##

c语言设计(翁恺慕课版本)

第一周：程序设计与c语言

算法

1.我们要让计算机做计算，就需要像这样找出计算的步骤，然后用编程语言写下来

2.计算机做的所有事情都叫做计算

程序的执行

1.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

2.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够直接执行了

解释语言vs编译语言

1.语言本来没有编译/解释之分

2.常用的执行方式不同而已

3.解释性语言有特殊的计算能力

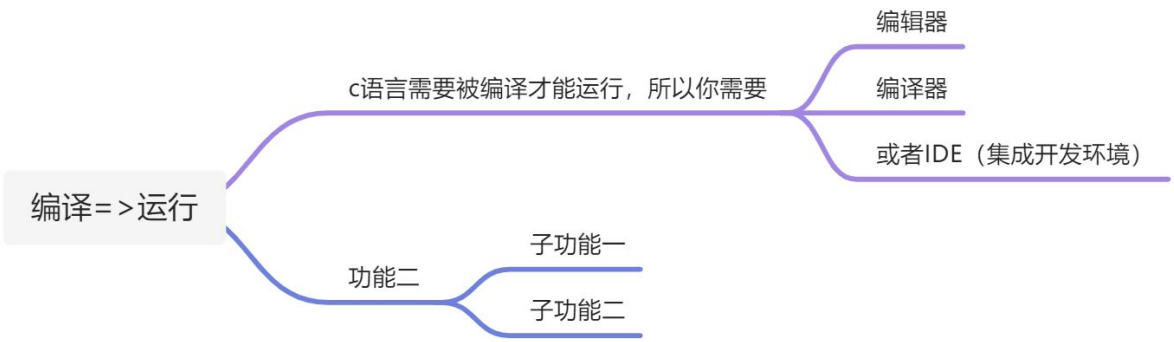
4.编译型语言有确定的运算性能

c语言用在哪里?

1.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

2.开发效率>>学习乐趣 开发效率>>开发乐趣 日常应用很少直接用c语言编写

3.学习c的过程主要是写练习代码



四则运算

四则运算	c符号	意义
+	+	加
-	-	减
×	*	乘
÷	/	除
	%	取余
（）	（）	括号

%表示取两个数相除以后的余数

第二周：计算

2.1变量

算找零

如何能在程序运行时输入那个数字，然后计算输出结果?

需要：1.有地方放输入的数字-printf(输出)，能够提示哪里输入数字

2.有办法输入数字-scanf(输入)，提供输入数字的通道

3.输入的数字能参与计算

如何输入

输入也在终端窗口中

输入就是以行为单位进行的，行的结束标志就是你按下回车键。在你按下回车之前，你的程序是不会读到任何东西的

变量

int price = 0;

这一行定义了一个变量。变量的名字是price，类型是int，初始值是0

变量是一个保存数据的地方，当我们需要在程序里保存数据时，比如上面的例子中要记录用户输入的价格，就需要一个变量来保存它。用一个变量保存了数据，它才能参加到后面的计算中，比如计算找零

变量定义

变量定义的一边形式：<类型名称><变量名称>

int price;

int amount;

int price,amount;

变量的名字

1.变量需要一个名字，变量的名字是一种"标识符"，意思是它是用来识别这个和那个的不同的名字

2.标识符有标识符的构造规则。基本的原则是：标识符只能由字母、数字和下划线组成，数字不可以出现在第一个位置上，c语言的关键字（有的地方叫他们保留字），不可以用作标识符

赋值和初始化

int price = 0;

这一行，定义了一个变量。变量的名字是price，类型是int，初始值是0

price = 0是一个式子，这里的"="是一个赋值运算符，表示将"="右边的值赋给左边的变量

赋值

和数学不同，a=b在数学中表示关系，即a和b的值是一样的；而在程序设计中，a=b表示要求计算机做一个动作：将b的值赋给a。关系是静态的，而动作是动态的。在数学中，a=b和b=a是等价的，而在程序设计中，两者的意思是完全相反的

初始化

当赋值发生在定义变量的时候，就像给变量price=0那样，就是变量的初始化。虽然c语言并没有强制要求所有的变量都在定义的地方做初始化，但是所有的变量在第一次被使用(出现在赋值运算符的右边)之前应该被赋值一次

问：如果没有初始化？答：变量会出现一个随机数，可能很大也可能很小不固定但一定不会是你想要的那个数

变量初始化

<类型名称><变量名称>=<初始值>;

int price = 0;

int amount = 100;

组合变量定义的时候，也可以在这个定义中单独给单个变量赋初值，如：int price = 0,amount = 0;

读整数

scanf("%d"，&price);

要求scanf这个函数读入下一个整数，读到的结果赋值给变量price

小心price前面的&

表达式

“=”是赋值运算符，有运算符的式子就叫做表达式

price = 0;

change = 100-price;

变量类型

int price = 0;

这一行，定义了一个变量。变量的名字是price，类型是int，初始值是0.

C是一种有类型的语言，所有的变量在使用之前必须定义或者声明，所有的变量都必须具有确定的数据类型。数据类型表示在变量中可以存放什么样的数据，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

c99可以在任何地方定义变量，ANSI C只能在代码开头的地方定义变量

常量

int change = 100 - price;

固定不变的数，是常数。直接写在程序里，我们称为直接量(literal)

更好的方式是定义一个常量

const int AMOUNT = 100;

const

const是一个修饰符，加在int的前面，用来给这个变量加上一个const(不变的)的属性。这个const的属性表示这个变量的值一旦初始化，就不能再修改了

int change = AMOUNT - price;

如果试图对常量做出修改，把他放在赋值运算符的左边，就会被编译器报错

tips

程序要求读入多个数字时，可以在一行输入，中间用空格分开，也可以在多行输入

在scanf的格式字符串中有几个%d，它就等待用户输入一个整数，当然，字符串后面也需要对应有那么多整数

两个整数运算的结果只能是整数

例如：10/3*3=9　10跟10.0在C中是完全不同的数

10.0是浮点数

浮点数

1.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(包括分数和无理数)的一种方式。另一种方式叫做定点数。人们借用浮点数这个词来表达所有带小数点的数。

2.当浮点数和整数放到一起运算时，C会将整数转换成浮点数，然后进行浮点数的运算

double

1.inch是定义为int类型的变量，如果把int换成double，我们就把它改成double类型的浮点数变量了。

2.double的意思是"双"，它本来是"双精度浮点数"的第一个单词，人们用来表示浮点数类型。除了double，还有float(意思就是浮点)表示单精度浮点数

在输入的时候数据类型定义为：%lf

在输出的时候数据类型定义为：%f

数据类型

整数：int　printf("%d"，...)　scanf("%d",...)

带小数点的数：　double　printf("%f")　scanf("%lf",....)

整数

整数类型不能表达有小数部分的数，整数和整数的运算结果还是整数。计算机里会有纯粹的整数这种奇怪的东西，是因为整数的运算比较快，而且占地方也小。其实人们日常生活中大量做的还是纯粹整数的计算，所以整数的用处还是很大的。

2.2表达式计算

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

1.运算符是指进行运算的动作，比如加法运算符"+"，减法运算符"-"

2.算子是指参与运算的值，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

```
int hour1,minute1;
int hour2,minute2;

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("时间差是%d小时%d分钟",t/60,t%60);
```

求平均值

写一个程序，输入两个整数，输出他们的平均值

```
int a,b;
scanf("%d %d",&a &b);
double c = (a + b)/2.0;
printf("%d和%d的平均数是:%f",a,b,c);
```

运算符优先级

运算符优先级

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

只有一个算子的运算符：+,-

例如-a，-b，+a，+b

表达式可以使用a*-b类似的形式

赋值运算符

1.赋值也是运算，也有结果

2.a=6的结果是a被赋予的值，也就是6

3.a=b=6　在计算机中形成原理 a = (b=6)

嵌入式赋值

提示：尽量避免"嵌入式赋值"，不利于阅读也容易产生错误

例如：

```
result = a = b = 3 + c;
result = 2;
result = (result = result * 2) * 6 * (result = 3 + result);
```

计算复利

在银行存定期的时候，可以选择到期后自动转存，并将到期的利息计入本金合并转存。如果1年期的定期利率是3.3%，那么连续自动转存3年后，最初存入的x元定期会得到多少本息金额?

本息合计：x(1 + 3.3%)三次方

```
int x;
scanf("%d",&x);
double amount = x * (1 + 0.033) * (1 + 0.033) * (1 + 0.033);
printf("%f",amount);
```

要计算任意年以后的本息金额，就需要做(1+0.033)的n次方计算

交换两个变量

如果已经有：int a=6，b = 5　如何交换a，b两个变量值

答：需要有第三个变量来进行缓冲，如下：

int t = a ， a= b ， b = t ；

复合赋值

5个算术运算符，加 减 乘 除 取余可以和赋值运算符“=”结合起来，形成复合赋值运算符：“+=”“-=”“*=”“/=”“%=”

total += 5;

total = total + 5;

注意两个运算符中间不要有空格

递增递减运算符

1."++""--"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递增和递减运算符，他们的作用就是给这个变量+1或者-1

2.以下都是同一个意思：

count++　count +=1　count = count + 1

3.递增递减可以放在前面(前缀)马上生效,也可以放在后面(后缀)延迟生效

4.可以单独使用，但不要组合进表达式

第三周：3.1判断

计算时间差

输入两个时间，每个时间分别输入小时和分钟的值，然后输出两个时间之间的差，也以几小时几分表示

问题：如果直接分别减，会出现分钟借位的情况：1点40分和2点10分的差?

如果

1.用分别减的方案，然后判断有没有出现借位行不行?

2.借位的表现是，分钟减的结果小于0，找小时借一位，如下:im代指分钟，ih代指小时

```
if(im < 0){
    im = 60 + im;
    ih--;
}
```

if(条件成立){

....

}

判断的条件

计算两个值之间的关系，所以叫做关系运算

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

关系运算的结果

1. 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
2. printf("%d\n",5==3);
3. printf("%d\n",5>3);
4. printf("%d\n",5<=3);

优先级

1. 所有的关系运算符的优先级比算数运算的低，但是比赋值运算的高
2. 7>=3+4 在这里面3+4先运算然后等于7，所以式子是可以运行的
3. int r = a > 0; 在这里a>0先运算，式子成立，答案为1，所以r = 1

找零计算器

1.找零计算器需要用户做两个操作：输入购买的金额，输入支付吧票面，而找零计算器则根据用户的输入做出相应的动作：计算并打印找零，或告知用户余额不足以购买。

2.从计算机程序的角度看，这就是意味着程序需要读用户的两个输入，然后进行一些计算和判断，最后输出结果

```
//初始化
int price = 0;
int bill = 0;
//读入金额和票面
printf("请输入金额：");
scanf("%d",&price);
printf("请输入票面：");
scanf("%d",&bill);
//计算找零
printf("应该找您：%d\n",bill - price);
```

双斜杠//注释

双斜杠//是注释(comment)的意思：插入在程序代码中，用来向读者提供解释信息。它们对于程序的功能没有影响，但是往往能使得程序更容易被人类读者理解

/* */注释

1.延续多行数行的注释，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

2.也可以用于一行内的注释

3.int ak = 47/*36*/，y = 9;

当票面不够的情况

1.当票面不够的情况怎么办？这个时候就需要进行判断用户读入的金额是否超过了票面

2.对计算找零这个步骤进行优化:

```
if( bill >= price){
   printf("需要找您：%d\n元",bill - price);
}
```

这个是当票面够的情况才会显示出需要找多少钱，如果钱不够的话则就不会显示出需要找钱了

3.如果当钱不够的情况需要提示用户"你的钱不够"怎么做?

错误示范:

```
if( bill >= price){
   printf("需要找您：%d/n元",bill - price);
}
printf("你的钱不够\n");
```

这种情况不管钱够不够都会输出 你的钱不够

else

else = 否则的话

正确的做法是:

```
if( bill >= price){
   printf("需要找您：%d/n元", bill - price);
}else{
   printf("你的钱不够\n");
}
```

比较数的大小

比较数的大小的各种方法

```
int a,b;
printf("请输入两个整数：");
scanf("%d %d", &a, &b);

int max = 0;
if(a>b){
    max = a;
}

printf("大的那个是%d\n", max);
```

这个代码里面没有解决b>a的问题，当a>b的条件不成立时，程序就结束了，max没有得到值

方案有很多，列举3个

```
int a,b;
printf("请输入两个整数：");
scanf("%d %d", &a, &b);

int max = 0;
if(a>b){
    max = a;
}
```

```
if(b>a){
    max = b;
}
printf("大的那个是%d\n", max);
int a,b;
printf("请输入两个整数：");
scanf("%d %d", &a, &b);

int max = 0;
if(a>b){
    max = a;
}else{
    max = b;
}

printf("大的那个是%d\n", max);
int a,b;
printf("请输入两个整数：");
scanf("%d %d", &a, &b);

int max = b;
if(a>b){
    max = a;
}

printf("大的那个是%d\n", max);
```

if语句

在上面中我们已经充分了解到了if语句的用法跟含义

但其实if也可以不带中括号去执行(不建议)，只能执行if接下来的一句内容

例如：

```
const double RATE = 8.25;
const double STANDARD = 40;
double pay = 0.0;
int hours;

printf("请输入工作的小时数：");
scanf("%d",&hours);
printf("\n");
if(hours > STANDARD)
    pay = STANDARD * RATE + (hours - STANDARD) * (RATE * 1.5);
else
    pay = hours * RATE;
printf("应付工资:%f\n",pay);
#include <stdio.h>

int main()
{

    const int MINOR = 35;
```

```
    int age = 0;

    printf("请输入你的年龄: ");
    scanf("%d", &age);

    printf("你的年龄是%d岁。\n", age);

    if (age < MINOR) {
        printf("年轻是美好的，");
    }

    printf("年龄决定了你的精神世界，好好珍惜吧。\n");

    return 0;
}
```

3.2分支

嵌套的判断

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或者if-else语句，这就是嵌套的if语句

else的匹配

else总是和最近的那个if匹配

缩进

缩进格式不能暗示else的匹配

```
if( code ==READY )
    if( count < 20 )
        printf("一切正常\n");
else
        printf("继续等待\n");
```

tips

1.在if或者else后面总是用{}

2.即使只有一条语句的时候

分段函数

```
if( x < 0 ){
  f = -1;
}else if( x == 0){
  f = 0;
} else {
  f = 2 * x;
}
```

if语句常见的错误

1.忘了大括号(永远在if和else后面加上大括号，即使当时后面只有一条语句)

2.if后面忘了分号

3.错误使用==和=

4.使人困惑的else

switch-case

1.控制表达式只能是整数型的结果

2.常量可以是常数，也可以是常数计算的表达式

3.根据表达式的结果，寻找匹配的case,并执行case后面的语句，一直到break为止

4.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){
case常量:
        语句
        ...
case常量:
        语句
        ...
default:
        语句
        ...
}
```

break

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。分支标号只是说明switch内部位置的路标，在执行完分支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

第四周：循环

4.1循环

三位数逆序的题

```
#include <stdio.h>
int main(){
    int x;
    int n=0;
    scanf("%d", &x);
    n++;
    x /= 10;
    while(x>0){
        n++;
```

```
        x /= 10;
    }
    printf("%d\n", n);
    return 0;
}
```

while循环

1.如果我们把while翻译作"当"，那么一个while循环的意思就是：当条件满足的时候，不断的重复循环体内的语句

2.循环执行前判断是否继续循环，所以可能一次循环一次也没有执行

3.条件成立是循环继续的条件

验证

1.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

2.个位数，10，0，负数

调试

在程序适当的地方插入printf()来输出变量的内容

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，不满足则循环结束

```
do{
  循环体语句
}while(循环条件);
```

两种循环

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

4.2循环计算

小套路

计算之前先保存原始的值，后面可能有用。重新定义一个变量来存放计算后的值

计数循环

```
#include <stdio.h>

int main()
{
    int n = 3;
```

```
    while ( n>= 0 ) {
        printf("%d ", n);
        n--;
    }
    printf("发射\n");

    return 0;
}
```

1.这个循环需要执行多少次?

2.循环停下来的时候，有没有输出最后的0?

3.循环结束后，count的值是多少?

技巧：如果要模拟运行一个很大次数的循环，可以模拟较少的循环次数，然后做出推断

循环应用(猜数游戏)

需求

让计算机来想一个数，然后让用户来猜，用户每输入一个数，就告诉用户是大了还是小了，直到用户猜到为止，最后还要告诉用户他猜了几次

思路

因为需要不断重复让用户猜，所以需要用到循环

在实际写出程序之前，我们可以先用文字描述程序的思路

核心重点是循环的条件

人们往往会考虑循环终止的条件

随机数

每次召唤rand()就可以得到一个随机数

代码块

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
    //不加srand(time(0));的话，每次循环的数都一样
    srand(time(0));
    int number = rand()%100+1;
    int count = 0;
    int a = 0;
    printf("我已经想好了一个1到100之间的数。");
    do {
        printf("请猜这个1到100之间数：");
```

```
        scanf("%d", &a);
        if ( a > number ) {
            printf("你猜的数大了。");
        } else if ( a < number ) {
            printf("你猜的数小了。");
        }
        count ++;
    } while (a != number);
    printf("太好了，你用了%d次就猜到了答案。\n", count);

    return 0;
}
```

%100

x%n的结果是[0,n-1]的一个整数

算平均数

需求

1.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

2.变量->算法->流程图->程序

变量

1.一个记录读到的整数的变量

2.平均数要怎么算?

3.只需要每读到一个数，就把它加到一个累加的变量里，到全部数据读完，再拿它去除读到的数的个数就可以了

4.一个变量记录累加的结果，一个变量记录读到的数的个数

算法

1.初始化变量sum和count为0;

2.读入number;

3.如果number不是-1，则将number加入sum，并将count加1，回到2;

4.如果number是-1，则计算和打印出sum/count(注意换成浮点数来计算)

代码块

```
#include <stdio.h>

int main()
{
    int sum = 0;
    int count = 0;
    int number;

    scanf("%d", &number);
```

```
    while ( number != -1 ) {
        sum += number;
        count ++;
        scanf("%d", &number);
    }

    double dsum = sum;
    printf("The average is %f.\n", dsum / count);

    return 0;
}
```

整数的分解

1.一个整数是由1-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

2.对一个整数做%10的操作，就得到它的个位数

3.对一个整数做/10的操作，就去掉了它的个位数

4.然后再对2的结果做%10,就得到原来数的十位数

....以此类推

数的逆序

1.输出一个正整数，输出逆序的数

2.结尾的0的处理

第五周：循环控制

##

阶乘

1.n! = 1×2×3×4×...×n

2.写出一个程序，让用户输入n，然后计算输出n!

3.变量：显然读用户的输入需要一个int的n，然后计算的结果需要用一个变量保存，可以是int的factor，在计算中需要有一个变量不断地从1递增到n，那可以是int的i

```
int n;

scanf("%d",&n);
int fact = 1;

int i = 1;
for( i = 1;i <= n; i++){
    fact *= i;
}

printf("%d!=%d\n",n,fact);
```

5.1 for循环(第三种循环)

for循环像是一个计数循环: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 步进 进行调整，比如加一或者减一

```
例如：
for(i = 0;i < 5;i++){
  printf("%d",i);
}


for(初始动作;条件;每轮的动作){
}
//for中的每一个条件都是可以省略的，for(;条件;) == while(条件)
```

for = 对于

for(count = 10; count > 0; count--)

可以解读成：对于一开始的count = 10，当count > 0的时候，重复做循环体，每一轮循环在做完循环体内语句后，使得count--

小套路

做求和的程序时，记录结果的变量应该初始化为0，而做求积的变量时，记录结果的变量应该初始化为1

循环控制变量i只在循环里被使用了，在循环外面它没有任何用处。因此，我们可以把变量i的定义写到for语句里面去

尝试(try)

1. 1×1还是1，所以程序的循环需不需要从1开始，那么改成多少开始合适呢？这样修改之后，程序对于所有的n都正确吗？这样的改动有价值吗？
2. 除了可以从1乘到n来计算n！，还可以从n乘到1来计算吧？试试换个方向来计算n。这时候，还需要循环控制变量i吗？

for == while

```
for( int i = 1;i <= n;i++ ){
    fact *= i;
}
-----------------------------------------
int i = 1;
while(i <= n){
    fact *=i;
    i++;
}
```

循环次数

for(i = 0; i < n; i++)

这样循环的次数是n，而循环次数结束以后，i的值是n。循环的控制变量i，是选择从0开始还是从1开始，是判断 i<n还是判断i <=n，对循环次数，循环结束后变量的值都是有影响的

Tips for Loops(小贴士循环)

1. 如果有固定次数，用for
2. 如果必须执行一次，用 do while
3. 其他情况用while

5.2循环控制

素数

只能被1和自身整除的数，不包括1

break VS continue

1. break:跳出循环
2. continue:跳过循环这一轮剩下的语句进入下一轮
3. 都只能对它所在的那层循环生效

嵌套的循环

1. 意思：循环的里面还是循环
2. 嵌套循环时的break只会跳出当前所在的循环，如果嵌套了多层循环就会被卡在下一层循环上而无法真正的跳出所有循环

100以内的素数

如何写程序输出100以内的素数?

```
#include <stdio.h>

int main()
{
    int x;
    //我们需要有一个循环，从 1 到 100
    for(x=2; x<=100; x++){
        int i;
        int isPrime = 1; // x是素数
        for ( i=2; i<x; i++ ) {
            if ( x % i == 0 ) {
                isPrime = 0;
                break;
            }
```

```
        }
        //当它是素数的时候就输出出来，不是素数就不要输出任何东西了
        if ( isPrime == 1 ) {
            printf("%d ", x);
        }
    }
    return 0;
}
```

凑硬币

如何用1角、2角和5角的硬币凑出10元以下的金额?

```
#include <stdio.h>

int main()
{
    int x;
    int one, two, five;
    int exit = 0;
    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 == x*10 ) {
                    printf("可以用%d个1角加%d个2角加%d个5角得到%d元\n",
                           one, two, five, x);
                    exit = 1;
                    break;
                }
            }
            if ( exit == 1 ) break;
        }
        if ( exit == 1 ) break;
    }
    return 0;
}
```

5.3循环应用

正序分解整数

1. 输入一个非负整数，正序输出它的每一位数字
2. 输入：13425
3. 输出：1 3 4 2 5

分解整数输出(未解决结尾空格问题版本)

```
int x;
scanf("%d",&x);

do{
```

```
   int d = x % 10;
    printf("%d",d);
        x /= 10;
}while( x > 0);
printf("\n");
//(未解决结尾空格问题版本)
-----------------------------------------
int x;
scanf("%d",&x);

do{
   int d = x % 10;
    printf("%d",d);
        if( x > 9){
           printf(" ");
        }
        x /= 10;
}while( x > 0);
printf("\n");

-----------------------------------------------
    (如果有一个mask的话)
int x;
scanf("%d",&x);

x = 13425;
int mask = 10000;
do{
   int d = x % 10;
    printf("%d",d);
        if( x > 9){
           printf(" ");
        }
        x %= mask;
        mask /= 10;
}while( mask > 0);
printf("\n");
-------------------------------------------------
 //    (计算x的位数)
x = 12345;
int mask = 10000;
int n = 0;
do{
  x /= 10;
     n++;
}while( x > 0);
printf("n = %d\n",n);
```

求最大公约数

1. 输入两个数a和b，输出它们最大的公约数
2. 输入：12，18
3. 输出：6
4. 枚举方法:过于麻烦，耗费较多资源

```c
#include <stdio.h>

int main()
{
int a,b;
int min;
scanf("%d %d", &a, &b);
if ( a<b ) {
min = a;
} else {
min = b;
}
int ret = 0;
int i;
for ( i = 1; i < min; i++ ) {
if ( a%i == 0 ) {
if ( b%i == 0 ) {
ret = i;
}
}
}
printf("%d和%d的最大公约数是%d.\n", a, b, ret);
return 0;
}
```

辗转相除法

1. 如果b等于0，计算结束，a就是最大的公约数;
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

```c
int a,b;
int t;

scanf("%d %d",&a,&b);
int origa = a;
int origb = b;
while( b != 0 ){
    t = a%b;
    a = b;
    b = t;
}
printf("%d和%d的最大公约数是%d",orida,origb,a);
```

第六周:数据类型

编程类型解析

```
//给定不超过6的正整数A，考虑从A开始的连续4个数字。请输出所有由它们组成的无重复数字的3位数
//输出格式：满足条件的3位数，要求从大到小，每行6个整数，整数间以空格分隔，但行末不能有多余空格
int main()
{
    int a;
    scanf("%d",&a);
    int i,j,k;
    int cnt = 0;

    i = a;
    while( i<=a+3){
        j = a;
        while( j<=a+3){
            k = a;
            while( k<=a+3){
                if( i!=j ){
                    if( i!=k ){
                        if( j!=k){
                            cnt++;
                            printf("%d%d%d",i,j,k);
                            if( cnt == 6){
                                printf("\n");
                                cnt = 0;
                            }else{
                                printf(" ");
                            }
                        }
                    }
                }
                k++;
            }
            j++;
        }
        i++;
    }
    return 0;
}
#include<stdio.h>
int main() {
    int n;
    //scanf("%d", &n);
    n=3;
    int first = 1;
    int i = 1;
    while(i < n){
        first *= 10;
        i++;
    }
    //printf("first=%d\n", first);
    //遍历100-999
    i = first;
    while(i < first*10){
        //需要一个临时的变量去记录 i
```

```
        int t = i;
        //需要一个"和"去记录每一位数的 n次幂
        int sum = 0;
        do {
            int d = t % 10;
            t /= 10;
            //d^2 = d*d;  d^3 = d*d*d;
            /*
            int p = 1;
            int j = 0;
            while(j < n){
                p *= d;
                j++;
            }
            //或者*/
            int p = d;
            int j = 1;
            while(j < n){
                p *= d;
                j++;
            }
            sum += p;
        } while(t > 0);
        if (sum == i) {
            printf("%d\n", i);
        }
        i++;
    }
    return 0;
}
#include<stdio.h>
int main() {
    int n;
    //scanf("%d", &n);
    n = 9;
    int i,j;
    i = 1;
    while( i <= n) {
        j = 1;
        while(j <= i){
            printf("%d*%d=%d", j,i,i*j);
            //i,j会输出9*1= 9*2=
            //我们想要输出1*9= 2*9=
            //如果i*j小于10，比如1*1=1 小于 10
            if( i*j < 10){
                //输出三个空格
                printf("   ");
            } else {
                //输出两个空格
                printf("  ");
            }
            j++;
        }
        //还需要在每一个行加个回车
        printf("\n");
```

```
        i++;
    }

    return 0;
}
#include<stdio.h>
int main() {
    int m,n;
    int i;
    //个数
    int cnt = 0;
    int sum = 0;

    scanf("%d %d", &m, &n);
    //如果m是 1，单独做一个特殊的判断
    if(m == 1)
        m=2;

    for(i=m; i<=n; i++){
        int isPrime = 1;
        int k;
        //需要有另外一个循环去证明 i是不是isPrime
        for(k=2; k<i-1; k++){
            if(i%k == 0){
                isPrime = 0;
                break;
            }
        }
        // 判断 i 是否素数
        if(isPrime) {
            cnt++;
            sum+=i;
        }
    }
    printf("%d %d\n", cnt, sum);

    return 0;
}
#include <stdio.h>
int main(){
    //随机数，猜测的最大次数
    int number,n;
    //用户每次猜测的数字
    int inp;
    //  finished为 1 则表示猜中，0 没猜中
    int finished = 0;
    // 记录猜测次数
    int cnt = 0;
    scanf("%d %d",&number,&n);
    while(scanf("%d",&inp)){
        cnt++;
        //如果这个数小于0 或 猜测次数大于最大猜测次数 ,则输入不合法
        if(inp < 0 || cnt > n){
            printf("Game Over\n");
            break;
```

```
        }
        // 输入猜测数太大
        if(inp > number){
            printf("Too big\n");
        // 输入猜测数太小
        }else if(inp < number){
            printf("Too small\n");
        // 输入猜测数与随机数相等
        }else{
            // 1次成功
            if(cnt == 1){
                printf("Bingo!\n");
             // 3次以内成功
            }else if(cnt <= 3){
                printf("Lucky You!\n");
            // 3次以上成功
            }else{
                printf("Good Guess!\n");
            }
            finished = 1;
            if(finished == 1) break;
        }
    }
    return 0;
}
#include<stdio.h>
int main() {
    int n;
    //分子，分母
    double dividend,divisor;
    double sum = 0.0;
    int i;
    double t;

    scanf("%d", &n);
    // n = 2000;
    dividend = 2;
    divisor = 1;
    for(i = 1; i <= n; i++){
        sum += dividend/divisor;
        t = dividend;
        dividend = dividend + divisor;
        divisor = t;
    }
    printf("%.2f\n", sum);

    return 0;
}
#include<stdio.h>
int main() {
    //分子，分母
    int dividend,divisor;
    scanf("%d/%d", &dividend, &divisor);

    int a = dividend;
```

```
    int b = divisor;
    int t;
    //辗转相除法算出最大公约数
    while(b > 0){
        t = a % b;
        a = b;
        b = t;
    }
    printf("%d/%d\n", dividend/a, divisor/a);

    return 0;
}
#include<stdio.h>
int main() {
    int x;
    scanf("%d", &x);

    if(x < 0){
        printf("fu ");
        x = -x;
    }
    int mask = 1;
    int t = x;
    //辗转相除法算出最大公约数
    while(t > 9){
        t /= 10;
        mask *= 10;
    }

    do{
        int d = x / mask;
        switch(d){
            case 0: printf("ling"); break;
            case 1: printf("yi"); break;
            case 2: printf("er"); break;
            case 3: printf("san"); break;
            case 4: printf("si"); break;
            case 5: printf("wu"); break;
            case 6: printf("liu"); break;
            case 7: printf("qi"); break;
            case 8: printf("ba"); break;
            case 9: printf("jiu"); break;
        }
        // 最后不要有行末的空格
        if (mask > 9) printf(" ");
        x %= mask;
        mask /= 10;
    } while(mask > 0);
    printf("\n");

    return 0;
}
#include<stdio.h>
int main() {
    int a,n;
```

```
    scanf("%d %d", &a, &n);

    int sum = 0;
    int i;
    //每一轮的数字
    int t = 0;
    // 0*10+2 2*10+2 (2*10+2)*10+2
    for(i=0; i<n; i++){
        t = t*10+a;
        sum += t;
    }
    printf("%d\n", sum);

    return 0;
}
```

C语言的类型

整数

1. char：一字节（8比特）-128至127
2. short：2字节 -32768至32767
3. int：取决于编译器（CPU），通常的意义都是"1个字"
4. long：取决于编译器（CPU），通常的意义是"1个字"(4或8字节)
5. long long：8字节

*整数的内部表达

1. 计算机内部一切都是二进制
2. 18 -> 00010010
3. 0 -> 00000000
4. -18 ->?

*如何表示负数

1. 十进制用"-"来表示负数，在做计算的时候
2. 加减是做相反的运算
3. 乘除时当作正数，计算完毕后对结果的符号取反

*二进制负数

1. 1个字节可以表达的数：0000 0000 - 1111 1111(0-255)

三种方案：

1. 仿照十进制，有一个特殊的标志表示负数
2. 取中间的数为0，如100 0000表示0，比它小的是负数，比它大的是正数
3. 补码(实际上使用的，方案1和2都有缺陷)

*补码

补码的意义就是拿补码和原码可以加出一个溢出的0

例子：因为0 - 1->-1,所以-1 = (1)0000 0000 - 0000 0001 ->1111 1111

1111 1111被当作纯二进制看待时，是255，被当作补码看待时是-1

同理，对于-a，其补码就是0-a，实际是2的n次方-a，n是这种类型的位数

数的范围

1. 对于一个字节(8位)，可以表达的是：0000 0000 ->1111 1111
2. 其中0000 0000->0 、1111 1111至1000 0000 ->-1至-128 、 0000 0001至0111 1111->1至127

浮点数

float、double、long double

逻辑

bool

指针

自定义类型

类型有何不同

1. 类型名称：int、long、double
2. 输入输出时的格式化：%d、%ld、%lf
3. 所表达的数的范围：char<short<int<float<double
4. 内存中所占据的大小：1个字节到16个字节
5. 内存中的表达形式：二进制数(补码)、编码

sizeof

1. 是一个运算符，给出某个类型或者变量在内存中所占据的字节数

例如：sizeof(int)、size(i)

1. 是静态运算符，它的结果在编译时刻就决定了
2. 不要在sizeof的括号里做运算，这些运算不会进行的

unsigned

1. 在整形类型前加上unsigned使得它们变成无符号的整数
2. 内部的二进制表达没变，变的是如何看待它们。如何输出
3. 1111 1111对于char来说是-1，对于unsigned char来说是255
4. 如果一个字面量常数想要表达自己是unsigned，可以在后面加上U或者u，255U
5. 用L或者l表示long(long)
6. unsigned初衷并非扩展数能表达的范围，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：

1. 1111 1111+1 -> 1 0000 0000 ->0
2. 0111 1111+1 -> 1 0000 0000 ->-128
3. 1000 0000-1 -> 0111 1111 ->127

整数的输入输出

只有两种形式：int或者long long

1. %d：int
2. %u：unsigned
3. %ld：long long
4. %lu：unsigned long long

8进制和16进制

1. 一个以0开始的数字字面量是8进制
2. 一个以0x开始的数字字面量是16进制
3. %o用于8进制，%x用于16进制
4. 8进制跟16进制只是如何把数字表达为字符串，与内部如何表达数字无关
5. 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
6. 8进制的一位数字正好表达3位二进制
7. 因为早期计算机的字长是12的倍数，而非8

选择整数类型

1. 为什么整数要有那么多种？为了准确表达内存，做底层程序的需要
2. 没有特殊需要就选择int
3. 现在CPU的字长普遍是32位或者64位，一次内存读写就是一个int，一次计算也是一个int，选择更短的类型不会更快，甚至可能更慢
4. 现代的编译器一般会设计内存对齐，所以更短的类型实际在内存中有可能也占据一个int的大小(虽然sizeof告诉你更小)
5. unsigned与否只是输出的不同，内部计算是一样的

浮点数

浮点类型

类型	字长	范围	有效数字
float	32	正负(1.20×100,正负inf，NaN)	7
double	64	正负(2.20×100,正负inf，NaN)	15

浮点的输入输出

类型	scanf	printf
float	%f	%f，%e
double	%lf	%f，%e

科学计数法

1. 可选的+或者-符号
2. 可以用e或者E
3. 整个词不能有空格
4. 小数点也是可选的

5. 符号可以是正负也可以省略(表示+)
6. 例如：-5.67E+16

输出精度

在%和f之间加上.n可以指定输出小数点后几位，这样的输出是做四舍五入的

1. printf("%.3f\n",-0.0049);
2. printf("%.30f\n",-0.0049);
3. printf("%.3f\n",-0.00 49);

超出范围的浮点数

1. printf输出inf表示超过范围的浮点数：+无穷
2. printf输出nan表示不存在的浮点数

浮点运算的精度

1. 带小数点的字面量是double而非float
2. float需要用f或者F后缀来表明身份

浮点数的内部表达

1. 浮点数在计算时是由专用的硬件部件实现的
2. 计算double和float所用的部件是一样的

选择浮点类型

1. 如果没有特殊需要，只使用double
2. 现代CPU能直接对double做硬件运算，性能不会比float差，在64位的机器上，数据存储的速度也不比float慢

字符

字符类型

char是一种整数，也是一种特殊的类型：字符。这是因为：

1. 用单引号表示的字符字面量：'a'，'1'
2. "也是一个字符
3. printf和scanf里用%c来输入输出字符

字符的输入输出

1. 如何输入'1'这个字符给char c?

scanf("%c"，&c);->1

scanf("%d"，&i);c=i ;->49

1. '1'的ASCII编码是49，所以当c==49时，它代表'1'
2. 这是一个49的各自表述

混合输入

有何不同

scanf("%d %c",&i,&c);

scanf("%d%c",&i,&c);

字符计算

1. 一个字符加一个数字得到ASCII码表中那个数之后的字符
2. 两个字符的减，得到它们在表中的距离

大小写转化

1. 字母在ASCII表中是顺序排列的
2. 大写字母和小写字母是分开排列的，并不在一起
3. ‘a’-‘A’可以得到两段之间的距离，于是a+‘a’-‘A’可以把一个大写字母变成小写字母，反之把这个运算中的‘a’与‘A’互相调换即可把一个小写字母变成大写字母

逃逸字符

用来表达无法印出来的控制字符或特殊字，在那些具有特殊符号的前面加上\即可将其当作普通字符使用

字符	意义
\b	回退一格
\t	到下一个表格位
\n	换行
\r	回车
"	双引号
'	单引号
\	反斜杠本身

制表位

1. 每行的固定位置
2. 一个\t使得输出从下一个制表位开始
3. 用\t能使上下两行对齐

逻辑类型

bool

1. 在宏定义中加上#include<stdbool.h>
2. 之后就可以使用bool和true跟false

bool的运算

1. bool实际上还是以int的手段实现的，所以可以当作int来计算
2. 也只能当作int来输入输出

类型转换

自动类型转换

1. 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
2. 大的意思是能表达的数的范围更大
3. char->short->int->long->long long
4. int->float->double
5. 对于printf，任何小于int的类型会被转化成int;float会被转换成double
6. 但是scanf不会，要输入short，需要%hd

强制类型转换

1. 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
2. 比如(int)10.2 (short)32
3. 需要注意这个时候的安全性，小的变量不总能表达大的量
4. 这个只是从那个变量计算出了一个新的类型的值，它并不改变那个变量，无论值还是类型都不改变
5. 强制类型转换的优先级高于四则运算

有些运算

逻辑运算

1. 逻辑运算是对逻辑量进行的运算，结果只有0或者1
2. 逻辑量是关系运算或逻辑运算的结果

运算符	描述	示例	结果
!	逻辑非	!a	如果a是true结果就是false如果a是fasle结果就是true
&&	逻辑与	a&&b	如果a和b都是是true结果就是true否则就是false
||	逻辑或	a||b	如果a和b有一个是true结果就是true两个都是false，结果才是false

try(尝试)

如何判断一个字符c是否是大写字母?

c >= ‘A’&&c<=‘Z’

逻辑优先级

!>&&>||

总体优先级排名

优先级	运算符	结合性
1	()	从左到右
2	！ +- ++ --	从右到左(单目的+和-)
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

1. 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了
2. 对于&&，左边是false时就不会做右边的运算了
3. 对于||，左边是true时就不会做右边的运算了
4. 不要把赋值，包括复合赋值组合进表达式

条件运算符

1. count = (count>20)？count-10：count+10;
2. 当count>20是真的时候执行前者，是假执行后者
3. 优先级:条件运算符的优先级高于赋值运算符，但是低于其他运算符
4. 不要嵌套条件表达式(很容易出问题)

逗号运算符

1. 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运算符中最低的，所以它两边的表达式会先计算;逗号的组合关系是自左像右，所以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
2. 比如在for中使用
3. for(i = 1,j=8;i<j;i++,j--)

第七周：函数

7.1函数的定义与使用

```
#include<stdio.h>
int main() {
```

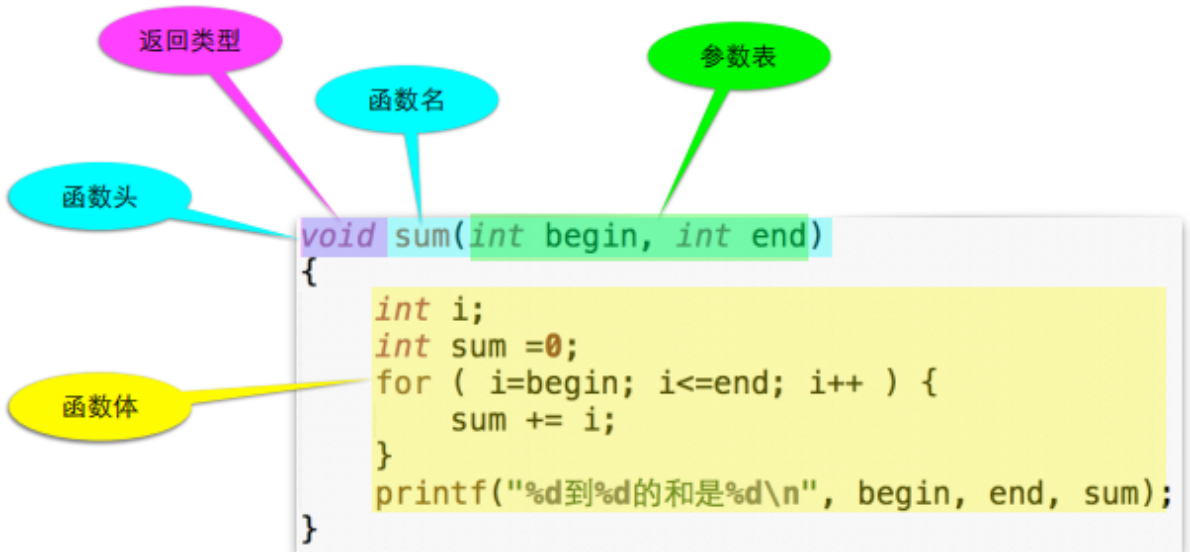
```
    int m,n;
    int sum = 0;
    int cnt = 0;
    int i;

    scanf("%d %d",&m, &n);
    // m=10,n=31;
    if(m == 1) m=2;
    for(i=m; i<=n; i++){
        int isPrime = 1;
        int k;
        for(k=2; k<i-1; k++){
            if(i%k == 0){
                isPrime = 0;
                break;
            }
        }
        if(isPrime){
            sum += i;
            cnt++;
        }
    }
    printf("%d %d\n", cnt, sum);

    return 0;
}

// 把这段代码取出来
int isPrime = 1;
        int k;
        for(k=2; k<i-1; k++){
            if(i%k == 0){
                isPrime = 0;
                break;
            }
        }
#include<stdio.h>
//求1到10、20到30和35到45的三个和
void sum(int begin,int end)
{
    int i;
    int sum = 0;
    for( i = begin;i <= end; i++){
        sum += i;
    }
    printf("%d到%d的和是%d\n",begin,end,sum);
}

int main()
{
    sum(1,10);
    sum(20,30);
    sum(35,40);

    return 0;
```

```
}
```

什么是函数?

1. 函数是一块代码，接收零个或者多个参数，做一件事情，并返回零个或一个值
2. 可以先想像成数学中的函数：y=f(x)

函数定义



调用函数

1. 函数名(参数值);
2. ()起到了表示函数调用的重要性，即使没有参数也需要()
3. 如果有参数，则需要给出正确的数量和顺序
4. 这些值会被按顺序依次用来初始化函数中的参数

函数返回

函数知道每一次是在哪里调用它，会返回到正确的地方

从函数中返回值

1. return停止函数的执行，并送回一个值
2. return;
3. return 表达式;
4. 一个函数里可以出现多个return语句
5. 例如:int c, c = max(10,12);=>可以赋值给变量或者再传递给函数甚至可以丢弃，有的时候要的是副作用

没有返回值的函数

1. void 函数名(参数表)
2. 不能使用带值的return
3. 可以没有return
4. 调用的时候不能做返回值的赋值
5. 如果函数有返回值则必须使用带值的return

7.2函数的参数和变量

函数原型

函数的先后关系

```
#include<stdio.h>
void sum(int begin,int end)
{
    int i;
    int sum = 0;
    for( i = begin; i <= end; i++){
        sum += i;
    }
    printf("%d到%d的和是%d\n",begin,end)
}

int main()
{
    sum(1,10);
    sum(20,30);
    sum(35,45);

    return 0;
}
```

1. 像这样把sum()写在上面，是因为：c的编译器自上而下顺序分析你的代码
2. 在看到sum(1,10)的时候，它需要知道sum()的样子
3. 也就是sum()要几个参数，每个参数的类型如何，返回什么类型。这样它才能检查你对sum()的调用是否正确

此函数原型

```
#include<stdio.h>
double max(double a,double b)    =>   这是函数原型

    int main()
{
int a,b,c;
    a = 5;
    b = 6;
    c = max(10,12);         =>这中间一段是根据原型判断
    printf("%d\n",c);
    max(12,13);

    return 0;
}

double max(double a, double b)   =>这是实际的函数头
```

1. 函数头：以分号";"结尾，就构成了函数的原型
2. 函数原型的目的是告诉编译器这个函数长什么样子：名称、参数(数量及类型)、返回类型
3. 旧标准习惯把函数原型写在调用它的函数里面
4. 现在一般写在调用它的函数前面
5. 原型里可以不写参数的名字，但是一般仍然写上

参数传递

调用函数

1. 如果函数有参数，调用函数的时候必须传递给它数量、类型正确的值
2. 可以传递给函数的值是表达式的结果，这包括:字面量、变量、函数的返回值、计算的结果

```
int a,b,c;
a = 5;
b = 6;
c = max(10,12);
c = max(a,b);
c = max(c,23);
c = max(max(23,45),a);
c = max(23+45,b);
```

类型不匹配

1. 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
2. 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
3. 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

```
#include<stdio.h>
void swap(int a,int b);  //这是参数

int main()
{
    int a = 5;
    int b = 6;

    swap(a,b);   //这是值

    printf("a = %d b = %d\n",a,b);

    return 0;
}

void swap(int a,int b)   //参数
{
    int t = a;
    a = b;
    b = t;
}
```

传值

1. 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
2. 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"(实参与形参)
3. 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，误会调用函数的时候把变量而不是值传进去，所以我们不建议继续使用这种古老的方式来称呼它
4. 我们认为，它们是参数和值的关系

本地变量

1. 函数的每次运行，就会产生一个独立的变量空间，在这个空间中的变量，是函数的这次运行所独有的，称作本地变量
2. 定义在函数内部的变量就是本地变量
3. 参数也是本地变量

变量的生存期和作用域

1. 生存期：什么时候这个变量开始出现了，到什么时候它消亡了
2. 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)
3. 对于本地变量,这两个问题的答案是统一的：大括号内——块

本地变量的规则

1.本地变量是定义在块内的

1. 它可以是定义在函数的块内
2. 也可以定义在语句的块内
3. 甚至可以随便拉一对大括号来定义变量

2.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

3.块外面定义的变量在里面仍然有效

4.块里面定义了和外面同名的变量则覆盖了外面的变量(块内的变量优先度更高)

5.不能在一个块内定义同名的变量

6.本地变量不会被默认初始化

7.参数在进入函数的时候被初始化了

其他细节

没有参数时

1. void f(void)还是void f();
2. 在传统C中，它表示f函数的参数表示未知，并不表示没有参数

逗号运算符?

1. 调用函数时的逗号跟逗号运算符字母区分?
2. 如下 => 调用函数时的圆括号内的逗号是标点符号，不是运算符
3. 逗号：f(a,b) 逗号运算符:f((a,b))

函数里的函数?

C语言不允许函数嵌套定义

这是什么?

1. int i,j,sum(int a,int b); //可以这样写但是不建议
2. return(i); //加不加括号其实都无所谓，值都一样的，但加上括号会让人误以为这是一个函数，不要这样写没有好处

关于main

1. int main()也是一个函数
2. 要不要写成int main(void)? //void加不加都一样，但如果上面参数都不打算加，不妨把void写下去
3. return的0有人看吗? //是可以看的起作用的，返回0表示正常的运行结束了，返回任何非0的值都是错误的

windows:if error level 1...

Unix Bash:echo $?

Csh:echo $status

第八周：数组

8.1-1初试数组

如何写入一个程序计算用户输入的数字的平均数?(不需要记录输入的每一个数)

```
#include<stdio.h>
int main(){
    int x;
    double sum = 0;
    int cnt = 0;
    scanf("%d",&x);
    while( x != -1){
        sum += x;
        cnt++;
        scanf("%d",&x);
    }
    if( cnt > 0 ){
        printf("%f\n",sum/cnt);
    }

    return 0;
}
```

1. 如何写一个程序计算用户输入的数字的平均数，并输出所有大于平均数的数?
2. 思路：必须先记录每一个输入的数字，计算平均数之后，再检查记录下来的每一个数字，与平均数比较，决定是否输出
3. 该使用数组啦

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

```c
int main(){
    int x;
    double sum = 0;
    int cnt = 0;
    int number[100];  //=>这里定义了数组，但具有隐患，当存放的内容超过了100个的时候就会出错，所有后续可以进行动态调整
    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){
        int i;
        double average = sum/cnt;
        for( i = 0;i < cnt; i++){
            if( number[i] > average){
                printf("%d",number[i]);  //遍历数组跟使用数组里面的元素
            }
        }
    }

    return 0;
}
```

8.1-2数组的定义和使用

定义数组

1.<类型>变量名称[元素数量];

1. int grades[100];
2. double weight[20];

2.元素数量必须是整数

3.C99之前：元素数量必须是编译时刻确定的字面量(了解下就行)

数组

是一种容器(放东西的东西)，特点是:

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

int a[10];

1. 一个int的数组
2. 10个单元：a[0],a[1],a[2]....a[9];
3. 每一个单元就是一个int类型的变量
4. 可以出现在赋值的左边或右边：a[2] = a[1]+6;
5. *在赋值左边的值叫左值(右边就叫右值咯)

数组的单元

1. 数组的每个单元就是数组类型的一个变量
2. 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:例如
   1.grades[0]2.grades[99],average[5]

有效的下标范围

1. 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
2. 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**
3. 也可能运气好，没造成严重的后果
4. 所有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]

计算平均数

```
int x;
double sum = 0;
int cnt;
printf("请输入数字的数量:");
scanf("%d",&cnt);
if( cnt > 0){
    int number[cnt];
    scanf("%d",&x);
    while( x != -1){
        number[cnt] = x;
        sum += x;
        cnt++;
        scanf("%d",&x);
    }
}
```

长度为0的数组?

int a[0];可以存在，但是没有卵用

8.1-3数组的例子：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 number = 10;  //数组的大小
    int x;
    int count[number];  //定义数组
    int i;

    for( i=0; i<number; i++){    //初始化数组
        count[i] = 0;
    }
    scanf("%d",&x);
    while( x != -1){
        if( x >= 0 && x <= 9){
            count[x]++;  //数组参与运算
        }
```

```
        scanf("%d",&x);
    }
    for( i = 0;i < number; i++){
        printf("%d:%d\n",i,count[i]);    //遍历数组输出
    }

    return 0;
}
```

8.2-1数组的运算

在一组给定的数据中，如何找出某个数据是否存在?

```
#include<stdio.h>
/**
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 如果找到,返回其在a中的位置;如果找不到则返回-1
*/
int search(int key,int a[], int length);

int main(void)
{
    int a[] = {2,4,6,7,1,3,5,9,11,13,23,14,32};
    int x;
    int loc;
    printf("请输入一个数字:");
    scanf("%d",&x);
    loc = search(x,a,sizeof(a)/sizeof(a[0]));
        if( loc != -1){
            printf("%d在第%d个位置上\n",x,loc);
        }else{
            printf("%d不存在\n",x);
        }
    return 0;
}

int search(int key, int a[], int length)
{
    int ret = -1;
    int i;
    for( i = 0; i <= length; i++){
        if( a[i] == key){
            ret = i;
            break;
        }
    }
    return ret;
}
```

数组的集成初始化

int a[] = {2,4,6,7,1,3,5,9,11,13,23,14,32};

1. 直接用大括号给出数组的所有元素的初始值
2. 不需要给出数组的大小，让编译器替你数

int b[20] = {2};

1. 如果给出了数组的大小，但是后面的初始值数量不足，则其后的元素被初始化为0

集成初始化时的定位

```
int a[10] = {
    [0] = 2,[2] = 3,6,
};
```

1. 用[n]在初始化数据中给出定位
2. 没有定位的数据接在前面的位置后面
3. 其他位置的值补零
4. 也可以不给出数组大小，让编译器算
5. 特别适合初始数据稀疏的数组

数组的大小

1. sizeof给出整个数组所占据的内容的大小，单位时字节

得到数组的个数: sizeof(a)/sizeof(a[0])

1. sizeof(a[0])给出数组中单个元素的大小，于是相除就得到了数组的单元个数
2. 这样的代码，一旦修改数组中初始的数据，不需要修改遍历的代码

数组的赋值

1. 数组变量本身不能被赋值
2. 要把一个数组的所有元素交给另一个数组，必须采用遍历

```
for( i = 0; i<length;i++){
    b[i] = a[i];
}
```

遍历数组

1. 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
2. 常见错误：循环结束条件是 <= 数组长度，或;　离开循环后，继续使用i的值来做数组元素的下标(错误的)

数组作为函数参数时：

1. 不能在[]中给出数组的大小
2. 不能再利用sizeof来计算数组的元素个数
3. **往往必须再用另一个参数来传入数组的大小**

8.2-2数组的例子:素数

判断素数

```
int is Prime(int x);

int main(void)
{
    int x;
    scanf("%d",&x);
    if( isPrime(x)){
        printf("%d是素数\n",x);
    }else{
        printf("%d不是素数\n",x);
    }
    return 0;
}
```

从2到x-1测试是否可以整除

```
int isPrime(int x)
{
    int ret = 1;
    int i;
    if( x == 1) ret = 0;
    for( i = 2;i < x; i++){
        if( x % i == 0 ){
            ret = 0;
            break;
        }
    }
    return ret;
}

//对于n要循环n-1遍，当n很大的时候就是n遍
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;
}
```

```
//如果x是偶数，立刻结束。否则要循环(n-3)/2+1遍。当n很大的时候就是n/2遍
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;
}
//只需要循环sqrt(x)遍，sqrt(x)是x的平方根的意思
```

判断是否可以被已知的且<x的素数整除

```
#include<stdio.h>
#include<math.h>
int isPrime(int x, int knownPrimes[],int numberOfKnownPrimes);

int main(void){
    const int number = 10;
    int prime[number] = {2};
    int count = 1;
    int i = 3;
    // 为了输出好看,添加测试语句
    {
        int i;
        printf("\t\t");
        for(i=0; i<10; i++){
            printf("%d\t", i);
        }
        printf("\n");
    }
    while(count < number){
        if(isPrime(i,prime,count)){
            prime[count++] = i;
        }
        // 加入调试输出的语句
        {
            printf("i=%d \tcnt=%d\t", i, count);
            int i;
            for(i=0; i<number; i++){
                printf("%d\t", prime[i]);
            }
            printf("\n");
        }
        i++;
    }
    for(i=0; i<number; i++){
        printf("%d",prime[i]);
        if((i+1)%5) printf("\t");
        else printf("\n");
```

```
    }
    return 0;
}

int isPrime(int x,int knownPrimes[],int numberOfKnownPrimes){
    int ret = 1;
    int i;
    for( i=0;i<numberOfKnownPrimes; i++){
        if(x%knownPrimes[i] == 0){
            ret = 0;
            break;
        }
    }
    return ret;
}
```

构造素数表

欲构造n以内的素数表

1. 令x为2
2. 将2x,3x,4x直至ax<n的数标记为非素数
3. 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

欲构造n以内(不含)的素数表

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
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

```
#include<stdio.h>
int main(void){
const int maxNumber = 25;
int isPrime[maxNumber];
int i;
int x;
for( i = 0;i < maxNumber;i++){
    isPrime[i] = 1;
}
for( x = 2;x < maxNumber;x++){
    if( isPrime[x] ){
        for(i = 2; i*x<maxNumber; i++){
            isPrime[i*x] = 0;
        }
    }
}
for( i = 2;i < maxNumber; i++){
    if( isPrime[i]){
        printf("%d\t",i);
    }
}
printf("\n");
    return 0;
}
```

8.2-3二维数组

二维数组

1. int a[3][5];
2. 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; i++){
        a[i][j] = i*j;
    }
}
```

1. a[i][j]是一个int
2. 表示第i行第j列上的单元
3. a[i,j]是什么？　//是一个表达式，不是正确的表达二维数组的方式

二维数组的初始化

```
int a[][5] = {
    {0,1,2,3,4},
    {2,3,4,5,6},
};
```

1. 列数是必须给出的，行数可以由编译器来数
2. 每隔一个{},逗号分隔
3. 最后一组逗号可以存在，以前的传统
4. 如果数组有部分省略没写，编译器会自动补零(表示补零)
5. 也可以使用定位

tic-tac-toe游戏

1. 读入一个3×3的矩阵，矩阵中的数字为1表示该位置有一个X，为0表示为O
2. 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或者O，或输出无人获胜

读入矩阵

```
const int size = 3;
int board[size][size];
int i,j;
int numOfX;
int numOfO;
int result = -1;   //-1：没人赢，1：X赢了，0：O赢了

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

检查行

```
//检查行
for( i = 0; i < size && result == -1; i++){
    numOfO = numOfX = 0;
    for( j = 0;j < size; j++){
        if( board[i][j] == 1 ){
            numOfX ++;
        }else{
            numOfO ++;
        }
    }
    if( numOfO == size){
        result = 0;
    }else if(numOfX == size){
        result = 1;
    }
}
```

检查列

```
if( result == -1){
    for( j = 0; j < size && result == -1; j++){
        numOfO = numOfX = 0;
        for( i = 0;i < size;i++){
            if( board[i][j] == 1){
                numOfX ++;
            }else{
                numOfO ++;
            }
        }
        if( numOfO == size){
            result == 0;
        }else if( numOfX == size){
            reshult == 1;
        }
    }
}
```

检查对角线

```
numOfO = numOfX = 0;
for( i = 0; i < size; i++){
    if( borad[i][size-i-1] == 1){
        numOfX ++;
    }else{
        numOfO ++;
    }
}


if( numOfO == size){
    result = 0;
}else if(numOfX == size){
    result == 1;
}
```

第九周：指针

9.1指针

9.1-1取地址运算

运算符&

1. scanf(("%d",&i);里的&
2. 获得变量的地址，它的操作数必须是变量　int i;printf("%x",&i);
3. 地址的大小是否与int相同取决于编译器　int i;printf("%p",&i);

&不能取的地址

1. &不能对没有地址的东西取地址，需要有明确的变量
2. &(a+b)可以
3. &(a++)不行
4. &(++a)不行

试试这些&

1. 变量的地址　　被分配在相邻紧挨着的地方
2. 相邻的变量的地址
3. &的结果的sizeof
4. 数组的地址
5. 数组单元的地址
6. 相邻的数组单元的地址

9.1-2 指针

scanf

1. 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? 可以
2. scanf("%d",&i);
3. scanf()的原型应该是怎么样的？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

就是保存地址的变量

1. int i
2. int* p = &i;
3. int* p,q;
4. int *P,q;
5. 这个*不管是靠近int还是靠近p，最后实际指向的都是p

指针变量

变量的值是内存的地址

1. 普通变量的值是实际的值
2. 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

1. void f(int *p);
2. 在被调用的时候得到了某个变量的地址：int i = 0;f(&i);
3. 在函数里面可以通过这个指针访问外面的这个i

访问那个地址上的变量*

1. *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
2. 可以做右值也可以做左值
3. int k = *p; *p = k + 1;

*左值之所以叫左值

1. 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;
2. a[0] = 2; *p = 3;
3. 是特殊的值，所以叫做左值

指针的运算符&*

1.互相反作用

1. *&yptr* ->(&yptr) -> *(yptr的地址) ->得到那个地址上的变量 ->yptr
2. &*yptr* ->*&(*yptr) ->&(y) ->得到y的地址，也就是yptr ->yptr

传入地址

为什么 int i;scanf("%d",i);编译没有报错?

但是运行会报错，只是刚好凑巧你传进去的整数大小跟地址大小一样，编译会拿你传进去的整数当做地址去运行，将整数传到了不该传的地方去了，所以运行就一定会出错

指针应用场景一

交换两个变量的值

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

指针应用场景二

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

指针应用场景二b

1. 函数返回运算的状态，结果通过指针返回
2. 常见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例子)
3. 但是当任何数值都是有效的可能结果时,就得分开返回了
4. 后续的语言(c++，java)采用了异常机制来解决这个问题

指针最常见的错误

定义了指针变量，还没有指向任何变量，就开始使用指针了

9.1-3指针与数组

传入的数组成了什么?

```
int isPrime( int x;int knownPrimes[],int numberOfKnownPrimes)
{
    int ret = 1;
    int i;
    for( i = 0; i < numberOfknownPrimes;i++){
        if( x%knownPrimes[i] == 0){
            ret = 0;
            break;
        }
    }
    return ret;
}
```

函数参数表中的数组实际上是指针

1. sizeof(a) == sizeof(int*)
2. 但是可以用数组的运算符[]进行运算

数组参数

以下四种函数原型是等价的

1. int sum(int*ar,int n);
2. int sum(int*,int);
3. int sum(int*ar[],int n);
4. 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

1. int a[10];int*P = a; //无需用&取地址
2. 但是数组的单元表达的是变量，需要用&取地址
3. a == &a[0]

[]运算符可以对数组做，也可以对指针做：

1. p[0]<==>a[0]

*运算符可以对指针做，也可以对数组做：

1. *a = 25;

数组变量是const的指针，所以不能被赋值

1. int a[] <==> int *const a = ...

9.1-4 指针与const

指针可以是const，值也可以是const

指针是const

表示一旦得到某个变量的地址，不能再指向其他变量

1. int *const q = &i; //q是const q指向i这个事实不能被改变
2. *q = 26; //OK
3. q++;error

所指是const

表示不能通过这个指针去修改那个变量(并不能使得那个变量成为const)

1. const int*p = &i;
2. *p = 26; //error!*(p)是const
3. i = 26; //ok
4. p = &j; //ok

判断哪个被const了的标志是const在*的前面还是后面

1. int i;
2. const int* p1 = &i;
3. int const* p2 = &i;
4. int *const p3 = &i
5. 2跟3其实是一样的，指针所指向的东西不可被修改
6. 4则是表示指针不可被修改

转换

1. 总是可以把一个非const的值转化成const的

void f(const int* x);

int a = 15;

f(&a);//ok

const int b = a;

f(&b);//ok

b = a + 1;//error!

当要传递的参数的类型比地址大的时候，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的修改

const数组

1. const int a[] = {1,2,3,4,5,6,};
2. 数组变量已经是const的指针了，这里的const表明数组的每一个单元都是const int
3. 所以必须通过初始化进行赋值

保护数组值

因为把数组传入函数时传递的是地址，所以那个函数内部可以修改数组的值

为保护数组不被函数破坏，可以设置参数为const

1. int sum(const int a[],int length);

9.2指针运算

9.2-1 指针是可计算的

1+1=2?

1. 给一个指针加1表示要让指针指向下一个变量

int a[10];

int *P = a;

*(p+1)——>a[1];

1. 如果指针不是指向一片连续分配的空间，如数组，则这种运算没有意义

指针计算

这些算术运算可以对指针做：

1. 给指针加、减一个整数(+,+=,-,-=)
2. 递增递减(++/--)
3. 两个指针相减

*p++

1. 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
2. *的优先级虽然高，但是没有++(单目运算符)高
3. 常用于数组类的连续空间操作
4. 在某些CPU上，这可以直接被翻译成一条汇编指令

指针比较

1. <,<=,>,>=,!=都可以对指针做
2. 比较它们在内存中的地址
3. 数组中的单元的地址肯定是线性递增的

0地址

1. 当然你的内存中有0地址，但是0地址通常是个不能随便碰的地址
2. 所以你的指针不应该具有0值
3. 因此可以用0地址来表示特殊的事情：
4. 1. 返回的指针是无效的
   2. 指针没有被真正的初始化(先初始化为0)
5. NULL是一个预定定义的符号，表示0地址
6. 1. 有的编译器不愿意你用0地址来表示0地址

指针的类型

1. 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
2. 但是指向不同类型的指针是不能直接相互赋值的
3. 这是为了避免用错指针

指针的类型转换

1. void*表示不知道指向什么东西的指针
2. 1. 计算时与char*相同(但不相通)
3. 指针也可以转换类型
4. 1. int*p* = &*i*;*void*q = (void*)p;
5. 这并没有改变p所指向的变量的类型，而是让后人用不同的眼光通过p看他所指的变量
6. 1. 我不在当你时int啦，我认为你就是个void!

用指针来做什么?

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

9.2-2动态内存分配

输入数据

1. 如果输入数据时，先告诉个数，然后再输入，要记录每个数据
2. int *a* = *(int)*malloc(n*sizeof(int));

malloc

#include<stdlib.h>

void*malloc(size_t size);

1. 向malloc申请的空间是以字节为单位的
2. 返回的结果是void*，需要类型转换为自己需要的类型(比如int)
3. 1. (int*)malloc(n*sizeof(int))

没空间了?

1. 如果申请失败则返回0，或者叫做NULL
2. 可以自己测测看自己电脑系统能给多少空间

free()

1. 把申请得来的空间还给"系统"
2. 申请过的空间，最终都应该要还的
3. 1. 出来混的，迟早都是要还的
4. 只能还申请来的空间的首地址
5. free(0)?

free()常见问题

1. 申请了没free->长时间运行内存逐渐下降
2. 1. 新手：忘了
   2. 不够老的老手：找不动合适的free的时机
3. free过了再free
4. 地址变过了，直接去free

9.2-3 函数间传递指针

好的模式

1. 如果程序中要用到动态分配的内存，并且会在函数之间传递，不要让函数申请内存后返回给调用者
2. 因为十有八九调用者会忘了free，或找不到合适的时机来free
3. 好的模式是让调用者自己申请，传地址进函数，函数再返回这个地址出来

在同一个地方malloc和free

除非函数的作用就是分配空间，否则不要再函数中malloc然后传出去用

函数返回指针

1. 返回指针没问题,关键是谁的地址?
2. 1. 本地变量(包括参数)?函数离开后这些变量就不存在了，指针所指的是不能用的内存
   2. 传入的指针？没问题
   3. 动态申请的内存？没问题
   4. 全局变量->以后会解释

函数返回数组

1. 如果一个函数的返回类型是数组，那么它实际返回的也是数组的地址
2. 如果这个数组是这个函数的本地变量，那么回到调用函数那里，这个数组就不存在了
3. 所以只能返回(和返回指针是一样的)
4. 1. 传入的参数：实际就是在调用者那里
   2. 全局变量或者动态分配的内存

第十周：字符串

10.1-1字符数组

1. char word[] = {'H','e','l','l','o','!'};

这(指1)不是C语言的字符串，因为不能用字符串的方式做计算

1. char word[] = {'H','e','l','l','o','!','\0'};

word[0]	**H**
word[1]	e
word[2]	l
word[3]	l
word[4]	o
word[5]	!
word[6]	\0

字符串

1. 以0(整数0)结尾的一串字符
2. 1. 0或者'\0'是一样的，但是和'0'不同
3. 0标志字符串的结束，但它不是字符串的一部分
4. 1. 计算字符串长度的时候不包含这个0
5. 字符串以数组的形式存在，以数组或者指针的形式访问
6. 1. 更多的是以指针的形式

string.h里有很多处理字符串的函数

1. C语言的字符串是以字符数组的形态存在的

2. 1. 不能用运算符对字符串做运算
   2. 通过数组的方式可以遍历字符串
3. 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
4. 以及标准库提供了一系列字符串函数

10.1-2字符串变量

1. char *str = "Hello";
2. char word[] = "hello";
3. char line[10] = "Hello";结尾编辑器会自动补0，多占据一个位置
4. "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0"
5. 两个相邻的字符串常量会被自动连接起来

字符串常量

char* s = "Hello,world";

1. s是一个指针，初始化为指向一个字符串常量
2. 1. 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
   2. 但是试图对s所指的字符串做写入会导致严重的后果
   3. 如果有两处相同的地方，指针会同时指向同一处地方，所以指针必须是只读的
3. 如果需要修改字符串，应该用数组：
4. 1. char s[] = "Hello,world!";
   2. 这个数组跟指针的区别就是，指针指向某一处地方，而数组则表示就在我这里
   3. 会将放在不可写的"Hello，world!"数组内容拷贝到你的s那里去

当我们需要一个字符串的时候，指针还是数组?

1. char*str = "Hello";
2. char word[] = "Hello";
3. 数组：这个字符串在这里
4. 1. 作为本地变量空间自动被回收
5. 指针：这个字符串不知道在哪里
6. 1. 处理参数
   2. 动态分配空间
   3. 用在只需要只读的，不打算去往里面写入东西的。表达函数的参数。
7. 如果要构造一个字符串->数组
8. 如果要处理一个字符串->指针

char*是字符串?

1. 字符串可以表达为char*的形式
2. char*不一定是字符串
3. 1. 本意是指向字符的指针，可能指向的是字符的数组(就像int*一样)
   2. 只有当char*所指向的字符数组有结尾的0，才能说它所指的是字符串

10.1-3字符串输入输出

1. char*t = "title";
2. char*s;
3. s = t;
4. 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

1. char string[8];
2. scanf("%s",string);
3. printf("%s\n",string);
4. scanf读入一个单词(到空格、tab或回车为止)，但scanf这样是不安全的，因为不知道要读入内容的长度

安全的输入

1. char string[8];
2. scanf("%7s",string);
3. 在%s中间可以加入数字来让编译器知道需要限制在多少字符范围内(或者说最多允许读入的字符数量)，比如%7s，限制在7个字符范围(超出部分就不会读入了)

常见错误

1. char*string;
2. scanf("%s",string);
3. 以为char("%s".string);
4. 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
5. 1. 由于没有对string初始化为0，所以不一定每一次运行都出错(实际上这是错误的，指针用错了，没有指向一个确定的地方)

空字符串

1. char buffer[100] = "";
2. 1. 这是一个空的字符串，buffer[0] == '\0'
3. char buffer[] = "";
4. 1. 这个数组的长度只有1！所以后面放不下任何的字符串

10.1-4字符串数组以及程序参数

字符串数组

1. char **a
2. 1. a是一个指针，指向另一个指针，那个指针指向一个字符(串)
3. char a[][]
4. 1. 一个错误的二维数组，因为没有说明几列，所以会报错
   2. 可以修改成char *a[]，本质上就相当于a[0]--->char*

程序参数

1. int main(int argc,char const*argv[])
2. argv[0]是命令本身
3. 1. 当使用Unix的符号链接时,反应符号链接的名字

```
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int i;
    for( i = 0; i < argc; i++){
        printf("%d:%s\n",i,argv[i]);
    }

    return 0;
}
```

10.2-1单字符输入输出,用putchar和getchar

putchar

1. int putchar(int c);
2. 向标准输出写一个符号
3. 返回写了几个字符，EOF(-1)表示写失败

getchar

1. int getchar(void);
2. 从标准输入读入一个字符(跟scanf的区别是scanf可以一次性读入多个字符)
3. 返回类型是int是为了返回EOF(-1)
4. 1. window-->Ctrl-Z
   2. Unix-->Ctrl-D(返回EOF)
   3. Ctrl-C会将shell与实际上显示的的通道关闭掉了

```
#include <stdio.h>

int main(int argc,char const *argv[])
{
    int ch;

    while( (ch = getchar()) != EOF ){
        putchar(ch);
    }
    printf("EOF\n");

    return 0;
}
```

在我们输入的东西(在键盘上敲出来的东西被称为行编辑的工作)的时候，那些都会被暂时放在shell里(类似缓冲区域),当我们按下回车之后，才会发送到实际上显示的地方上

10.2-(2-6)字符串函数strlen

string.h

1. strlen
2. 1. size_t strlen(const char *s);
   2. 返回s的字符串长度（不包括结尾的0）
3. strcmp
4. 1. int strcmp(const char *s1,const char *s2);
   2. 比较两个字符串,返回: 0：s1==s2，1：s1>s2，-1：s1<s2
5. strcpy
6. 1. char*strcpy(char*restrict dst,const char *restrict src);
   2. 把src的字符串拷贝到dst
7. 1. 1. restrict表明src跟dst不重合
8. 1. 返回dst
9. 1. 1. 为了能链起代码
10. 1. 复制一个字符串
11. 1. 1. char*dst = (char*)malloc(strlen(src)+1); //之所以加一是因为结尾会自带\0，所以需要多一个位置
    2. strcpy(dst,src);
12. strcat
13. 1. char*strcat(char *restrict s1,const char *restrict s2);
    2. 把s2拷贝到s1的后面，接成一个长的字符串
    3. 返回s1
    4. s1必须具有足够的空间
14. strchr
15. 1. 字符串中找字符
    2. char*strchr(const char *s,int c);表示从左边找过来
    3. char*strrchr(const char*s,int c);表示从右边找过来
    4. 返回NULL则表示没有找到
    5. 如何寻找第二个？寻找第二个的方法：

p = strchr(p+1,'l');

printf("%s\n",p);

1. strstr
2. 1. 在字符串中寻找单个字符的
3. 1. 1. char* strstr(const char *s1,const char *s2);
4. 1. 在寻找的时候忽略大小写
5. 1. 1. char*strcasestr(const char *s1,const char *s2);

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

size_t mylen(const char* s)
{
    int cnt = 0;
    int idx = 0;
```

```
    while(s[idx] != '\0' ){
        idx++;
        cnt++;
    }
    return cnt;
}

int main(int argc,char const *argv[])
{
    char line[] = "Hello";
    printf("strlen=%lu\n",mylen(line));
    printf("sizeof=%lu\n",sizeof(line));

    return 0;
}
#include <stdio.h>
#include <string.h>

int mycmp( const char* s1, const char* s2)
{
    //int idx = 0;
    //while( s1[idx] == s2[idx] && s1[idx]!='\0' ){
//  idx ++;
//}
    while( *s1 == *s2 && *s1 != '\0'){
        s1++;
        s2++;
    }
    return *s1 - *s2;
}

int main(int argc, char const *argv[])
{
    char s1[] = "abc";
    char s2[] = "abc";
    printf("%d\n",mycmp(s1,s2));
    printf("%d\n",'a','A');

    return 0;
}
#include <stdio.h>
#include <string.h>

char* mycpy(char* dst, const char* src)
{
    int idx = 0;
    while(src[idx] != "\0"){
        dst[idx] == src[idx];
        idx++;
    }
    dst[idx] = '\0';
 //   char* ret = dst;
 //   方法1：while(*src != '\0'){
 //       *dst++ = *src++;
 //   }
```

```
    //方法2：while(*dst++ = *src++);    嗯，没了，就一行直接替换掉了方法1，还有比方法1代码还有更长的版本我没有记录
 // *dst = '\0'; 这是指针的写法
    return dst;
}

int main(int argc, char const *argv[])
{
    char s1[] = "abc";
    char s2[] = "abc";
    strcpy(s1,s2);

    return 0;
}
```

安全问题

1. strcpy跟strcat都可能出现安全问题
2. 1. 如果目的地没有足够的空间?
   2. 建议是尽量不要去使用他
3. 安全版本
4. 1. char* strncpy(char*restrict dst, const char* restrict src,size_t n);
   2. char* strncat(char*restrict s1, const char* restrict s2,size_t n);
   3. size_t n表示最多能够接受多少个n个字符，多了就直接掐掉
   4. int strncmp(const char *s1,const char *s2,size_t n);
   5. 这个则表示最多能够判断几个字符，超出则不判断

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

int main(int argc, char const *argv)
{
    char s[] = "hello";
    char *p = strchr(s,'l');

    printf("%s\n",p);  //结果为llo
    //寻找第二个的方法：
    //p = strchr(p+1,'l');
    //printf("%s\n",p);结果为lo

    return 0;
}

-----------------------------------------------------
   将起选取的内容拷贝到其他地方的方法
    int main(int argc, char const *argv)
{
    char s[] = "hello";
    char *p = strchr(s,'l');
    char *t = (char*)malloc(strlen(p)+1);
    strcpy(t,p);
    printf("%s\n",t);
    free(t); // 申请来的空间记得释放掉哦
```

```
    //这是将llo的字符拷贝走了
    return 0;
}
-------------------------------------------------------
      将起选取的内容拷贝到其他地方的方法2版本
    int main(int argc, char const *argv)
{
    char s[] = "hello";
    char *p = strchr(s,'l');
    char c = *p;
    *p = '\0';
    char *t = (char*)malloc(strlen(s)+1);
    strcpy(t,s);
    printf("%s\n",t);
    free(t); // 申请来的空间记得释放掉哦
    //这是将he的字符拷贝走了
    return 0;
}
```

第十一周：结构类型

11.1-1枚举

常量符号化

1. 用符号而不是具体的数字来表示程序中的数字

```
#include<stdio.h>

const int red = 0;
const int yellow = 1;
const int green = 2;

int main(int argc, char const *argv[])
{
    int color = -1;
    char *colorName = NULL;

    printf("请输入你喜欢的颜色的代码:");
    scanf("%d",&color);
    switch( color ){
            case red:colorName = "red";break;
            case yellow:colorName = "yellow";break;
            case green:colorName = "green";break;
            default:colorName = "default";break;
    }
    printf("你喜欢的颜色是%d",colorName);

    return 0;
}
```

枚举

1. 用枚举而不是定义独立的const int变量

```
#include<stdio.h>

enum COLOR{RED,YELLOW,GREEN}; //枚举

int main(int argc, char const *argv[])
{
    int color = -1;
    char *colorName = NULL;

    printf("请输入你喜欢的颜色的代码:");
    scanf("%d",&color);
    switch( color ){
            case RED:colorName = "red";break;
            case YELLOW:colorName = "yellow";break;
            case GREEN:colorName = "green";break;
            default:colorName = "default";break;
    }
    printf("你喜欢的颜色是%d",colorName);

    return 0;
}
```

1. 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：
2.
    1. enum 枚举类型名字{名字0,.....,名字n};
    2. 枚举类型名字可以省略
3. 枚举类型名字通常并不真的使用，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：
4.
    1. enum colors{red,yellow,green};
    2. 这样就创建了3个常量，red的值是0,yellow的值是1,而green的值是2
    3. 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名字

```
#include<stdio.h>

enum color{ red,yellow,green};

void f(enum color c);

int main(void)
{
    enum color t = red;

    scanf("%d",&t);
    f(t);

    return 0;
}

void f(enum color c)
{
```

```
    printf("%d\n",c);
}
```

1. 枚举量可以作为值
2. 枚举类型可以跟上enum作为类型
3. 但是实际上是以整数来做内部计算和外部输入输出的

```
#include<stdio.h>

enum COLOR{RED,YELLOW,GREEN,NumCOLOR};

int main(int argc, char const *argv[])
{
    int color = -1;
    char *ColorNames[NumCOLOR] = {"red","yellow","green",};
    char *colorName = NULL;

    printf("请输入你喜欢的颜色的代码:");
    scanf("%d",&color);
    if( color >= 0 && color < NumCOLORS ){
        colorName = ColorNames[color];
    }else{
        colorName = "unknown";
    }
    printf("你喜欢的颜色是%s",colorName);

    return 0;
}
```

1. 这样需要遍历所有的枚举量或者需要建立一个用枚举量做下标的数组的时候就很方便
2. 上面的套路：在进行枚举的时候最后面在放上一个数(NumCOLORS)，这样就能够表示NumCOLORS前面有几个数了(例如里面有3个数，索引值到0-2，在后面加上一个数，索引值刚好等于实际我们想要表达的数量)

枚举量

1. 声明变量可以指定值
2. 1. enum COLOR{RED = 1,YELLOW,GREEN = 5};

```
#include<stdio.h>

enum COLOR {RED = 1; YELLOW,GREEN=5,NumberCOLORS};

int main(int argc,char const *argv[])
{
    printf("code for GREEN is %d\n",GREEN);

    return 0;
}
//这样输出的话，会从1开始计数，YELLOW则变成1+1，然后到GREEN的5之前都会跳过了
```

枚举只是int

1. 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

枚举

1. 虽然枚举类型可以当作类型使用，但是实际上很(bu)少(hao)用
2. 如果有意义上排比的名字，用枚举比const int方便
3. 枚举比宏(macro)好，因为枚举有int类型，宏没有类型(宏我在后面会解释是啥)

11.2-1结构类型

声明结构类型

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    struct date{
        int month;
        int day;
        int year;
    };//声明在这里，最后要加上分号哦，这是在函数内部声明的，通常放在函数外面

    struct date today;//在这里我们定义了一个变量是today，类型是struct date的

    today.month = 07;
    today.day = 31;
    today.year = 2014;

    printf("Today's date is %i-%i-%i.\n",today.year,today.month,today.day);

    return 0;
}

//声明结构类型跟定义结构变量是两件事情哦
```

1. 声明在函数内还是函数外?
2. 1. 和本地变量一样(就是局部变量)，在函数内部声明的结构类型只能在函数内部使用
   2. 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

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

struct point p1,p2;

p1和p2都是point
里面有x和y值 //这是第一种声明方式

-----------------------------------------------
```

```
struct{
    int x;
    int y;
}p1,p2;
p1和p2都是一种无名结构，里面有x和y   //这是第二种形式，没有名字(没有声明point)
    //只是定义了两个变量，因为作者并不打算接下来继续在其他地方去调用

-----------------------------------------------------
struct point{
    int x;
    int y;
}p1,p2;
p1和p2都是point，里面有x和y的值t   //这是第三种声明方式
```

结构的初始化

```
#include<stdio.h>

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

int main(int argc,char const *argv[])
{
    struct date today = {07,31,2014};
    struct date thismonth = {.month = 7,.year = 2014};

    printf("Today's date is %i-%i-%i.\n",today.year,today.month,today.day);
    printf("This month is %i-%i-
%i.\n",thismonth.year,thismonth.month,thismonth.day);
    //给的值会被填进去，没给的值跟数组一样默认为0
    return 0;
}
```

结构成员

1. 结构和数组有点像
2. 数组用[]运算符和下标访问其成员
3. 1. a[0] = 10;
4. 结构用.运算符和其名字访问其成员
5. 1. today.day
   2. student.firstName
   3. p1.x
   4. p2.y

结构运算

1. 要访问整个结构，直接用结构变量的名字
2. 对于整个结构，可以做赋值、取地址，也可以传递给函数参数
3. p1 = (struct point){5,10}; //相当于p1.x = 5;p1.y = 10;
4. p1 = p2; //相当于p1.x = p2.x;p1.y = p2.y;
5. 数组无法做这两种运算！但结构可以

结构指针

1. 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
2. struct date *pDate = &today;

```
#include<stdio.h>

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

int main(int argc,char const *argv[])
{
    struct date today;

    today = (struct date){07,31,2014};

    struct date day;

    struct date *pDate = &today;  //指向地址

    printf("Today's date is %i-%i-%i.\n",today.year,today.month,today.day);
    printf("The day's date is %i-%i-%i.\n",day.year,day.month,day.day);

    printf("address of today is %p\n",pDate);

    return 0;
}
```

11.2-2结构与函数

结构作为函数参数

int numberOfDays(struct date d)

1. 整个结构可以作为参数的值传入函数
2. 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
3. 也可以返回一个结构
4. 跟数组完全不一样

```
#include<stdio.h>
#include<stdbool.h>
struct date{
    int month;
```

```c
    int day;
    int year;
};

bool isLeap(struct date d);
int numberOfDays(struct date d);

int main(int argc,char const *argv[]){
    struct date today,tomorrow;
    //输入今天的日期，月 日 年
    printf("Enter today's date (mm dd yyyy):");
    scanf("%i %i %i",&today.month,&today.day,&today.year);

    if( today.day != numberOfDays(today)){
        tomorrow.day = today.day+1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }else if( today.month == 12 ){
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year+1;
    }else{
        tomorrow.day = 1;
        tomorrow.month = today.month+1;
        tomorrow.year = today.year;
    }

    printf("Tomorrow's date is %i-%i-%i.\n",
    tomorrow.year,tomorrow.month,tomorrow.day);

    return 0;
}

int numberOfDays(struct date d){
    int days;

    const int daysPerMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

        if(d.month == 2 && isLeap(d))
            days = 29;
        else
            days = daysPerMonth[d.month-1];

    return days;
}

bool isLeap(struct date d){
    bool leap = false;

    if((d.year %4 == 0 && d.year %100 != 0) || d.year%400 == 0 )
        leap = true;

    return leap;
}
```

输入结构

1. 没有直接的方式可以一次scanf——一个结构
2. 如果我们打算写一个函数来读入结构
3. 但是读入的结构如何送回来呢?
4. C语言在函数调用时是传值的
5. 在函数读入了p的数值之后，没有任何东西回到main，所以y还是{0,0}
6. 1. 解决方案
   2. 之前的方案，把一个结构传入了函数，然后在函数中操作，但是没有返回回去
7. 1. 1. 问题在于传入函数的是外面那个结构的克隆体，而不是指针
      2. 传入结构和传入数组是不同的
8. 1. 在这个输入函数中，完全可以创建一个临时的结构变量，然后把这个结构返回给调用者

```
#include<stdio.h>

struct point {
    int x;
    int y;
};

void getStruct(struct point);
void output(struct point);

int main(int argc,char const *argv[])
{
    struct point y = {0,0};
    getStruct(y);
    output(y);
}

void getStruct(struct point p)
{
    scanf("%d",&p.x);
    scanf("%d",&p.y);
    printf("%d,%d",p.x,p.y);
}

void output(struct point p)
{
    printf("%d,%d",p.x,p.y);
}
```

结构指针作为参数

1. K&R说过(p.131)

指向结构的指针

用->表示指针所指的结构变量中的成员

```
struct date{
    int month;
```

```
    int day;
    int year;
}myday;

struct date *p = &myday;

(*p).month = 12;
p->month = 12;
//第九行跟第十行是一样的意思，第十行会更便捷
#include<stdio.h>

struct point {
    int x;
    int y;
};

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

int main(int argc,char const *argv[])
{
    struct point y = {0,0};
    getStruct(&y);
    output(y);
    output(*getStruct(&y));
    print(getStruct(&y));
    *getStruct(&y) = (struct point){1,2};
}

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

void output(struct point p)
{
    printf("%d,%d",p.x,p.y);
}

void print(const struct point *p);
{
    printf("%d,%d",p->x,p->y);
}
```

11.2-3结构中的结构

结构数组

1. struct date dates[100]; //这是在初始化数组
2. struct date dates[] = {{4,5,2005},{2,4,2005}};

```
#include<stdio.h>
struct time{
    int hour;
    int minutes;
    int seconds;
};

struct time timeUpdate(struct time now);

int main(void){
    struct time testTimes[5] = {
        {11,59,59},{12,0,0},{1,29,59},{23,59,59},{19,12,27}
    };
    int i;

    for(i=0; i<5; ++i){
        printf("Time is %.2i:%.2i:%.2i",
               testTimes[i].hour,testTimes[i].minutes,testTimes[i].seconds);

        testTimes[i] = timeUpdate(testTimes[i]);

        printf("...one second later it's %.2i: %.2i: %.2i\n",
               testTimes[i].hour,testTimes[i].minutes,testTimes[i].seconds);
    }

    return 0;
}

struct time timeUpdate(struct time now){
    ++now.seconds;
    if(now.seconds == 60 ){
        now.seconds = 0;
        ++now.minutes;

        if(now.minutes == 60 ){
            now.minutes = 0;
            ++now.hour;

            if(now.hour == 24 ){
                now.hour = 0;
            }
        }
    }
}
struct dateAndTime{
    struct date sdate;
    struct time stime;
};
```

嵌套的结构

```
struct point{
    int x;
    int y;
};
struct rectangle{
    struct point pt1;
    struct point pt2;
};

如果有变量 struct rectangle r;
就可以有：
    r.pt1.x、r.ptl.y
    r.pt2.x、r.pt2.y

如果有变量定义：
    struct rectangle r,*rp;
    rp = &r;

那么下面的四种形式是等价的:(结构中的结构)
    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)
```

结构中的结构的数组

```
#include<stdio.h>

struct point{
    int x;
    int y;
};

struct rectangle{
    struct point p1;
    struct point p2;
};

void printRect(struct rectangle r)
{
printf("<%d,%d> to <%d,%d>\n",r.p1.x,r.p1.y,r.p2.x,r.p2.y);
}

int main(int argc,char const *argv[])
{
    int i;
    struct rectangle rects[] = {{{1,2},{3.4}},{{5,6},{7,8}}};//2 rectangles
    for(i = 0;i < 2;i++)printRect(rects[i]);
}
```

11.3-1类型定义

自定义数据类型(typedef)

1. 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
2. 比如：typedef int Length;
3. 使得Length成为int类型的别名
4. 这样,Length这个名字就可以替代int出现在变量定义和参数声明的地方了:
5. 1. Length a,b,len;
   2. Length numbers[10];

Typedef

1. 声明新的类型的名字
2. 1. 新的名字是某种类型的别名
   2. 改善了程序的可读性

```
typedef long int64_t;   //重载已有的类型名字  新名字的含义更清晰  具有移植性
typedef struct ADate{
    int month;
    int day;
    int year;
}Date; //简化了复杂的名字

//在这里Date等价于struct ADate，Date代表了到达struct ADate之前的所有

int64_t i = 10000000000;
Date d = {9,1,2005};
typedef int Length;//Length就等价于int类型

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

typedef struct node{
    int data;
    struct node*next;
}aNode;

或

typedef struct node aNode;//这样用aNode就可以替代struct node
```

11.3-2联合

联合

1. 存储
2. 1. 所有的成员共享一个空间
   2. 同一时间只有一个成员是有效的
   3. union的大小是其最大的成员
3. 初始化
4. 1. 对第一个成员做初始化

```c
#include<stdio.h>

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

int main(int argc,char const argv[])
{
    CHI chi;
    int i;
    chi.i = 1234;
    for( i = 0;i < sizeof(int);i++){
        printf("%02hhX",chi.ch[i]);
    }
    printf("\n");

    return 0;
}

//输出D20400，            1234的十六进制是0x04D2
//低位在前(相当于倒置)，小端的方式
```

第十二周：程序结构

12.1-1全局变量：定义在函数之外的变量，全局的生存期和作用域

全局变量

1. 定义在函数外面的变量是全局变量
2. 全局变量具有全局的生存期和作用域
3. 1. 他们和任何函数无关
   2. 在任何函数内部都可以使用他们

```c
#include<stdio.h>

int f(void);

int gAll = 12;//全局变量

int main(int argc,char const *argv[])
{
    printf("in %s gAll=%d\n",__func__,gAll);
    f();
    printf("agn in %s gAll = %d\n",__func__,gAll);
    return 0;
}
//_func_是一个字符串，表达的是当前函数的名字，也就是main的名字
int f(void)
{
    printf("in %s gAll=%d\n",__func__,gAll);
```

```
    gAll += 2;
    printf("agn in %s gAll=%d\n",__func__,gAll);
    return gAll;
}
```

全局变量初始化

1. 没有做初始化的全局变量会得到0值
2. 1. 指针会得到NULL值
3. 只能用编译时刻已知的值来初始化全局变量
4. 它们的初始化发生在main函数之前

被隐藏的全局变量

1. 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(局部的优先度会更高噢，会覆盖掉外面的变量)

12.1-2静态本地变量：能在函数结束后继续保有原值的本地变量

静态本地变量

1. 在本地变量定义时加上static修饰符就成为静态本地变量
2. 当函数离开的时候，静态本地变量会继续存在并保持其值
3. 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

```
#include<stdio.h>

int f(void);

int gAll = 12;//全局变量

int main(int argc,char const *argv[])
{
    f();
    f();
    f();
    return 0;
}
//_func_是一个字符串，表达的是当前函数的名字，也就是main的名字
int f(void)
{
    static int all = 1;
    printf("in %s gAll=%d\n",_func_,gAll);
    gAll += 2;
    printf("agn in %s gAll=%d\n",_func_,gAll);
    return gAll;
}
```

1. 静态本地变量实际上是特殊的全局变量
2. 它们(静态本地变量和全局变量)位于相同的内存区域
3. 静态本地变量具有全局的生存期，函数内的局部作用域

4. static 在这里的意思是局部作用域(本地可访问)

12.1-3后记：返回指针的函数，使用全局变量的贴士

*返回指针的函数

1. 返回本地变量的地址是危险的
2. 返回全局变量或静态本地变量的地址是安全的
3. 返回在函数内malloc的内存是安全的，但是容易造成问题
4. 最好的做法是返回传入的指针

tips(贴士)

1. 不要使用全局变量来在函数间传递参数和结果
2. 尽量避免使用全局变量
3. 1. 丰田汽车的案子(ks) 这里给出传送门(想知道的可以看看)：https://www.sohu.com/a/133455549_464086
4. *使用全局变量和静态本地变量的函数是线程不安全的

12.2-1宏定义

编译预处理指令

1. #开头的是编译预处理指令
2. 它们不是c语言的成分，但是C语言程序离不开它们
3. #define用来定义一个宏 ，其实只是一个原始的文本替换

#define

1. #define<名字><值>
2. 注意没有结尾的分号，因为不是c的语句
3. 名字必须是一个单词，值可以是各种东西
4. 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
5. 1. 完全的文本替换
6. gcc——save-temps

宏

1. 如果一个宏的值中有其他宏的名字，也是会被替换的
2. 如果一个宏的值超过来一行，最后一行之前的行末需要加\
3. 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

1. #define_DEBUG
2. 这里宏是用于条件编译的，后面有其他的编译预处理指令来检查这个宏是否已经被定义过了

预定义的宏

1. *LINE*表达代码所在的行号
2. *FILE*表达代码所在的文件名
3. *DATE*编译时候的日期
4. *TIME*编译时候的时间
5. *STDC*

12.2-2带参数的宏

像函数的宏

1. #define cube(x)((x)*(x)*(x))
2. 宏可以带参数

```
#include <stdio.h>

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

int main(int argc,char const *argv[])
{
    printf("%d\n",cube(5));
    //cube(5)会被替换成cube((5)*(5)*(5));

    return 0;
}
```

错误定义的宏

1. #define RADTODEG(x)(x*57.29578)
2. #define RADTODEG(x)(x)*57.29578

带参数的宏的原则

1. 一切都要括号
2.
    1. 整个值要括号
    2. 参数出现的每个地方都要括号
3. #define RADTODEG(x)((x)*57.29578)

带参数的宏

1. 可以带多个参数
2.
    1. #define MIN(a,b)((a)>(b)?(b):(a))
3. 也可以组合(嵌套)使用其他宏
4. 定义宏的时候后面千万不要加分号
5. 在大型程序的代码中使用非常普遍
6. 可以非常复杂，如"产生"函数
7.
    1. 在#和##这两个运算符的帮助下
8. 存在中西方文化差异
9. 部分宏会被inline函数替代

宏的缺陷

没有可以去检查宏有没有问题的机制

其他预编译

1. 条件编译
2. error
3. ...

12.3-1多个源代码文件

多个.c文件

1. main()里的代码太长了适合分成几个函数
2. 一个源代码文件太长了适合分成几个文件
3. 两个独立的源代码文件不能编译形成可执行的程序

项目

因为看的是翁恺老师2014年的版本(比后来的课程内容多不少，后来的课程缩水了)，所以这部分已经有更好的就进行省略了

编译单元

1. 一个.c文件是一个编译单元
2. 编译器每次编译只处理一个编译单元

12.3-2头文件

1. 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

1. #include是一个编译预处理指令,和宏一样，在编译之前就处理了
2. 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
3. 所以也不是一定要在.c文件的最前面#include

“”还是<>

1. #include有两种形式来指出要插入的文件
2.
    1. “”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找，例如“MAX.h”啥的自己设定的那些
    2. <>让编译器只在指定的目录去找，例如<stdio.h>
3. 编译器自己知道自己的标准库的头文件在哪里
4. 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

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

头文件

1. 在使用和定义这个函数的地方都应该%include这个头文件
2. 一般的做法就是任何.c都有对应的同名的.h，把所有对外公开的函数的原型和全局变量的声明都放进去
3. 全局变量是可以在多个.c之间共享的

不对外公开的函数

1. 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
2. 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了

12.3-3声明

变量的声明

1. int i;是变量的定义
2. extern int i;是变量的声明

声明和定义

1. 声明是不产生代码的东西
2. 1. 函数原型
   2. 变量声明
   3. 结构声明
   4. 宏声明
   5. 枚举声明
   6. 类型声明
   7. inline函数
3. 定义是产生代码的东西

头文件

1. 只有声明可以被放在头文件中
2. 1. 是规则不是法律
3. 否则会造成一个项目中多个编译单元里有重名的实体
4. 1. *某些编译器允许几个编译单元中存在同名的函数，或者用week修饰符来强调这种存在

重复声明

1. 同一个编译单元里，同名的结构不能被重复声明
2. 如果你的头文件里有结构的声明，很难这个头文件不会在一个编译单元里被#include多次所以需要"标准头文件"

标准头文件结构

1. 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
2. #pragma once也能起到相同的作用，但是不是所有的编译器都支持

```
#ifndef __LIST_HEAD__
#define __LIST_HEAD__

#include"node.h"

typedef struct _list{
    Node*head;
    Node*tail;
}List;

#endif
```

第13周：文件

13.1-1格式化输入输出格式

%[flags][width][.prec][hIL]type

Flag	含义
-	左对齐
+	在前面放+或-
(space)	正数留空
0	0填充

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    printf("%9d\n",123);//需要占9个字符的空间，123前还有6个空格
    printf("%-9d\n",123);//一样需要占9个字符,123后面还有6个空格
    printf("%+9d\n",123);//在123前面多上一个+，是可以0时跟左对齐同时进行的
    printf("%09d\n",123);//空格地方变成0，000000123

    return 0;
}
```

width或prec	含义
number	最小字符数
*	下一个参数是字符数
.number	小数点后的位数
.*	下一个参数是小数点后的位数

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    printf("%9.2f\n",123.0);//占据9个字符且其中有两位小数
    printf("%*d\n",6,123);//让格式更灵活，6这个数是会替换到*号上的，具体效果就是占据6个字符

    return 0;
}
```

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

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    printf("%hhd\n",12345);//只能输入单个字符，输出多个报错
    printf("%hhd\n",(char)12345);//但可以强制类型转换，可以得到57的结果
    printf("%9.2f\n",123.0);

    return 0;
}
```

type	用于	type	用于
i或者d	int	g	float
u	unsigned int	G	float
o	八进制	a或者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

type	用于	type	用于
f或者F	float，6	p	指针
e或者E	指数	n	输入/写出的个数

```c
#include<stdio.h>

int main(int argc,char const *argv[])
{
    int num;
    printf("hhd%n\n",(char)12345,&num);
    printf(num);//以上的意思是，截至到%n为止前面有几位字符，会将多少位字符传递给指针&num的地址
    //然后在下面num就会输出2，因为上面(char)12345的值输出是57，是2位数

    return 0;
}
```

scanf：%[flag]type

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

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可以将读入的八进制和十六进制格式的数值转化为十进制	[…]	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		

printf和scanf的返回值

1. 读入的项目数
2. 输出的字符数
3. 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

13.1-3文件输入输出

文件输入输出

1. 用>和<做重定向

FILE

1. FILE*fopen(const char**restrict path,const char*restrict mode*);
2. int fcolose(FILE*stream);
3. fscanf(FILE,...)
4. fprintf(FILE*,...)
5. fprintf(FILE,...)

打开文件的标准代码

```
FILE*fp = fopen(“file(文件名)”,“r”);

if(fp){
    fscanf(fp,...);
    fcolose(fp);
}else{
    ...
}
```

```
    #include<stdio.h>

    int main(int argc,char const *argv[])
{
    FILE *fp = fopen("12.in","r");
    if( fp ){
        int num;
        fscanf(fp,"%d",&num);
        printf("%d\n",num);
        fclose(fp);
    }else{
        printf("无法打开文件\n");
    }
    return 0;
}
```

fopen

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	只新建，如果文件已存在则不能打开(防止对文件造成破坏)

13.1-3二进制文件

1. 其实所有的文件最终都是二进制的
2. 文件无非是用最简单的方式可以读写的文件
3. 1. more、tail
   2. cat
   3. vi
4. 而二进制文件是需要专门的程序来读写的文件
5. 文本文件的输入输出是格式化，可能经过转码

文本文件 VS 二进制文件

1. Unix喜欢用文本文件来做数据存储和程序配置
2. 1. 交互式终端的出现使得人们夏欢用文本和计算机“talk”
   2. Unix的shell提供了一些读写文本的小程序
3. windows喜欢二进制文件
4. 1. DOS是草根文化，并不继承和熟悉Unix文化
   2. PC刚开始的时候能力有限，DOS的能力更有限，二进制进行输入输出更接近底层

优劣：

1. 文本的优势是方便人类读写，而且跨平台
2. 文本的缺点是程序输入输出要经过格式化，开销大
3. 二进制的缺点是人类读写困难，而且不跨平台
4. 1. int的大小不一致，大小端的问题
5. 二进制的优点是程序读写快

程序为什么要文件

1. 配置
2. 1. Unix用文本，Windows用注册表
3. 数据
4. 1. 稍微有点量的数据都放数据库了
5. 媒体
6. 1. 这个只能是二进制的

7. 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

二进制读写(可跳过，底层)

1. size_t fread(void*restrict ptr*,*size_t size*,*size_t nitems*,*FILE*restrict stream);
2. size_t fwrite(const void*restruct ptr*,*size_t size*,*size_t nitems*,*FILE*restrict stream);
3. 注意FILE指针是最后一个参数
4. 返回的是成功读写的字节数

为什么nitems

1. 因为二进制文件的读写一般都是通过对一个结构变量的操作来进行的
2. 于是nitem就是用于说明这次读写几个结构变量!

在文件中定位

1. long ftell(FILE*stream);
2. int fseek(FILE*stream,long offset,int whence);
3. 1. SEEK_SET：从头开始
   2. SEEK_CUR：从当前位置开始
   3. SEEK_END：从尾开始(倒过来)

```
#include<stdio.h>
#include"student.h"

void read(FILE*fp,int index);

int main(int grac,char const grav[])
{
    FILE*fp = fopen("student.data","r");
    if( fp ) {
        fseek(fp,0L,SEEK_END);
        long size = ftell(fp);
        int number = size/sizeof(Student);
        int index = 0;
        printf("有%d个数据,你要看第几个:",number);
        scanf("%d",&index);
        read(fp,index-1);
        fcolose(fp);
    }
    return 0;
}

void read(FILE*fp,int index)
{
    fseek(fp,index*sizeof(Student),SEEK_SET);
    Student stu;
    if( fread(&stu,sizeof(Student),1,fp) == 1){
        printf("第%d个学生:",index+1);
        printf("\t姓名:%s\n",stu.name);
        printf("\t性别:");
        switch ( stu.gender ){
            case 0:printf("男\n");break;
            case 1:printf("女\n");break;
```

```
            case 1:printf("其他\n");break;
        }
        printf("\t年龄:%d\n",stu.age);
    }
}
```

可移值性

1. 这样的二进制文件不具有可移植性
2. 1. 在int为32位的机器上写成的数据文件无法直接在int为64的机器上正确读出
3. 解决方法之一就是放弃使用int,而是使用typedef具有明确大小的类型
4. 更好的方案是用文本

13.2*位运算

13.2-1按位运算

1. C有这些按位运算的运算符：其实就是把整数的东西当做二进制来进行运算

·&	按位的与
·|	按位的或
·~	按位取反
·^	按位的异或
·<<	左移
·>>	右移

按位与&

1. 其实就是两组二进制的数，对应的数字必须都为1，新形成的数对应的数才会是1，否则就是0
2. F:1111 E:1110，FE的作用就是使得跟他对应形成新的数最低位为0

按位或|

1. 也是两组二进制的数，对应的数字必须都为0，新形成的数对应的数才会是0，否则就是1。跟上面那个相反

按位取反~

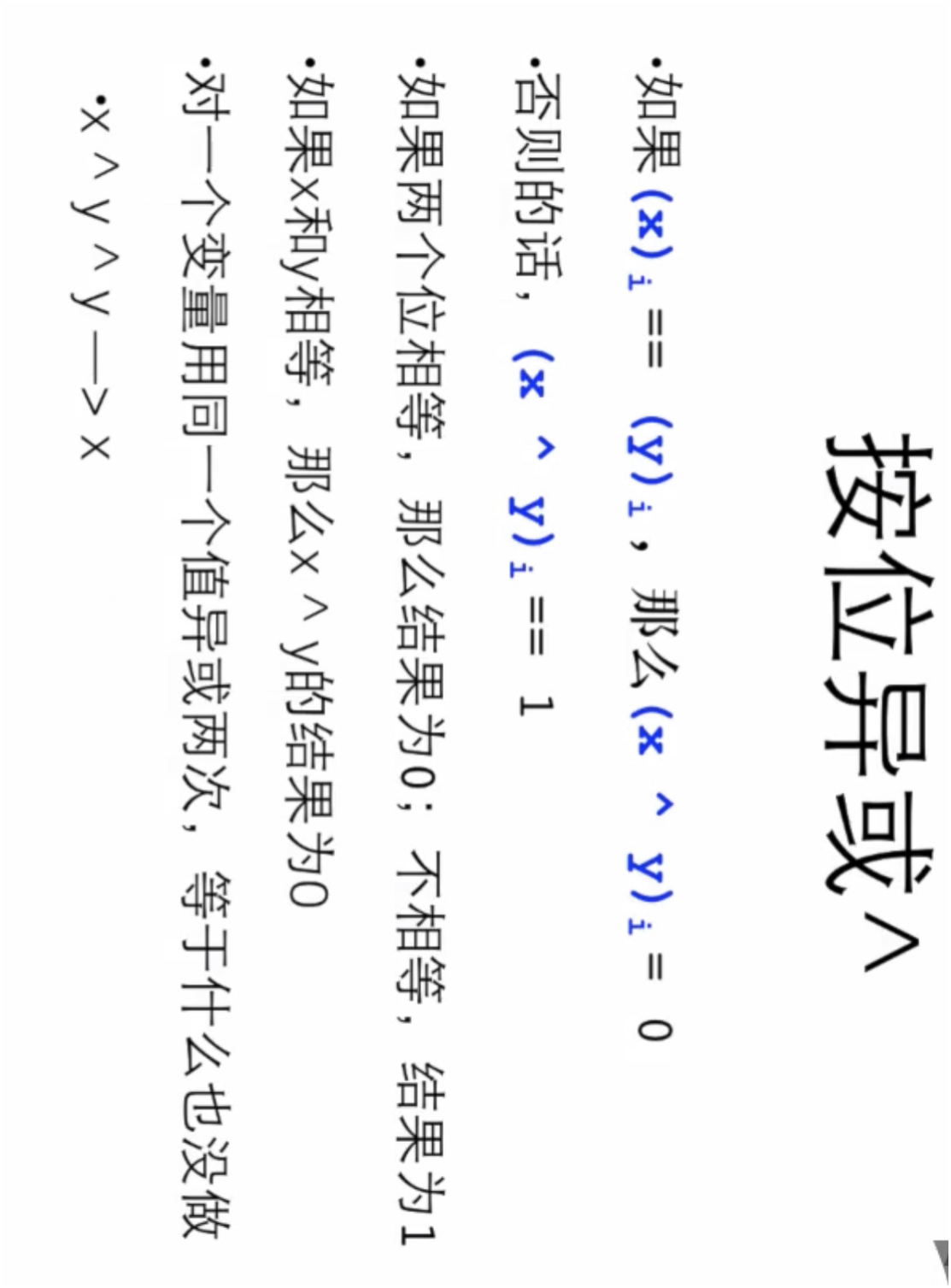
```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    unsigned char c = 0xAA;
    printf("% c=%hhx\n",c);//aa
    printf("%~c=%hhx\n",(char)~c);//按位取反   55
    printf("%-c=%hhx\n",(char)-c);//补码    56
}
```

逻辑运算vs按位运算

1. 对于逻辑运算，它只看到两个值：0和1
2. 可以认为逻辑运算相当于把所有非0值都变成1，然后做按位运算
3. 1. 5&4——>4而5&&4——>1&1——>1
   2. 5|4——>5而5||4——>1|1——>1

3. ~4——>3而!4——>!1——>0

按位异或^



1. 两组二进制，上下位数值对应相等为0，上下不相等为1
2. 做两次相同的异或运算数值就翻回去了

移位运算

左移<<

1. i << j
2. i中所有的位向左移动j个位置，而右边填入0
3. 所有小于int的类型，移位以int的方式来做，结果是int
4. x <<= 1 等价于x*=2
5. x <<= n 等价于x*=2的n次方

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    unsigned char c = 0xA5;
    printf("  c=%d\n",c);//165
    printf("c<<=%d\n",c<<2);//660
    return 0;
}
```

右移>>

1. i >> j
2. 所有小于int的类型，移位以int的方式来做，结果是int
3. 对于unsigned的类型，左边填入0
4. 对于signed的类型，左边填入原来的最高位(保持符号不变)
5. x >>= 1 等价于x/=2
6. x >>= n 等价于x/=2的n次方

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    int a = 0x80000000;
    unsigned int b = 0x80000000;
    printf("a=%d\n",a);//-2147483648
    printf("b=%u\n",b);//2147483648
    printf("a>>1=%d\n",a>>1);//-1073741824
    printf("b>>1=%u\n",b>>1);//1073741824
    return 0;
}
```

no zuo no die

1. 移位的位数不要用负数，这是没有定义的行为
2. 1. x<<-2 //!!NO!!

13.2-3位运算例子

输出一个数的二进制

```
#include<stdio.h>

int main(int argc,char const *argv[])
{
    int number;
    scanf("%d",&number);
    number = 0x55555555;//输出01010101...，其实就是16个01(32个值)
    unsigned mask = 1u<<31;//int被省略但是其实是有生效的
    for(; mask ; mask >>=1 ){
        printf("%d",number & mask?1:0);
    }
    printf("\n");

    return 0;
}
```

13.2-4位段

1. 把一个int的若干位组合成一个结构

```
struct{
    unsigned int leading : 3;//冒号后面的数字表示占几个比特

    unsigned int FLAG1: 1;

    unsigned int FLAG2: 1;

    int trailing:11;
};
```

1. 可以直接用位段的成员名称来访问
2. 1. 比移位、与、或还方便
3. 编译器会安排其中的位的排列，不具有可移植性
4. 当所需的位超过一个int时会采用多个int

第十四周：*链表

14.1-1*可变数组

the Interface

1. Array array_create(int init_size);创建数组
2. void array_free(Array*a);回收数组
3. int array_size(const Array*a);告诉我们数组里面现在有几个单元可以使用
4. int*array_at(Array*a*,int index);访问数组某个单元
5. void array_inflate(Array*a,int more_size);让数组

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
```

```
typedef struct {
    int *array;
    int size;
}Array;

Array array_create(int init_size);
void array_free(Array *a);
int array_size(const Array*a);
4. int*array_at(Array*a,int index);
5. void array_inflate(Array*a,int more_size);

#endif
#include "array.h"

Array array_create(int init_size)
{
    Array a;//首先是数组的创建，需要用到动态内存分配，结合所需要的size，来创建一个数组
    a.size = init_size;
    a.array = (int*)malloc(sizeof(int)*a.size);
    return a;
}
void array_free(Array *a)//再然后就是内存的释放、得到数组的大小
{
    free(a->array);
    a->array = NULL;
    a->size = 0;
}
 int array_size(const Array*a);
 int*array_at(Array*a,int index);
 void array_inflate(Array*a,int more_size);

int main(int argc,char const *argv[])
{
    Array a = array_create(100);

    return 0;
}
```

14.1-2可变数组的数据访问

```
//从上述第16行内容延续，这里需要多加两个标准头文件
#include<stdio.h>
#include<stdlib.h>
//3-6行的代码叫做封装,能够将里面的内容保护起来，这样别人就不知道你里面什么样子的了，保持神秘感哈哈
 int array_size(const Array*a)
 {
     return a->size;
 }
 int*array_at(Array*a,int index)//在然后是进行该数组的访问和修改
     //这里面需要注意的是array_at的返回需要是一个指针，如此便可以做到修改
 {
     return &(a->array[index]);
```

```
 }

int*array_get(const Array*a,int index);
{
    return a->array[index];
}

void array_set(Array *a,int index, int value)
{
    a->array[index] = value;
}
 void array_inflate(Array*a,int more_size);

int main(int argc,char const *argv[])
{
    Array a = array_create(100);
    printf("%d\n",array_size(&a));
    //printf("%d\n",a.size);十七行跟十六行一个意思
    *array_at(&a,0) = 10;//将一个值写到数组里面
    printf("%d\n",*array_at(&a,0));
    array_free(&a);

    return 0;
}
```

14.1-3可变数组的自动增长

```
//从上述24行延续
 void array_inflate(Array*a,int more_size)
 {
     int*p = (int*)malloc(sizeof(int)(a->size + more_size));
     int i;
     for( i = 0; i < a->size; i++ ) {
         p[i] = a->size[i];
     }//可以换成标准库的函数mencpy，效率更高
     free(a->array);
     a->array = p;
     a->size += more_size;
 }//核心代码

int main(int argc,char const *argv[])
{
    Array a = array_create(100);
    printf("%d\n",array_size(&a));
    //printf("%d\n",a.size);十七行跟十六行一个意思
    *array_at(&a,0) = 10;//将一个值写到数组里面
    printf("%d\n",*array_at(&a,0));
    int number;
    int cnt = 0;
    while(number != -1 ){
        scanf("%d",&number);
        if( number != -1 ){
        *array_at(&a,cnt++) = number;
        //scanf("%d",array_at(&a,cnt++));
```

```
    }//无限的读入整数，让自己不断的自己增长
    array_free(&a);

    return 0;
}
//第九行的内容改动

int*array_at(Array*a,int index)//在然后是进行该数组的访问和修改
     //这里面需要注意的是array_at的返回需要是一个指针，如此便可以做到修改
 {
    if( index >= a->size ){
        array_inflate(a,(index/BLOCK_SIZE+1)-a->size+1);//这里有点乱，后续要来修正
    }
     return &(a->array[index]);
 }
```

14.2-1可变数组的缺陷

1. 申请使用内存，当我们需要更大的内存而之前的不需要了之后，之前的就会被废弃掉，在内存受限的情况下(比如单片机)就会导致内存明明还有，但是却已经申请不了更大的内存了(浪费内存空间)
2. 效率极低

14.2-2链表

1. 这是为百度进来补充的图，翁恺的课程是没有静态的图，用动态进行演示的

链表插入元素

同顺序表一样，向链表中增添元素，根据添加位置不同，可分为以下 3 种情况：

- 插入到链表的头部（头节点之后），作为首元节点；
- 插入到链表中间的某个位置；
- 插入到链表的最末端，作为链表中最后一个数据元素；

虽然新元素的插入位置不固定，但是链表插入元素的思想是固定的，只需做以下两步操作，即可将新元素插入到指定的位置：

1. 将新结点的 next 指针指向插入位置后的结点；
2. 将插入位置前结点的 next 指针指向插入结点；

例如，我们在链表 {1,2,3,4} 的基础上分别实现在头部、中间部位、尾部插入新元素 5，其实现过程如图 1 所示：

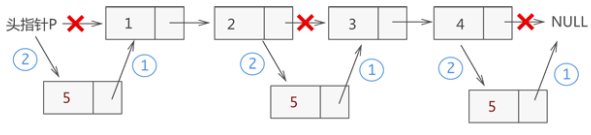


图 1 链表中插入元素的 3 种情况示意图

从图中可以看出，虽然新元素的插入位置不同，但实现插入操作的方法是一致的，都是先执行步骤 1，再执行步骤 2。

注意：链表插入元素的操作必须是先步骤 1，再步骤 2；反之，若先执行步骤 2，会导致插入位置后续的部分链表丢失，无法再实现步骤 1。

```
#include"node.h"
#include<stdio.h>
#include<stdlib.h>

//typedef struct _node{
//    int value;
//    struct _node *next;
//}Node;

int main(int argc,char const argv[])
{
    Node*head = NULL;
```

```
    int number;
    do{
        scanf("%d",&number);
        if( number != -1 ) {
            //add to linked-list
            Node*p = (Node*)malloc(size(Node));
            p->next = NULL;
            //find the last
            Node*last = head;
            if( last ){
                while (last ->next){
                    last = last->next;
                    }
                //attach
                last->next = p;
                }else{
                    head = p;
            }
        }
    }while( number != -1 );

    return 0;
}
```

14.2-3链表的函数

```
#include"node.h"
#include<stdio.h>
#include<stdlib.h>

//typedef struct _node{
//    int value;
//    struct _node *next;
//}Node;
void add(Node*head,int number);

typedef struct _list{
    Node*head;
    Node*tail;
}List;

int main(int argc,char const argv[])
{
    Node*head = NULL;
    List list;
    int number;
    list.head = list.tail = NULL;
    do{
        scanf("%d",&number);
        if( number != -1 ){
            add(&list,number);
        }
    }while( number != -1);
```

```
    return 0;
}

void add(List*pList,int number)
{
       //add to linked-list
    Node*p = (Node*)malloc(size(Node));
    p->value = number;
    p->next = NULL;
    //find the last
    Node*last = pList->head;
       if( last ){
          while (last ->next){
              last = last->next;
            }
          //attach
           last->next = p;
          }else{
            head = p;
    }
    return head;
}
```

14.2-4链表的搜索

```
//从上述16行延续
void print(List *pList)//函数原型置顶，另外说明我没有把这个置顶到其他代码块里面
void add(List*pList,int number)

int main(int argc,char const argv[])
{
    Node*head = NULL;
    List list;
    int number;
    list.head = list.tail = NULL;
    do{
        scanf("%d",&number);
        if( number != -1 ){
            add(&list,number);
        }
    }while( number != -1);

    printf(&list);
    scanf("%d",&number);
    Node*p;
    int isFound = 0;
    for( p = list.head; p; p = p->next){
        if( p->value == number ){
            printf("找到了\n");
            isFound = 1;
            break;
        }
    }
    if ( !isFound ){
```

```
        printf("没找到\n");
    }
    //Node*p;
    //for( p=list.head; p; p = p->next){
    //    printf("%d\t",p->value);
    //}//遍历，把链表每个节点的值打出来
    //printf("\n");  这些内容转移到下面了，并且有了一部分的改动

    return 0;
}

void add(List*pList,int number)
{
       //add to linked-list
    Node*p = (Node*)malloc(size(Node));
    p->value = number;
    p->next = NULL;
    //find the last
    Node*last = pList->head;
       if( last ){
          while (last ->next){
              last = last->next;
            }
          //attach
           last->next = p;
       }else{
            head = p;
       }
}

void print(List *pList){
    Node*p;
    for( p=list->head; p; p = p->next){
        printf("%d\t",p->value);
    }//遍历，把链表每个节点的值打出来
    printf("\n");
}
```

14.2-5链表的删除

```
//从上述第5行开始
int main(int argc,char const argv[])
{
    Node*head = NULL;
    List list;
    int number;
    list.head = list.tail = NULL;
    do{
        scanf("%d",&number);
        if( number != -1 ){
            add(&list,number);
        }
    }while( number != -1);
```

```
    printf(&list);
    scanf("%d",&number);
    Node*p;
    int isFound = 0;
    for( p = NULL; p=list.head; q = p,p = p->next){
        if( p->value == number ){
            //需要考虑到边界效应，q没有进行限制需要进行限制
            if(q){
                q->next = p->next;
            } else {
                list.head = p->next;
            }
            //q->next = p->next;
            free(p);
            break;
        }
    }
    //Node*p;
    //for( p=list.head; p; p = p->next){
    //    printf("%d\t",p->value);
    //}//遍历，把链表每个节点的值打出来
    //printf("\n");  这些内容转移到下面了，并且有了一部分的改动

    return 0;
}
```

14.2-6链表的清除

如何整个链表都清除掉

```
    //从上述19行开始
    for( p = NULL; p=list.head; q = p,p = p->next){
        if( p->value == number ){
            //需要考虑到边界效应，q没有进行限制需要进行限制
            if(q){
                q->next = p->next;
            } else {
                list.head = p->next;
            }
            //q->next = p->next;
            free(p);
            break;
        }
    }
    for( p=head; p; p=q ){
        q = p->next;
        free(p);
    }//链表这样就删除掉了


    return 0;
}
```