

'*'号是单目运算符，优先级很高，但没有++高

如果指针不是指向一片连续分配的内存空间，是离散的，则这种加1运算没有意义。

```
int a[10];
int *p = a;
*p相当于a[0]
*(p+1) <-> a[1]    //加1相当于地址上增加了一个sizeof(int)
*(p+n) <->a[n]
```

这些算术运算和关系运算可以对指针做：

- 给指针加、减一个整数 `(+, +=, -, -=)`
- 递增递减 `(++/--)` ++运算符的优先级比*高
- 两个指针相减：结果是**地址差/sizeof(数据类型)**

```
int a[2];
int *p1=a;
int *p2=&a[1];
printf("%#p\n",p1); //0x8aff5ff5e8
printf("%#p\n",p2); //0x8aff5ff5ec
printf("%d",p2-p1); //1    相减就是上述的地址差/sizeof(int)
```

*p++

```
*p++的意思：   ++优先级比*高
取出p所指的那个数据来，完事之后顺便把p移到下一个位置去
常用于数组类的连续空间操作
在某些CPU上，这可以直接被翻译成一条汇编指令

//使用*p++进行遍历
int main(){
    char c[]={2,4,6,8,-1}; //-1不是有效数据
    char *p=0;
    for(p=c;*p!=-1;){  //可以改为while(*p!=-1)
        printf("%d\n",*p++);
    }
}
```

`<, <=, ==, >, >=, !=` 都可以对指针做：

- 比较它们在内存中的地址
- 在数组中单元的地址肯定是线性递增的

0地址

内存中具有0地址，但是0地址通常是个不能随便碰的地址。

所以你的指针不应该具有0值，因此可以用0地址来表示**特殊的事情**：

- 返回的指针是无效的

- 指针没有被真正初始化（**先初始化为0**）
- NULL（必须大写）是一个预先定义的符号，表示0地址。有的编译器不愿意你用0来表示0地址。

指针的类型和类型转换

无论指向什么类型，所有的指针的大小都是一样的，因为都是地址

但是**指向不同类型的指针是不能直接互相赋值的**，这是避免错用指针。

指针类型转换：

void* 表示事先不知道指向什么类型的指针，先指向void，**GCC默认情况下，它计算时与char*相同。**

void 指针可以指向任意类型的数据，即可以用任意类型的指针对 void 指针进行赋值。

如果要将 void 指针 p 赋给其他类型的指针，则需要强制类型转换。

因为"无类型"可以包容"有类型"，而"有类型"则不能包容"无类型"！！！

```
int *a;
void *p;
p=a;
a=（int *）p;  //强制类型转换


这并没有改变p所指的变量的类型，当通过q去看i时就是char，可以让别人用不同的眼光通过p看它所指的变量。
我不再当你是int，看成是char类型!
int *p=&i;
char *q=(char*)p;
```

用指针来做什么

- 需要传入较大的数据时用作参数
- 传入数组后对数组做操作
- 函数返回不止一个结果
- 需要用函数来修改不止一个变量
- 动态申请的内存...

动态分配内存

`#include <stdlib.h>`

`int *a = (int*)malloc(n*sizeof(int));` 这里malloc返回的是void *，需要强制转换成int *

```
#include <stdio.h>
#include <stdlib.h>
//C99可以直接在[]中写入变量
int main(){
    int number;
    printf("请输入数量：");
    scanf("%d",&number);
    int a[number];
}
```

```c
//使用malloc函数动态分配空间
int main(){
    int number;
    int i;
    int *a=0;
    printf("请输入数量：");
    scanf("%d",&number);
    a=(int *)malloc(number*sizeof(int)); //malloc需要的是内存空间的大小，即字节数
    for(i=0;i<number;i++){
        scanf("%d",&a[i]);  //输入数据
    }

    for(i=number-1;i>=0;i--){
        printf("%d ",a[i]); //输出
    }

    free(a);    //释放内存空间
}
```

malloc用法：

#include <stdlib.h>

void* malloc(size_t size);

• 向malloc申请的空间的大小是以字节为单位的

• 返回的结果是void*，需要类型转换为自己需要的类型

• (int*) malloc(n*sizeof(int))

malloc如果申请内存空间失败则返回0，或者叫做NULL

```c
//能申请的内存空间是有限的
#include <stdio.h>
#include <stdlib.h>
int main(){
    void *p=0;
    int cnt=0;
    while(p=malloc(100*1024*1024)){  //1B*1024=1KB;1KB*1024=1MB;1MB*100=100MB
        cnt++;
    }
    printf("分配了%d00MB的空间",cnt);  //分配了19400MB的空间
}
```

free( )

把申请得来的空间还给“系统”，申请过的空间，最终都应该要还

只能释放**申请来的空间的首地址。**不能free系统自己分配的内存。

free(NULL)或free(0)是可以的，因为0地址申请不到，free(NULL)默认不做事情。

```c
int *p=0; //习惯性的初始化工作
free(p);  //搭配使用
```

常见问题：

• 申请了没free—>长时间运行内存会逐渐下降

  • 新手：忘了
  • 老手：找不到合适的free的时机

• free过了再free

• 地址变过了，直接去free

malloc申请的空间是连续的，连续多次进行malloc会申请间隔固定的大小进行分配内存

malloc(0)也可以返回一个地址，返回值取决于特定的库实现（它可能是空指针，也可能不是空指针）。其行为是由实现定义的（implementation-defined）。如果拿到一个指向一小块内存的指针，这个指针指向的（分配给我们的）内存的大小是由机器决定的。

8.6函数指针

实际上指针除了指向一个变量之外，也可以指向一个函数，当然函数指针本身还是一个指针，所以依然是用变量表示，但是它代表的是一个函数的地址（编译时系统会为函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址）

我们来看看如何定义：

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(){
    //返回值类型 (*指针变量名称)(函数参数类型...)  //注意一定要把*和指针变量名称括起来，不然优先级不够
    int (*p)(int, int) = sum;
    printf("%p", p);
}
```

这样我们就拿到了函数的地址，既然拿到函数的地址，那么我们就可以通过函数的指针调用这个函数了：

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(){
    int (*p)(int, int) = sum;

    int result = (*p)(1, 2);   //就像我们正常使用函数那样，(*p)表示这个函数，后面依然是在小括号里面填上实参
    int result = p(1, 2);  //当然也可以直接写函数指针变量名称，效果一样（咋感觉就是给函数换了个名呢）
    printf("%d", result);
}
```

有了函数指针，我们就可以编写函数回调了（所谓回调就让别人去调用我们提供的函数，而不是我们主动来调别人的函数），比如现在我们定义了一个函数，不过这个函数需要参数通过一个处理的逻辑才能正常运行：

```
int sum(int (*p)(int, int), int a, int b){  //将函数指针作为参数传入
    //函数回调
    return p(a, b);     //让别人帮你实现
}

int sumImpl(int a, int b){   //这个函数实现了a + b
    return a + b;
}

int main(){
    int (*p)(int, int) = sumImpl;  //拿到实现那个函数的地址
    printf("%d", sum(p, 10, 20));
}
```

九、字符串

9.1 初识字符串

字符串是以0（整数0）结尾的一串字符

- `0`或`'\0'`是一样的，`'\0'`更常用，因为整数0是int类型，很大。但是它们和`'0'`不同

0标志字符串的结束，但它不是字符串的一部分

- 计算字符串长度的时候不包含这个0

字符串以字符数组的形式存在，可以用数组或指针的形式访问

- 更多的是以指针的形式

string.h 里有很多处理字符串的函数

```
//最后一位放\0就使得这个word是一个字符串，同时它也是一个字符数组
//可以使用字符串的运算方式进行运算
char word[] = {'H', 'e', 'l', 'l', 'o','!','\0'}; //最后一位'\0'其实就代表整数0，可以写成整数0
```

字符串变量的形式：

```
编译器会自动生成结尾的'\0'
• char *str = "Hello";
• char word[] = "Hello";
• char line[10] = "Hello";
```

字符串常量：

使用双引号括起来，比如"Hello"

"Hello" 会被编译器变成一个字符数组放在某处，这个数组的**长度是6，结尾还有表示结束的整数0**

- 两个**相邻**的字符串常量会被**自动连接起来**（中间无其他东西，或者仅有空格、缩进或换行符）

• 行末的\表示下一行还是这个字符串常量，续到下一行中

```
printf("你好啊，\
        世界很美妙");    //这里第二行开头的两个tab制表符也会被当成字符串的一部分

printf("你好啊，\
世界很美妙");        //这样顶格才可以
```

总结：

• C语言的字符串是以字符数组的形态存在的
    • 不能用运算符对字符串做运算
    • 通过数组的方式可以遍历字符串

• 唯一特殊的地方是字符串字面量可以用来初始化字符数组

• 标准库提供了一系列字符串函数

9.2 字符串变量

`char* s = "Hello, world!";`
• s 是一个指针，初始化为指向一个字符串常量，**字符串常量位于代码段，是只读的**，编译时刻已经有值了。
    • 由于这个常量所在的地方，所以实际上s是 const char* s ，但是由于历史的原因，编译器接受不带const的写法
    • 试图对s所指的字符串做写入会导致严重的后果！！！

• 如果需要修改字符串，应该使用数组：
`char s[] = "Hello, world!";` 因为上面的指针是指向了代码段不可读的字符串，而数组则是**将代码段的字符串拷贝到数组的这个本地变量中。**

```
#include <stdio.h>

int main(){
    char *s1="Hello World";
    char *s2="Hello World";
    char s3[]="Hello World";
    printf("s1=%#p",s1);    //s1=0x7ff788c1a000
    printf("s2=%#p",s2);    //s2=0x7ff788c1a000    它们保存的地址相同

    printf("%#p",s3);    //0x337c3ffcf4，数组s3会将字符串常量拷贝到新的本地变量的地址上
    s3[0]='K';
    printf("%c\n",s3[0]); //K   可以将s3[0]修改为K

    //修改拷贝后地址上的s3[0]不影响原来地址上的值
    printf("s1[0]=%c, s2[0]=%c",s1[0],s2[0]);  //s1[0]=H, s2[0]=H
}
```

指针还是数组形式?

• `char *str = “Hello”;`
• `char word[] = “Hello”;`
• 数组：这个字符串拷贝在本地变量中
    • 作为本地变量空间自动被回收

• 指针：这个字符串不知道在哪里
    • 处理参数
    • 动态分配空间

• 如果要构造一个字符串—>数组
• 如果要处理一个字符串—>指针

关于char*

字符串可以表达为char*的形式；但是char*不一定是字符串，它可以是一个指针指向了一个字节或一串连续的字节。

• 本意是指向字符的指针，可能指向的是字符数组（就像int*一样）
• 只有它所指的字符数组有结尾的整数0，即`'\0'`才能说它所指的是字符串

字符串的赋值

并没有产生新的字符串，只是让指针s指向了t所指的字符串，对s的任何操作就是对t做的。

```
char *t = “title”;
char *s;
s = t;
```

9.3 字符串的输入和输出

`scanf`读入一个单词（**到空格、tab或回车为止**）

`使用%s的scanf`是不安全的，因为不知道要读入的内容的长度

可以使用`%长度s`，这个长度代表最多读几个字符，应该比数组长度小1，超过它的部分则舍弃。如果有第二个scanf，会接着读入刚刚被舍弃的部分。

```
char string[8];
scanf(“%s”, string);  //%s代表字符串格式
printf(“%s”, string);

char string[8];
scanf("%7s",string); //最多读入7位
printf("%s",string);
```

常见错误

1、指针没有初始化

```
• char *string;
• scanf(“%s”, string);
```

以为char*是字符串类型，定义了一个字符串类型的变量string就可以直接使用了。
但是由于没有对string初始化为0，可能它指向的地方会出错。

2、空字符串

• `char buffer[100]=””;`
    • 这是一个空的字符串，但是也会使得buffer[0] == ‘\0’

- `char buffer[] = “”;`
  - 这个数组的长度只有1，因为只有buffer[0]是'\0'

9.4 字符串数组

- **char **a**
  - a是一个指针，它指向另一个指针，那个指针又指向一个字符（串）。这不是一个字符串数组。
- **char a[ ][ ]**
  - a是一个二维数组，但是第二个维度的大小不知道，所以不能编译，会报错！！！

以下两者作为字符串数组时的区别：

- **char a[ ][10]**
  - a是一个二维数组，a数组中每一个元素是一个char[10]，比如a[0]就相当于一个char[10]，会限定大小。**它相当于在a的地址中即自己的内部有一个个的char[10]。**
- **char *a[ ]**
  - a是一个一维数组，a数组中每一个元素是一个char*，比如a[0]相当于char *。**它的每个元素都是一个指针，指向某个地址处的字符串。**

字符串数组的应用---main函数参数

```
• int main(int argc, char const *argv[])
```

第二个参数是一个字符串数组，保存了在终端中编译时候的命令行参数。第0个参数是本程序的可执行文件名，数组中后面的参数为命令行中用户输入的参数，以空格作为分隔。

前面的int表示后面的数组中有几个字符串。`argc`为argument count

例：
`./text.c hello 200 is ok`
此时argc=5
argv[1]指向“hello”
argv[2]指向“200”
argv[3]指向“is”
argv[4]指向“ok”
argv[0]指向本程序的可执行文件名，完整名字

```
//查看argv数组的内容
int main(int argc,char const *argv[]){
    int i;
    for(i=0;i<argc;i++){
        printf("%d:%s\n",i,argv[i]);
    }
}
//0:D:\CLion\c_project\c_study\cmake-build-debug\c_study.exe     输出了可执行文件路径名
```

9.5 字符串函数

1、putchar和getchar

putchar

`• int putchar(int c);` 参数是int，**但只能接收一个字符**，而不是一次接收4个字符。
• 会向标准输出即终端中**输出一个字符**
• 函数返回写了几个字符，正常为1。EOF（-1）表示写失败（文件结束），end of file，这是个宏定义

getchar

`• int getchar(void);`
• **返回从标准输入中读入的一个字符**，int类型是字符的ASCII码的值。
• 返回类型是int而不是char，是为了可以返回EOF（-1），代表输入结束。

```
int main(void){
    int ch;
    while((ch=getchar())!=EOF){
        putchar(ch);
    }
    printf("EOF");
}
```

上述程序的解释：

键盘、显示器等外设和C语言程序中间还有一个shell（命令解析器command interpreter），键盘输入会经过shell处理之后再给C程序，反之亦然。

shell会将键盘输入的东西进行一个行编辑的工作， **在按下回车之前字符都不会送入程序中**，会停留在缓冲区，包括回车键。

程序执行到getchar()函数时，自动从输入缓冲区中去找字符，如果输入缓冲区中没有字符或者字符读完，那么继续就等待用户输入字符。

键盘得到EOF的方式：

• Windows—>Ctrl + Z
• Unix—>Ctrl + D

注：在Clion中需要进入调试模式（debug）中才可以输入EOF，使用的是ctrl + D。

这就代表shell得到了EOF并且shell将它放入了缓冲区，getchar读到之后循环会终止。

ctrl + c 代表强制终止程序。shell不会将它放入缓冲区，而是直接终止了程序。

只有一个getchar

如果程序只有一个getchar函数，从键盘输入字符'A'，'B'，'C'并按下回车后，getchar会从缓冲区中读取一个字符，即A，同时字符'A'也被缓冲区释放，而字符'B'，'C'仍然被留在了缓冲区。

而这样是很不安全的，有可能下次使用的时候，我们的缓冲区会读到一些垃圾。但是最终当程序结束的时候，缓冲区它会自动刷新。

2、标准库中的字符串函数

`#include <string.h>;`

需要加入以上的头文件

strlen

`size_t strlen(const char *s);` //这里参数中的const代表函数内部不会修改字符串内容。

它返回**s的字符串长度（不包括结尾的0）**。

```
#include <string.h>

int mylen(const char *s);
int main(){
    char line[]="hello";
    printf("strlen=%d\n",strlen(line)); //strlen=5
    printf("sizeof=%d",sizeof(line));   //sizeof=6，代表6个字节数

    printf("mylen=%d",mylen(line)); //mylen=5
}

//自己设计一个strlen函数
int mylen(const char *s){
    int count=0;
    int index=0;
    while(s[index]!=0){   //'\0'就等于整数0
        index++;
        count++;
    }
    return count;
}
```

strcmp

`int strcmp(const char *s1, const char *s2);` //在比较过程中不会修改字符串

从首字符开始逐个字符的进行比较，直到某个字符不相同或者其中一个字符串比较完毕才停止比较。字符的比较为ASCII码的比较

比较两个字符串，返回：

- 0: s1和s2的字符对应相等
- >0: s1>s2 //在不相等时会返回一个差值，不同的编译器可能返回不同值，有的是正数1，有的是差值
- <0: s1<s2

```
char s1[]="abc";
char s2[]="abc";
//if(s1==s2)   //如果直接用两个数组名字比较会永远不相等，两个地址不同
printf("%d\n",strcmp(s1,s2));


//自己设计一个strcmp函数
int mycmp(const char *s1, const char *s2){
    int index=0;
    while(1){
        if(s1[index]!=s2[index]){
```

```
            break;
        }
        else if(s1[index]=='\0'){
            break;
        }
        index++;
    }
    return s1[index]-s2[index];
}
//内部while循环的其他表示方式：
while(s1[index]==s2[index]&&s1[index]!='\0'){
    index++;
}
//方式2：
while(*s1==*s2&&*s1!=0){
    s1++;
    s2++;
}
return *s1-*s2;
```

strcpy

`char * strcpy(char *restrict dst, const char *restrict src);`
把src的字符串拷贝到dst中， **restrict关键字表明src和dst不能重叠（C99）**，重叠是指两个字符串中有一部分内存空间是重叠的，拷贝时可能只是向前挪动了一部分，不是完全复制到一片新的空间。

restrict不能重叠的好处是当计算机是多核时，会将拷贝工作分成多段，让一个核去完成一段工作，这时不重叠会提升性能和工作效率。

restrict是c99标准引入的，**它只可以用于限定和约束指针**，并表明指针是访问一个数据对象的唯一且初始的方式。它告诉编译器，所有修改该指针所指向内存中**内容**的操作都必须**通过该指针来修改,而不能通过其它途径(其它变量或指针)来修改。**它能帮助编译器进行更好的优化代码,生成更有效率的汇编代码。常用于高性能场景或者两个指针之间不能重叠时

strcpy函数返回dst字符串，在第一个参数的位置。这样可以将本函数的结果参与其他运算。

strcpy复制字符串常见做法：

```
malloc动态分配一片内存空间，然后将字符串复制到这片空间中
char *dst = (char*)malloc(strlen(src)+1);  //这里记得要加1，因为还有结尾的0
strcpy(dst, src);
```

```
//实现strcpy函数,不带restrict
char *mycpy(char *dst,const char *src){
    int index=0;
    while(src[index]!=0){  //可以写成while(src[index])
        dst[index]=src[index];
        index++;
    }
    dst[index]=0;
    return dst;
}

//while循环的第二种写法：
char *res=dst;
while(*dst++=*src++);
return res;
```

strcat

`char * strcat(char *restrict dst, const char *restrict src);`
把src拷贝到dst的后面，连接形成一个长的字符串，catenation是连接的意思

返回dst，当然dst必须具有足够的空间

> 其实相当于令dst[strlen(dst)]=src[0]，从这个位置开始拷贝。

```
//实现strcat
char* my_strcat(char* dest, const char* str)
{
    char* ret = dest;
    //找到目的字符串里的'\0'
    while (*dest != '\0')
    {
        dest++;
    }
    //追加
    while (*dest++ = *str++);
    return ret;
}
```

安全问题

strcpy和strcat都可能出现安全问题，如果目的地没有足够的空间呢?

1、strncpy

`char * strncpy(char *restrict dst, const char *restrict src, size_t n);`

把 **src** 所指向的字符串复制到 **dst**，最多复制 **n** 个字符。当 src 的长度小于 n 时，dst 的剩余部分将用空字符填充，直到达到n个字符为止。

注：**strncpy不会向dst追加`'\0'`，即不会添加终止符。因为可能复制的字符只是dst的前一部分，需要根据具体情况判断是否要自己手动添加。**

```
//手动实现strncpy,n不超过src的字符个数
char* My_strncpy(char* dest, const char* src, int n)
{
    assert(dest != NULL);//保证dest非空
    assert(src != NULL); //保证src非空
    char* ret = dest;    //将dest首地址储存在ret中，在之后dest++运算中，可以方便找到
    while (n)   //一次复制一个字符，要复制n次
    {
        *dest = *src;  //复制
        src++;    //源地址往后+1
        dest++;   //目标地址往后+1
        n--;      //跳出循环条件
    }
    return ret;  //返回目的数组的首地址
}
```

2、strncat

`char * strncat(char *restrict dst, const char *restrict src, size_t n);`

从字符串src的开头拷贝n 个字符到dst字符串尾部，直到 **n** 字符长度为止。若strncat会将dest字符串最后的'\0'覆盖掉，字符追加完成后，会再追加'\0'。

```
//手动实现strncat，这里n不超过src字符个数
char* My_strncat(char* dest, const char* src, int n)
{
    char* ret = dest;  //将dest首地址储存在ret中，在之后dest++运算中，可以方便找到
    while (*dest != '\0') //用指针往后一个个找，找到dest结尾的‘\0’
        dest++;
    while (n && (*dest++ = *src++))//把src里的字符一个个放入dest后
         n--;   //循环跳出条件
    *dest = '\0';       //字符追加完成后，再追加'\0'，这里会发生短路现象，所以要加上0
    return ret; //返回dest字符串起始地址
}
```

3、strncmp

`int strncmp(const char *s1, const char *s2, size_t n);`

strncmp不是为了安全，有时候可能只需要对比前n个字符的大小，不需要全部比较完。

字符串中寻找字符

`char * strchr(const char *str, int c);`

在参数 **str** 所指向的字符串中搜索第一次出现字符 **c**（一个无符号字符）的位置的地址指针。**c**以 int 形式传递，但是最终会转换回 char 形式。

该函数返回在 str 中第一次出现字符 c 的**地址指针**，如果未找到该字符则返回 NULL。

`char * strrchr(const char *s, int c);` //多了个r即从右边开始找，用法和上面的相同

在参数 **str** 所指向的字符串中搜索最后一次出现字符 **c**（一个无符号字符）的位置的地址指针。

寻找字符第2个位置的地址指针及其他套路

```
//寻找第二个'l'的位置
char s[]="hello";
char *p=strchr(s,'l');
p=strchr(p+1,'l');
printf("%s",p);  //lo

//从找到的位置复制到另一片内存空间上
char s[]="hello";
char *p=strchr(s,'l');
char *t=(char *)malloc(strlen(p)+1);
strcpy(t,p);
printf("%s\n",t);
free(t);

//将找到位置的前面部分复制到另一片内存空间
char s[]="hello";
char *p=strchr(s,'l');
char c=*p;
*p='\0';
```

```
char *t=(char *)malloc(strlen(s)+1);  //s变短了，因为提前具有了'\0'
strcpy(t,s);
printf("%s\n",t);
*p=c;  //改回原来的值
free(t);
```

字符串中查找字符串

- `char * strstr(const char *s1, const char *s2);`

在**s1**中查找第一次出现字符串 **s2** 的位置，不包含终止符 '\0'。

- `char * strcasestr(const char *s1, const char *s2);`　　//忽略大小写

十、结构类型

10.1 枚举

定义

常量符号化：用符号而不是具体的数字来表示程序中的数字，提升可读性

枚举是一种用户定义的数据类型，它用关键字 enum 以如下语法来声明：

enum 枚举类型名字 {名字0, ..., 名字n} ;　　**//注意分号，这是一条语句**

枚举类型名字通常并不真的使用，我们要用的是在大括号里的名字，因为它们就是常量符号，**它们的类型是int，值则依次从0到n。**如：

enum colors { red, yellow, green } ;

这里就创建了三个常量，red的值是0，yellow是1，而green是2。

当需要一些可以排列起来的常量值时，定义枚举的意义就是给了这些常量值名字。

- 枚举量可以作为值
- 枚举类型名字前面可以跟上enum作为一种用户自定义类型
- 实际上**是以整数来做内部计算和外部输入输出的**

```
enum color{red,yellow,green};
void f(enum color c);   //类型就是enum color
int main(void){
    enum color t=red;
    scanf("%d",&t);   //输入输出作为整数
    f(t);
}
void f(enum color c){
    printf("%d",c);
}
```

套路：自动计数的枚举

```
enum COLOR{RED,YELLOW,GREEN,NumCOLORS}; //正好最后一个NumCOLORS就是其中的数量
```

枚举量

声明枚举量的时候可以指定值

• enum COLOR { RED=1, YELLOW, GREEN = 5}; //跟在RED后面的YELLOW如果不指定的话默认就是2

枚举只是int，即使给枚举类型的变量赋不存在的整数值也没有任何warning或error。

虽然枚举类型可以当作类型使用，但是实际上很(bu)少(hao)用。

如果有意义上是排比的名字，用枚举比const int方便

枚举比宏（macro）好，因为枚举有int类型

10.2 结构

结构是一种复合的数据类型，内部可以有各种类型的成员。

结构类型声明和使用：

```
//声明方式1：
struct point{
    int x;
    int y;
};  //这是一条语句，注意分号！！
//使用：
struct point 变量1，变量2;


//声明方式2：无名结构，只是定义了两个变量
struct{
    int x;
    int y;
}变量名1，变量名2;


//声明方式3：
struct point{
    int x;
    int y;
}变量名1，变量名2;
```

和本地变量一样，在函数内部声明的结构类型只能在函数内部使用。所以通常在函数外部声明结构类型，这样就可以被多个函数所使用了。通常会把结构类型的声明放在外面

结构变量的初始化

```
struct date{
  int month;
  int day;
  int year;
};

int main(){
    struct date today={08,16,2022};  //初始化方式1
    struct date thismonth={.month=8,.year=2022};  //初始化方式2，没有给的值默认为0
    printf("today's date is %i-%i-%i",today.year,today.month,today.day);
    printf("this month is %i-%i-
%i",thismonth.year,thismonth.month,thismonth.day);
}
```

结构成员

结构和数组有点像，区别是结构的成员可以是不同类型，而数组的单元必须是相同类型。
数组用[ ]运算符和下标访问其成员，而结构用.运算符和名字访问其成员

- `today.day`
- `student.firstName`

结构运算

可以直接用**结构变量名.成员变量**的方式进行访问。（**.的优先级很高，比&和*高**）

点运算符的左边必须是一个结构变量

对于整个结构，**可以做赋值、取地址，也可以传递给函数参数**

```
struct point{
    int x;
    int y;
};  //这是一条语句，注意分号！！

//初始化之后一样可以做如下运算：      但是数组无法做这两种运算！！！
p1 = (struct point){5, 10};      // 相当于p1.x = 5; p1.y = 10;

p1 = p2;   // 相当于p1.x = p2.x; p1.y = p2.y;
```

结构指针

和数组不同，结构变量的名字并不是结构变量的地址，必须使用&运算符

`struct date *pDate = &today;`

10.3 结构与函数

整个结构可以作为**参数的值**传入函数，这时候是在函数内新建一个结构变量，并复制调用者的结构的值，这与数组完全不同！！！即结构是值传递的。

- 当然也可以返回一个结构

`int numberOfDays(struct date d);`

```
//没有直接的方式可以接收结构类型的输入
//由于结构类型是值传递的，可以采用创建临时变量的方式返回
struct point inputPoint(void);
```

```c
int main(void){
    struct point y={0,0};
    y=inputPoint();
}

struct point inputPoint(void){
    struct point temp;  //创建临时变量
    scanf("%d",&temp.x);    //.的优先级比&高
    scanf("%d",&temp.y);
    return temp;  //返回值
}
```

指向结构的指针

在传递结构给函数时，我们可以传递结构的指针，这是更加被推荐的。因为值传递一个结构类型是非常浪费时间和空间的。

可以用`->`表示指针所指的结构变量中的成员

```c
struct date{
    int month;
    int day;
    int year;
}myday;

struct date *p=&myday;
(*p).month=12;
p->month=12;  //和上一行是同一个意思，用->表示指针所指的结构变量中的成员
```

```c
struct point{
    int x;
    int y;
};

struct point *getStruct(struct point *p);
void output(struct point p);
void print(const struct point *p);

//传递指针完成输入结构
int main(void){
    struct point y={0,0};
    getStruct(&y);
    outPut(y);
    output(*getStruct(&y));
    print(getStruct(&y));
}

struct point *getStruct(struct point *p){
    scanf("%d",&p->x);
    scanf("%d",&p->y);
    printf("%d %d\n",p->x,p->y);
    return p;
}

void output(struct point p){
```

```
    printf("%d %d\n",p.x,p.y);
}

void print(const struct point *p){  //可以保护结构不被修改，当然其中的成员变量也不能修改！！！
    printf("%d %d",p->x,p->y);
}
```

结构数组

数组中的每个元素是一个date类型的结构变量。

```
struct date dates[100];
struct date dates[] = {
{4,5,2005},{2,4,2005}};
```

结构中的结构

```
struct point{
    int x;
    int y;
};

struct rectangle{
    struct point pt1;
    struct poitn pt2;
}r;
//嵌套表示：r.pt1.x


//如果有变量定义：
struct rectangle r, *rp;
rp = &r;
//那么下面面的四种形式是等价的：
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
//但是没有rp->pt1->x （因为pt1不是指针）！！！
```

人类的组合是无限的，结构，数组可以互相嵌套组合，形成各种效果。

结构体的sizeof:

结构体的**总大小为结构体最宽基本类型成员大小的整数倍。**

结构体大小的计算：
1、当为空结构体时，其大小为1

2、选取结构体中**类型字节数最大**的作为**对齐符。**注意：是最大的类型字节数，例如：int a[10],并不是以40作为对齐符

3、每次申请对齐符个字节大小的内存。当内存不够时才继续申请。

```
举例：
struct A
```

```
{
    int a[10];
    char b;
    double c;
}

//该结构体中最大的类型是double，所以对齐符为8
1、首先分配8个字节
2、int a[10]占40个字节
3、所以每两个数组元素申请8个字节
4、直到申请5次（40个字节）才将int a[10]全部分配完毕
5、再为b分配8个字节（此时已经48个字节了）
6、最后为c分配，此时为b申请的8个字节还剩7个字节，并不够用，还得再分配8个字节。
7、总共56个字节
```

10.4 联合

typedef

C语言提供了一个叫做 typedef 的功能**来声明一个已有的数据类型的新名字。**比如：

`typedef int Length;`

使得 Length 成为 int 类型的**别名。**
这样， Length 这个名字就可以代替int出现在变量定义和参数声明的地方了：
`Length a, b, len ;`
`Length numbers[10] ;`

声明新的类型的名字:
新的名字是某种类型的别名，可以简化一些复杂的名字
改善了程序的可读性

typedef将其语句中间所有的东西命名为最后一个单词！！！最后一个单词是别名

```
typedef int Length;  // Length就等价于int类型

typedef *char[10] Strings; // Strings 是10个字符串的数组的类型

typedef struct Adate{
    int month;
    int day;
    int year;
} Date;     //相当于结构类型叫做Date
Date d={8,16,2022};

//结构类型或者像这样用aNode就可以代替struct node
typedef struct node aNode;
```

联合

```
union AnElt {
    int i;
    char c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```

联合的语法和struct是一样的，**但是联合的每个成员变量占据相同的内存空间，它们占据的空间只有一份。**

`sizeof(union …) =sizeof(每个成员)的最大值`

联合的存储：

所有的成员共享同一个内存空间

同一时间只有一个成员是有效的

union的大小是其最大的成员

当改变一个联合成员的值时，实际上修改了该联合所有成员的值。联合内成员的类型不同，允许程序员采用不同的方式解释内存中的同一组字节值。

联合的初始化只需要对一个成员变量进行。默认是对第一个成员做初始化

```
union AnElt elt1={12}; //默认是对第一个i进行的初始化
union AnElt elt2={.c='a'};   //可以指定某一个变量进行初始化
```

联合的使用场景：

得到数据(整数、小数)内部的各个字节。

X86的CPU是小端机器，即代表低位在前存放。

```
typedef union{
    int i;
    char ch[sizeof(int)];
} CHI;

int main(void){
    CHI chi;
    int i;
    chi.i=1234;     //1234的16进制是0x04D2
    for(i=0;i<sizeof(int);i++){
        printf("%02hhX",chi.ch[i]);  //每个元素以2个16进制数字的方式显示，占据一个char类型
    }                                 //结果为D2040000，因为机器是低位在前，逆过来了
}
```

十一、程序结构

11.1 全局变量

定义在函数外面的变量是全局变量

- 全局变量具有全局的生存期和作用域
- 它们与任何函数都无关
- 在任何函数内部都可以使用它们

```
//使用全局变量：
//__func__代表当前函数的名字，字符串形式
int f(void);
int all=12;  //all为全局变量
int main(int argc,const char *argv[]){
    printf("in %s all=%d\n",__func__,all);  //in main all=12
    f();
    printf("again in %s all=%d\n",__func__,all);  //again in main all=14
}

int f(void){
    printf("in %s all=%d\n",__func__,all);  //in f all=12
    all+=2;
    printf("again in %s all=%d",__func__,all);  //again in f all=14
    return all;
}
```

全局变量初始化

没有做初始化的全局变量会默认得到0值，指针会得到NULL值

只能用编译时刻已知的值来初始化全局变量

它们的初始化发生在main函数之前

```
int all=12;
int g2=all;   //这是不行的！！！ERROR
int g3=f();   //错误！！！ERROR

const int all=12;
int g2=all;   //这是可以的
int main(){
    ...
}
```

注：**如果函数内部存在与全局变量同名的变量，则全局变量被隐藏**

11.2 静态本地变量

在本地变量定义时加上static修饰符就成为静态本地变量。当函数离开的时候，静态本地变量会继续存在并保持其值

静态本地变量的初始化只会**在第一次进入这个函数时做，以后进入函数时会保持上次离开时的值。若没有初始化则会自动初始化为0或NULL**

```
int f(void);
int main(int argc,const char *argv[]){
    f();
```

```
    f();
    f();
}

//定义一个静态本地变量
int f(void){
    static int all=1;  //只有第一次调用会初始化为1，后两次就变为3和5
    printf("in %s all=%d\n",__func__,all);  //in f all=1
    all+=2;
    printf("again in %s all=%d",__func__,all);  //again in f all=3
    return all;
}
```

静态本地变量**实际上是特殊的全局变量，它们位于相同的内存区域**

静态本地变量具有全局的生存期，函数内的局部作用域。只能在函数内部访问

static在这里的意思是局部作用域（本地可访问）

11.3 返回指针的函数

返回本地变量的地址是危险的，因为一旦离开了函数那么本地变量就不存在了，它会被用于分配给其他的变量。**返回全局变量或静态本地变量的地址是安全的。**

注：上述不存在是指**不再受控了**，但是可能还会保留一段时间。并不是一定就立刻变化。

返回在函数内malloc的内存是安全的，但是容易造成问题。因为被调用的函数（callee）分配的内存，调用方（caller ）是不清楚应该怎么释放（甚至要不要释放）的。

从逻辑上来说，如果资源是某一方分配的，也应该由它来释放，只有它知道状态是怎么管理的，分配释放有什么内部机制（甚至将来你想做什么优化，比如用上池，都得这样设计接口才好改）。

最好的做法是返回传入的指针。

tips:
- 不要使用全局变量来在函数间传递参数和结果
- 尽量避免使用全局变量
- 使用全局变量和静态本地变量的函数是线程不安全的

11.4 编译预处理和宏

以#开头的是编译预处理指令，它们不是C语言的成分，但是C语言程序离不开它们

#define用来定义一个宏

#define <名字> <值>

- 注意**没有结尾的分号，因为不是C的语句！！！**
- 名字必须是一个单词，值可以是各种东西
- 在C语言的编译器开始编译之前，编译预处理程序（cpp）会把程序中的名字换成值
- 它是**完全的文本替换**
- gcc 文件名.c —save-temps //可以保存编译过程中的临时文件

.c(源代码进行编译预处理得到.i) -->.i (编译得到.s) -->.s(汇编代码文件) -->.0(目标代码文件，链接、装入生成可执行文件) -->.exe(在linux一般为.out)

C语言在编译之前会进行一个编译预处理，.c文件会一步步生成许多临时文件。

```
#define PI 3.14159  //PI是宏的名字，3.14159是宏的值。在预处理时会把程序中所有的PI替换成3.14159
//const double PI=3.14159;
```

如果一个宏的值中有其他的宏的名字，也是会被替换的

如果一个宏的值超过一行，最后一行之前的行末需要加`\`，即续行符

宏的值后面出现的注释不会被当作宏的值的一部分

没有值的宏

`#define _DEBUG`
这类宏是用于**条件编译**的，后面有其他的编译预处理。比如检查它是否存在，如果存在则编译一部分代码；如果不存在，则编译另一部分代码。

会有指令来检查这个宏是否已经被定义过了。

预定义的宏

前后都有两条下划线。

```
• __LINE__   //源代码文件当前所在行的行号
• __FILE__   //源代码文件的文件名（这里是绝对路径名）
• __DATE__   //当前日期，一个以 “MMM DD YYYY” 格式表示的字符串常量。
• __TIME__   //当前时间，一个以 “HH:MM:SS” 格式表示的字符串常量。
• __STDC__   //当编译器以 ANSI 标准编译时，则定义为 1；判断该文件是不是标准 C 程序。
• __func__   //代表当前函数的名字，字符串形式
```

带参数的宏

`#define cube(x) ((x)*(x)*(x))` //这里参数没有类型

宏可以带参数。

一切都要括号：

- 整个值要括号
- 在值内部参数出现的每个地方都要括号

因为宏是纯文本替换，不加括号可能会计算优先级错误。**宏不要加分号！！！**

```
#define cube(x) ((x)*(x)*(x))

int main(void){
    printf("%d",cube(5));
}
```

可以带多个参数

`#define MIN(a,b) ((a)>(b)?(b):(a))`

当然也可以组合（嵌套）使用其他宏

带参数的宏特点：

- 在大型程序的代码中使用非常普遍
- 可以非常复杂，如“产生”函数，在#和##这两个运算符的帮助下
- 存在中西方文化差异，国内很少使用宏
- 部分宏会被inline函数替代

```
#和##这两个语法的功能都比较简单，且都是在预处理阶段做一些工作：
//#主要是将宏参数转化为字符串
//##主要是将两个标识符拼接成一个标识符

//#的简单使用
#define STR(str) #str

//##的简单使用
#define  CMB(a,b) a##b
```

11.5 大程序结构

多个源代码文件

main()里的代码太长了则适合分成几个函数，一个源代码文件太长了则适合分成几个文件。

两个独立的源代码文件不能编译形成可执行的程序，需要放到同一项目中。

一个.c文件是一个编译单元，编译器每次编译只处理一个编译单元。

头文件

当自定义函数文件与main函数文件分开时，仍然需要保留函数原型声明。否则编译器会默认猜测该函数需要int类型的参数，返回类型也是int，虽然可以链接，但是执行时会发生错误。

编译器在编译的时候只看当前的一个编译单元，它不会去看同一个项目中的其他编译单元来找出那个函数的原型。

需要在调用函数的地方给出函数的原型，以告诉编译器那个函数究竟长什么样。

把函数原型放到一个头文件（以.h结尾）中，在需要调用这个函数的源代码文件（.c文件）中#include这个头文件，就能让编译器在编译的时候知道函数的原型。

```
#include "文件名.h"   //.h文件中保存函数原型
```

在使用和定义这个函数的地方都应该#include这个头文件！！！定义的地方会检查对外宣称的函数原型和实际定义的地方是不是一致

一般的做法就是任何.c都应该有对应的同名的.h，把所有对外公开的函数原型和全局变量的声明都放进去。（**全局变量可以在多个.c文件之间共享！！！**）

头文件可以看作是两个源文件的桥梁

不对外公开的函数

在函数前面加上**static**就使得它成为只能在所在的编译单元中被使用的函数。

在全局变量前面加上static就使得它成为只能在所在的编译单元中被使用的全局变量。

#include编译预处理

#include是一个编译预处理指令，和宏一样，在编译之前就处理了。

- **它把那个文件的全部文本内容原封不动地插入到它所在的地方**
- 所以也不是一定要在.c文件的最前面#include

#include有两种形式来指出要插入的文件：

- `“ ”`要求编译器首先在当前目录（.c文件所在的目录）寻找这个文件，如果没有，到编译器指定的目录去寻
找
- `< >`让编译器只在指定的目录去找

- 编译器自己知道自己的标准库的头文件在哪里
- 环境变量和编译器命令行参数也可以指定寻找头文件的目录

#include的误区：

- #include不是用来引入库的
- stdio.h里只有printf的函数原型，printf的代码在另外的地方，某个.lib(Windows)或.a(Unix)中
- 现在的C语言编译器默认会引入所有的标准库
- #include <stdio.h>只是为了让编译器知道printf函数的原型，保证你调用时给出的参数值是正确的类型。

声明

在.h文件中加入全局变量的声明：可以跨文件使用

```
extern int all;
```

声明是不产生代码的东西！！！例如：函数原型，变量声明，结构声明，宏声明，枚举声明，类型声明等

只有声明可以被放在头文件中，否则会造成一个项目中多个编译单元里面有重名的实体。

了解一下：某些编译器允许几个编译单元中存在同名的函数，或者用weak修饰符来强调这种存在

同一个编译单元里，同名的结构不能被重复声明。如果你的头文件里有结构的声明，很可能这个头文件会在一个编译单元里被#include多次。所以需要**“标准头文件结构”**来避免重复声明

```
//标准头文件结构     MAX_H是头文件名
#ifndef _MAX_H_  //如果没有定义MAX_H就定义，如果已经定义过了就不再定义。
#define _MAX_H_
...
#endif
```

运用条件编译和宏，保证这个头文件在一个编译单元中只会被#include一次

`#pragma once`也能起到相同的作用，但是不是所有的编译器都支持

十二、文件

12.1 格式化输入输出

输入输出形式：
- printf
    - %[flags][width][.prec][hlL]type
- scanf
    - %[flag]type

printf

flag	含义
-	左对齐（默认是靠右对齐的）
+	输出中在前面放 + 号，如果数据是负数则没有+号，正负号放最前面
(space)	左边用空格填充
0	左边以0填充，如果%-04d则左边不填充0了，直接左对齐，也只有1位。

width或prec	含义
number	占据字符数（整体的）
*	下一个**参数**是总字符数，printf("%*d" , 3 , a); 这里3是字符数
.number	小数点后的位数，四舍五入
.*	下一个**参数**是小数点后的位数

类型修饰HIL	含义
hh	单个字节，char
h	short
l	long
ll	long long
L	long double

type	用于	type	用于
i 或 d	int	g	float
u	unsigned int	G	float
o	八进制	a 或 A	十六进制浮点
x	十六进制	c	char
X	字母大写的十六进制	s	字符串
f 或 F	float, 6	p	指针
e 或 E	指数	n	读入/写出的个数

```
int num;
printf("%dty%n",12345,&num);    //%n可以得到前面的数据个数并且返回一个指针中去
printf("%d",num);  //7
```

scanf

flag	含义	flag	含义
*	跳过一个输入	l	long,double
数字	最大字符数	ll	long long
hh	char	L	long double
h	short		

scanf：%[flag]type

type	用于	type	用于
d	int	s	字符串（单词）
i	整数，可能为十六进制或八进制	[...]	所允许的字符
u	unsigned int	p	指针
o	八进制		
x	十六进制		
a,e,f,g	float		
c	char		

```
//$GPRMC,004319.00,A,3016.98468,N,12006.39211,E,0.047,,130909,,,D*79

//读入上面的GPS数据
scanf("%*[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,]",
sTime,sAV,sLati,&sNW,sLong,&sEW,sSpeed,sAngle,sDate);
```

printf和scanf的返回值--int类型接收

- 输出的字符数
- 读入的项目数（几个数据）

在要求严格的程序中，应该判断每次调用scanf或printf的返回值，从而了解程序运行中是否存在问题

文件概述

我们学习C语言都是数据的处理，这些数据都是在内存中的。一旦程序结束程序结束退出，数据也将灰飞烟灭。

文件操作的实现将帮助我们把数据存储到文件中，既硬盘上的文件，比如我们所熟知的txt格式，或其他各种后缀的文件，避免程序结束后数据丢失，实现存储数据的功能，甚至充当“**数据库**”的功能。

在C语言中，除了我们认识的文件，还有系统设备都将视为文件来看待。对于文件的操作分为三个步骤：

- 第一步：**打开文件**，用**fopen函数**来实现，这一步作用主要是建立程序和文件的关系，获取文件在内存中的文件指针
- 第二步：**读写文件**，分为**fprintf**、**fscanf**或者**fwrite**、**fread**或者**fputs**、**getss**等多组函数来实现。每组函数都分别是写和读文件。就像我们熟知的printf和scanf这组输入输出文件一样，不过这里的读写是向文件的。
- 第三步：**关闭文件**，需要**fclose函数**实现。

打开文件

在C语言中，对文件操作之前，首先需要打开文件，使用的函数是**fopen函数**，它的作用是**打开文件**，获取该文件的文件指针，方便后续操作。函数原型为：

```
FILE *fopen(const char *filename, const char *mode);
```

该函数需要两个字符串类型的参数，第一个是**文件路径名**，既要操作的文件对象。第二个是**打开方式**，这里的打开方式只是，对文件以何种模式打开。

12.2 文件的输入输出

在命令行终端可以用>和<做重定向，>代表输出到其右边的文件中，<代表输入到左边的文件中

FILE

`FILE* fopen(const char * restrict path, const char *restrict mode);`

`int fclose(FILE *stream);`

`fscanf(FILE*, ...)`

`fprintf(FILE*, ...)`

```
//打开文件的标准代码
FILE* fp = fopen(“file”,“r”);
if ( fp ) {    //代表文件不是NULL
    fscanf(fp,...);
    fclose(fp);
} else {
    ...比如输出文件无法打开
}
```

```
FILE *fp = fopen("test.txt","r");
if(fp){
    int num;
    fscanf(fp,"%d",&num);
    printf("%d",num);
    fclose(fp);
}
else{
    printf("无法打开文件");
}
```

mode	含义
r	打开只读
r+	打开读写，从文件头开始
w	打开只写。如果不存在则新建，如果存在则清空
w+	打开读写。如果不存在则新建，如果存在则清空
a	打开追加。如果不存在则新建，如果存在则从文件尾开始
..x（加在上述符号后面的）	只新建，如果文件已存在则不能打开

12.3 二进制文件

其实所有的文件最终都是二进制的
文本文件无非是用最简单的方式可以读写的文件，linux可以使用如下命令：

- more、tail
- cat
- vi

而二进制文件是需要专门的程序来读写的文件，比如图片、声音、视频等需要专门的程序。

文本文件和二进制文件比较

文本的优势是方便人类读写，而且跨平台

文本的缺点是程序输入输出要经过格式化，开销大

二进制的缺点是人类读写困难，而且不跨平台

- int的大小不一致，大小端的问题...

二进制的优点是程序读写快

程序为什么要文件

- 配置
  - Unix用文本，Windows用注册表
- 数据
  - 稍微有点量的数据都放在数据库中
- 媒体
  - 这个只能是二进制的
- 现实是，程序通过第三方库来读写文件，很少直接读写二进制文件了

二进制文件的读写：

```
size_t fread(void *restrict ptr, size_t size, size_tnitems, FILE *restrict
stream);
size_t fwrite(const void *restrict ptr, size_t size,size_t nitems, FILE *restrict
stream);
```

- 注意这里的FILE指针是最后一个参数
- 返回的是成功读写的字节数

12.4 位运算

可以对每一位进行运算

位运算符	含义
&	按位的与
|	按位的或
~	按位取反
^	按位的异或
<<	左移
>>	右移

按位与 &

如果 (x)i == 1 并且(y)i == 1，那么 (x & y)i =1，否则的话 (x & y)i = 0

•按位与常用于两种情况：

•让某一位或某些位为0：`x & 0xFE`

FE代表1111 1110，就会使得一个数的最低位变为0

•取一个数中的一段：`x & 0xFF`

FF代表1111 1111，可以截取一段数字

按位取|

如果 (x)i == 1 或 (y)i == 1，那么 (x | y)i = 1，否则的话，(x | y)i == 0

•按位或常用于两种应用：

•使得一位或几个位为1：`x | 0x01`

•把两个数拼起来：`0x00FF | 0xFF00`

按位取反 ~

(~x)i = 1 - (x)i，**把1位变0，0位变1**

想得到全部位为1的数：~0

7的二进制是0111，x | 7使得低3位为1，而x & ~7，就使得低3位为0

```
#include <stdio.h>

int main(void){
    unsigned char c=0xAA; //无符号char，即没有符号位，二进制表示为1010 1010

    //16进制直接以补码形式输出
    printf("%hhX\n",c);   //AA  无符号数的8位，补码形式直接输出

    printf("%hhx\n",~c);  //55  按位取反，二进制代表0101 0101

    //在C语言中，对于任意整数值x，计算表达式-x和~x + 1（即算补码）得到的结果完全一样。
    //hh代表截取低8位，即0101 0110，正好是0x56
```

```
    printf("%hhx\n",-c);  //56
    printf("%hx\n",-c);   //ff56  这里h代表16位，即1111 1111 0101 0110，正好是ff56

}
```

关于类型转换时位的变化

1、char型与int型

将int类型赋值给char类型时，直接截取低8位存储在char型中。

将char类型赋值给int类型时，分两种情况：

- 无符号char类型（即unsigned char型），数据存储在int类型低8位，剩下24位补0
- 有符号char类型，数据存储在int型低8位。若char首位是0，则int型剩下24位补0；若char首位是1，则int型剩下24位补1。

2、int型与long类型（只考虑int型与long类型长度不一致情况）

将long类型赋值给int类型时，直接截断数据，将低位原封不动存储在int型中。

将int类型赋值给long类型时，分两种情况：

- 无符号int类型（即unsigned int型），数据存储在long类型低位，剩下位补0
- 有符号int类型，数据存储在long类型低位。若int首位是0，则剩下位补0；若int首位是1，则剩下位补1。

总结：大类型转换为小类型直接截断，小类型转换为大类型前面补0或1。

按位异或^

如果(x)i == (y)i ，那么(x ^ y)i = 0，否则的话，(x ^ y)i == 1

两个位相等，那么结果为0；不相等，结果为1。

如果x和y相等，那么x ^ y的结果为0

对一个变量用同一个值异或两次，等于什么也没做

•x ^ y ^ y —> x

左移 <<

i << j，i中所有的位向左移动j个位置，而右边的低位填入0

所有小于int的类型，移位以int的方式来做，结果是int。

•x <<= 1 十进制等价于 x *= 2

•x <<= n 十进制等价于 x *= 2的n次方.

右移 >>

i >> j，i中所有的位向右移j位

所有小于int的类型，移位以int的方式来做，结果是int

对于unsigned的类型，左边高位填入0

对于signed的类型，左边高位填入原来的最高位（保持符号不变）

•x >>= 1 十进制等价于 x /= 2

•x >>= n 十进制等价于 x /= 2的n次方.

输出一个数的二进制

```
int main(void){
    int num;
    scanf("%d",&num);
    //mask是用来逐一对比num每一位是什么，如果是1则输出1，如果是0则为0
    unsigned mask=1u<<31;  //unsigned默认为unsigned int，左移31位
    for(;mask;mask>>1){
        printf("%d",num & mask?1:0);
    }
    printf("\n");
}
```

位段

可以同时控制多个比特位。

把一个int的若干位组合成一个结构：

```
struct {
    unsigned int leading : 3; //代表这个成员占3个比特位
    unsigned int FLAG1 : 1;
    unsigned int FLAG2 : 1;
    int trailing : 27;
};
```

可以直接用位段的成员名称来访问多位

比移位、与、或还方便

编译器会安排其中的位的排列，不具有可移植性

当所需的位超过一个int时会采用多个int

十三、常用系统库介绍

数学库

#include <math.h>

```
#include <stdio.h>
#include <math.h>

int main() {
    int a = 2;
    double d = sqrt(a);    //使用sqrt可以求出非负数的算术平方根（底层采用牛顿逼近法计算）
    printf("%lf", d);
}

int main() {
    int a = 2;
```

```
    double d = pow(a, 3);    //使用pow可以快速计算乘方，这里求的是a的3次方
    printf("%lf", d);
}

int main() {
    printf("%f", tan(M_PI));    //这里我们使用正切函数计算tan180度的值，注意要填入的是弧度值
    //M_PI也是预先定义好的π的值，非常精确
}

int main() {
    double x = 3.14;
    printf("不小于x的最小整数：%f\n", ceil(x));
    printf("不大于x的最大整数：%f\n", floor(x));
}

int main() {
    double x = -3.14;
    printf("x的绝对值是：%f", fabs(x));    //   求绝对值
}
```

通用工具库stdlib

#include <stdlib.h>

```
int main() {
    int arr[] = {5, 2, 4, 0, 7, 3, 8, 1, 9, 6};
    //工具库已经为我们提供好了快速排序的实现函数，直接用就完事
    //参数有点多，第一个是待排序数组，第二个是待排序的数量（一开始就是数组长度），第三个是元素大小，第四个是排序规则（我们提供函数实现）
    //最后一个参数实际上就是函数回调，因为函数不知道你的比较规则是什么，是从小到大还是从大到小？所以我们需要编写一个函数来对两个待比较的元素进行大小判断。
    qsort(arr, 10, sizeof(int), compare);

    for (int i = 0; i < 10; ++i) {
        printf("%d ", arr[i]);
    }
}

int compare(const void * a, const void * b) {  //参数为两个待比较的元素，返回值负数表示a比b小，正数表示a比b大，0表示相等
    int * x = (int *) a, * y = (int *) b;    //这里因为判断的是int所以需要先强制类型转换为int *指针
    return *x - *y;    //其实直接返回a - b就完事了，因为如果a比b大的话算出来一定是正数，反之同理
}

//当然还有malloc和free

//搭配#include <time.h>使用随机数
#include <stdlib.h>
#include <time.h>
```

srand(time(0))是作为rand()的种子出现的，因为电脑取随机数是伪随机，只要种子一样，则取出的数一定一样。

这里用time(0)这个函数，则是返了当前的时间值。这个值是按照时间而变化的，所以，srand((time(NULL)))这个函数的作用，就是一个简单的设定随机数的种子方法。通过这个函数，可以得到每次都不一样的随机数。

```
srand(time(0));           //这里会得到一个很大的随机数
int num = rand()%100+1;   //rand()%n会得到0到n-1，则%100会得到0-99，再加1得到1-100之间的数字
```

time()函数的用途
返回一个值，即格林尼治时间1970年1月1日00:00:00到当前时刻的时长，时长单位是秒。
t1=time(NULL)或t1=time(0)，将空指针传递给time()函数，并将time()返回值赋给变量t1

十四、内存分区

内存四区分为栈区、堆区、数据区与代码区。

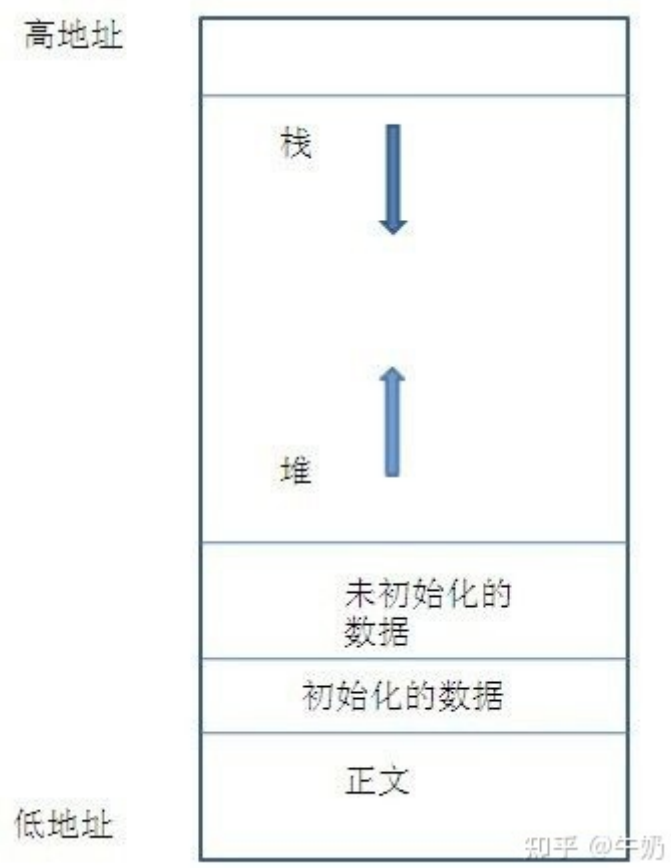
- 栈区指的是存储一些临时变量的区域，临时变量包括了**局部变量、局部常量、返回值、参数、返回地址等**，当这些变量超出了当前作用域时将会自动弹出。该栈的最大存储是有大小的，该值固定，超过该大小将会造成栈溢出。
- 堆区指的是一个比较大的内存空间，主要用于对动态内存的分配；在程序开发中一般是**开发人员进行分配与释放**，若在程序结束时都未释放，系统将会自动进行回收。
- 数据区指的是主要存放全局变量、**全局常量、字符串常量和静态变量**的区域，数据区又可以进行划分，分为全局区与常量区。全局变量与静态变量将会存放至该区域。
- 代码区就比较好理解了，主要是存储可执行代码，该区域的属性是只读的。

栈的顶部为地址的“最小”索引，随后往下依次增大，但是由于堆栈的特殊存储结构，我们将变量 a 先进行存储，那么它的一个索引地址将会是最大的，随后依次减少；第二次存储的值是 b，该值的地址索引比 a 小，由于 int 的数据大小为 4，所以在 a 地址为 2293324 的基础上往上减少 4 为 2293320，在存储 c 的时候为 char，大小为 1，则地址为 2293319。

栈容量具有固定大小，超过最大容量将会造成栈溢出。在平常开发中若需要大容量的内存，需要使用堆。**堆使用需要手动释放内存，否则容易造成内存泄露。**

内存泄漏是指在动态分配的内存中，并没有释放内存或者一些原因造成了内存无法释放，轻度则造成系统的内存资源浪费，严重的导致整个系统崩溃等情况的发生。

关于栈：



栈的实际形状是一个杯子。杯子的底部永远称栈底，杯子的顶部永远称栈顶。所以对于栈来说上面是栈底下面是栈顶，而对于堆来说，上面是堆顶下面是堆底。

栈和堆大概还有以下区别：

（1）栈是系统自动分配和释放，而堆需要程序员手动分配释放。所以总体上栈的分配效率要比堆高。

（2）栈中存放的内容包括函数返回地址、相关参数、局部变量（这个用最多）和寄存器内容等。当主函数调用另外一个函数的时候，要对当前函数执行断点进行保存。而堆中具体存放内容是由程序员来填充的。

（3）每个进程拥有的栈的大小要远远小于堆的大小。

十五、数据缓冲策略

printf打印数据时，一般会先把数据放入C缓冲区，然后再刷新到内核缓冲区，最后再写入硬件

这个过程中，**数据从C缓冲区迁移到内核缓冲区的操作我们称为缓冲（也可以理解为刷新）**

下面我们将介绍 缓冲策略的三种情况

1、不缓冲（不刷新），如write

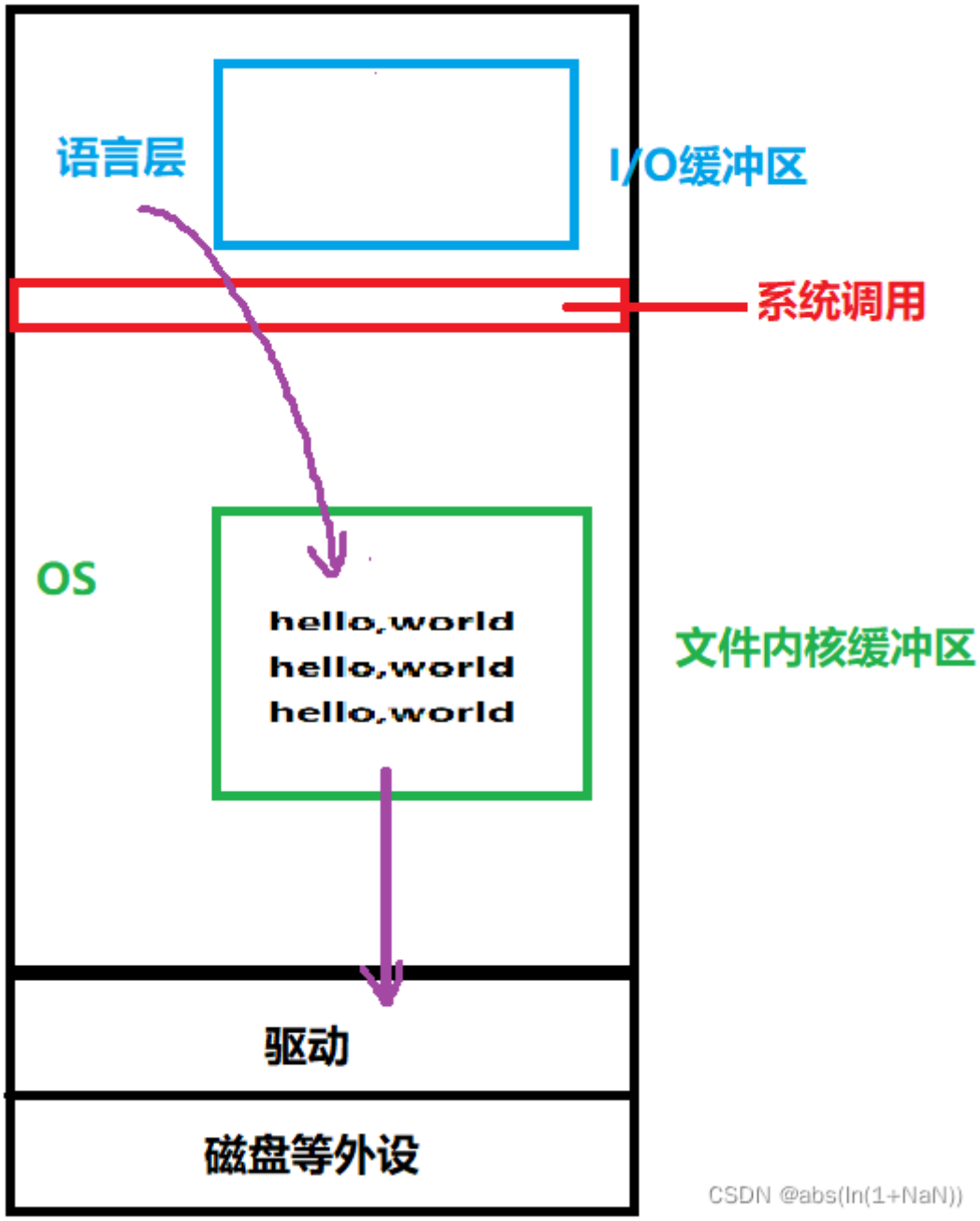
2、行缓冲（行刷新），遇到换行符\n才会刷新

3、全缓冲（缓冲区满了才刷新），如写入数据到磁盘

不缓冲（不刷新）

不缓冲的意思就是，**不会向C缓冲区存入内容，而是直接写入到内核缓冲区中**

比如系统调用函数 write函数，直接刷新文件内核缓冲区，然后再定时调用写入磁盘的驱动函数 write_disk将数据写入磁盘



行缓冲

行缓冲的意思是，遇到换行符的时候才会将数据从**C缓冲区**迁移到 **内核缓冲区**

验证方式也比较简单

```c
#include <unistd.h>

int main(){
    printf("hello，world");
    sleep(5); //延迟5秒
}
```

图中代码没有加 \n，为了让效果更明显，我们延迟5s退出进程

所以最后的**现象**是运行程序以后，过了5s，"hello,world"才会被打印，而且没有换行

printf会将数据存入 C缓冲区，由于没有换行符，不会刷新到内核缓冲区

只能等到进程结束以后，才会将C缓冲区的内容刷新到内核缓冲区，最终打印到显示器上

全缓冲

全缓冲的意思是，只有等缓冲区满了以后，才会将缓冲区的内容刷新到内核缓冲区

一般不会存在溢出，因为满了就刷新，然后从0开始存