基础知识

数据类型

基本类型数据：无法再进行分割的数据：浮点数，字符，整数

整数分为多种

int 用32位 即4个字节来储存一个整数 范围+-32767

short int 用16位即2个字节来储存一个整数 范围与int一般一致

long int用至少32位来储存一个整数 范围+-2147483648

long long int 用64位来存储一个整数 范围+-9.2*e18

unsigned int 用32位来存储一个非负数，范围（0，65535）

int i=10 说明i变量只能是整数

如果输入 int i=10.6 就会输出10

浮点数

A分为单精度浮点数float四个字节 和双精度浮点数double 八个字节

float 可以在取六位有效数字的情况下存储e-37~e37之间的数 使用32位

特别注意：float只能取到六位有效数字，因此float可能无法精确存储大量的数。

例如说，它可能无法储存一个特别趋近于0的数，比如0.0000000023

double可以在取到13个有效数字，占用8个字节

浮点数默认采取double精度，可以在数字后方注明f或L使其变成float精度和long double精度。

字符可以是单个字符 char

char i = M 一个字符占据一个字节

复合类型数据：基本数据拼在一起
结构体 枚举 共用体（已被淘汰）

变量

变量就是一个容器 用来保存数据

变量的本质：内存中的一块存储空间

变量为什么要进行初始化（赋值）？

当程序终止时，它的数据会残留在它使用过的内存里而不会自动删除，这些残留数据称为垃圾数据。如果一个新变量调用了旧程序遗留下来的内存但没有初
始化（赋新值），这时，他们调用的内存里实际上是垃圾数据，操作系统发现这一情况以后，就会自动输出一个十分混乱巨大的数字，警告使用者的变量没有初始化。因此，对每一个变量和数组分配空间以后都必须初始化。

如何定义变量?
数据类型 变量名 赋值

进制

十进制锋十进一 二进制逢二进一

十进制
1 2 3 4 5 6 7 8 9 10

二进制
1 10 11 100 101 110 111 1000 1001 1010 1011 1100

冯·诺依曼首次将计算机的十进制改为二进制
十进制的5和二进制的101本质上是完全一样的数字

八进制 用0~7 表示
十六进制 用0~9十个数字加上A~F六个字母表示

输入/输出控制符

%d以十进制输出 %ld以长整型输出 %hd以短整型输出

%x以十六进制输出 %#x 十六进制带符号0x输出 %lx %hx

%o以八进制输出 %#o 八进制带符号0输出 %lo %ho

%u 以unsigned即非负整数输出

%e 浮点数 科学计数法

%f 浮点数 十进制

%5.2f 打印一个字段宽度5 保留到小数点后两位的浮点数

常量的表示

整数：十进制时 可以直接用传统写法

十六进制 前面加0X或0x

八进制 前面加零

浮点数 传统的写法、科学计数法

科学计数法
 float x = 3.2e3 x= 3200
 float x = 1.45e-4 x = 0.000145

字符 单个字符用单引号表示 用单引号括起来
字符串用双引号括起来

常量的存储

计算机存储的是二进制代码

例如 int x = 86 86会被转化成二进制代码。

整数是以补码形式存储的

浮点数是以IEEE754形式存储的

字符是以补码形式存储的

字节

字节是存储数据的单位，是硬件能访问的最小单位。
一个字节等于8位 每一位代表一个0/1代码
1K=1024字节
1M=1024K
1G=1024M
硬盘标示的容量和实际能使用的容量为什么有差距?
因为标示的容量的G是以1000为单位的，而计算机内部的G是以1024为单位的

ASCII编码

定义字符
char ch = 'A' 正确
char ch = "A" 错误 因为""表示字符串
char ch = "AB"错误 同理
char ch = 'A'
char ch = 'B'同时出现 错误 字符不能被连续定义两次
char ch = 'A'
ch = 'B' 同时出现 正确 字符可以先被定义 再被重新赋值。
ASCII码是一种规定 ， 规定了不同的字符用哪个整数值去表示。
例如 它规定了'A' = 65 'B' = 66 'a' = 97 '0' = 48
字符先被转化成ASCII码对应的整数 接着又转化成二进制代码，因此字符和整数的存储本质上是一致的

基本的输入输出函数

printf

输出变量的值

printf（）将变量的内容输出到显示器上

第一种：printf（"字符串"\n）例 printf（"哈哈！\n"） \n表示换行;
第二种：printf（"输出控制符"，"输出参数"）例 printf (d%\n, i) d%\n是输出控制符;
第三种：printf ("输出控制符1 输出控制符2……", 输出参数1, 输出参数2.。。。);
printf ("d% d%\n", j, k) 注意 输出控制符和输出参数的 个数应该匹配;
第四种：printf ("输出控制符 非输出控制符", 输出参数);

输出控制符的种类：%d %ld %f %c %lf %x/%x/%#X;
例如 printf ("I = d% j =d%\n", I, j);

printf为什么需要输出控制符？01组成的代码可以表示数据也可以表示指令，输出控制符确定了解读01代码的方式.如果01代码表示的是数据，那么同样的01代码以不同的进制或格式输出就会有不同的结果

scanf

通过键盘将数据输入到变量中scanf有两种用法
第一种 scanf ("输入控制符", "输入参数");
将键盘输入的字符转化为输入控制符规定的数据 然后存进输入参数指定的变量中;
为什么？因为从键盘输入的东西本质并不是数字，只是字符，必须有输入控制符来解读键盘输入的字符;
例 scanf ("d%\n", &i) &是取地址符;

第二种 scanf ("非输入控制符 输入控制符", 输入参数)
在这种情况下非输入控制符必须被原样输入;
例如scanf ("m d%/n", &i) 用户必须输入m+一个合法十进制数字 输入才被允许;
如何用scanf赋值多个变量?

int i, j;
scanf ("%d %d", &I, &j);
printf("I = %d j = %d", I, j); 这个程序要求输入两个数 然后把这两个数返还;

scanf使用指南:

scanf对用户非法输入的处理

```
// scanf 对用户非法输入的处理

int i;
scanf("%d", &i);
printf("%d\n", i)

//用户本来应该输入一个十进制的数字，但是一不小心输了一个123m，此时，输出的仍然是123

//又例如:
int i;
scanf("%d", &i);
printf("%d", i)

int j
scanf("%d", &j)
printf("%d\n",j)

//如果这时候用户不小心输了一个123m 那就会出错了。

    //这说明 两个scanf之间应该有一行代码，清除掉前面已经出错的值，因此应该是

char ch
    int I;
    scanf("%d", &i);
    printf("%d\n", i)
while((ch = getchar ())!='\n')
    continue
int j
    scanf("%d", &j)
    printf("%d\n",j)
```

运算符

算术运算符

- ◦ ▪ / % (取余数)

整型变量相除则商也是整数。但如果被除数和除数中只要有至少一个是浮点数，那商也将是浮点数。

取余的变量必须是整型。结果会是整除或者余数

输出余数的代码

printf("%d %d %d %d\n", 3%3, 13%3, 15%3, 20%3. -13%4, 20%5)

关系运算符

> , <, >=, <=, !=,==

逻辑运算符

!(非)， &&(并且), ||(或)

并且的用法：真&&真=真，真&&假=假，真&&假=假

或的用法：真||真=真，真||假=真

C对真假的判断：非0是真，0是假。真在二进制代码用1表示，假用0表示

```
//例
# include <stdio.h>
int main (void)
{
    int i = 20;
    int j =10;
    int m;
    m = (3>2)&&(j=8);//表面上看，j=8好像是错的，但实际上=是赋值，也就是把8赋给了j，也就不存在错误的问题了，故输出的m肯定是真的。
    //特别地：当&&左边为假时，右边的代码就不会执行。
    //例如当 m =(1>2)&&(j =8)时，上述代码将输出0 10反之，当||左边为真时，右边的表达式便不会执行，只有左边是假的右边才会执行

    printf("%d %d\n", m, j);

    return 0;
}
```

赋值运算符

=， += （a+=3就是a=a+3,即把a+3重新赋值给a）
同理a/=3 等价于 a=a/3
还有*=，+=，-=.

优先级：算术>关系>逻辑>赋值
例如 int k = 3+24&&5||6-4+=6 = (3+24)&&5||(6-4)*6

流程控制

什么是流程控制

程序代码执行的顺序就是流程控制

流程控制的分类

顺序执行、选择执行、循环执行
图灵证明了用三种流程控制，可以解决出世界上所有的流程问题

选择流程（if/switch）

选择：某些代码可能执行，也可能不执行。

```
if(3>2)
    printf("222222\n")
    //如果3>2 输出22222 如果表达式为真，执行代码，表达式为假则不执行代码
```

```
if (0)
    printf(“cccccc\n”)
    //此时，什么也不会输出

if(1>2)
    printf(“aaaaaaa”)
    printf(“AAAAA”)
    //程序会输出AAAAA这表明第二个语句不受if的控制，if默认控制一个语句
    //但 if {。。。。。}可以控制括号内所有语句
if 1
  A;
else if 2
  B;
else if 3
  C;
else
  D
    //if else 的标准结构

//练习 编写一个给分数分级的程序
# include <stdio.h>

int main(void)
{
        float score;
        scanf(“%d”, &score);
        if (score > 100)
        printf(“做梦！\n”);
        else if(score>=90 && score <=100);
        printf(“优秀\n”);

        ……
        return 0;
}
```

排序的思想

给 a b c d e f g 等若干个数排序，应该采取这样的算法：
比较a与b，若a比b大，则保留a，若a比b小，ab互换
比较a与c，重复上述步骤
这样若干步结束后，就会使a到g从高到低排列

循环流程(for)

循环流程就是指某些代码会被重复执行
for (a = 1①, a <= 100②, ++a③)
sum = sum + a④;
首先执行 a=1
检验a <=100 然后执行 sum = sum + a, 最后执行++a 这称为一个循环
执行一个循环后，重新检验a<=100 开始一个新的循环
即语句的执行顺序为1 243 243 243 243 243……

```
for(1, 2, 3)
    for (4, 5, 6)
    A;
B;
```

执行顺序是什么?
1-2（判断为真）-4-5（判断为真）-A-6-5（判断为假）-3-2（判断为假）-B

自增与自减

前自增 ++I 后自增 i++
相同点：最终都使i的值加1
不同点：++i == i+1 i++ == i
即m = i++时 虽然i的值会变为i+1 但i+1的值并不会被赋给m，i会被赋给m

while循环

while的基本结构：
while(A)
    B;

Do while

基本结构：
do
{
.....
} while (A);
while与for是等价的，while和for与do while是不等价的。在执行时，程序必定会先把do后面括号的内容执行一次，之后再检查while中的值是否成立，成立则继续执行。

Switch

Switch (val)
{
case 1:
.....
case 2:
.....
case 3:
.....
case 4:
.....
Default:
.....
}
Switch与if的区别？如果val被发现符合任何一个case，那么从该case开始向下，之后每一个case控制的代码都将被依次执行。除非break语句跳出了switch

break

break的两大功能：终止循环，跳出switch。
对于while循环来说，break终止循环后，循环直接结束，并且开始执行循环之后的语句。
对于for循环来说，例如for（1；2；3）4；break；break会使循环结束，但语句3还会再执行一次，然后再进入for后面的语句部分。
对于switch来说，如果某一个case下方的语句是以break结尾的，那这个case下方的所有case都不会被执行，程序会直接离开switch语句开始执行后面的语句。
注意：break只会终止离自己最近的那个循环。

continue

continue的功能是跳过本次循环剩余的所有语句，直接开始下一次循环。
对于while和do while循环来说，continue会使程序跳过循环中剩余的语句库，直接进入下一轮循环的测试条件。
对于for（1；2；3）4；continue来说，continue会使程序首先执行3，然后再进入测试条件2

循环的利用：回文数与斐波那契数列

思考：如何让计算机证明一个数是回文数?
对于回文数A = x1x2x3x4x5x6...xn,有
A = x1E(n-1)+x2E(n-2)+x3E(n-3)+...+xn-1E1+xn
故 xn = A%10
xn-1 = (A/10)%10
xn-2 = (A/100)%10
.....
x1 = (A/En-1)%10
要证明A是回文数，就必须验证
B = xnEn-1+xn-1En-2+xn-2En-3+...+x2E+x1 = A

思考：如何求出一个斐波那契数列的第n项?

```
// 定义变量a1=1 a2=2 sum = 0要求用户输入n
if (n=1)
    an = a1 = 1;
else if (n = 2)
    an = a2 = 2;
else
for (i = 1; i <= n; i++)
    {
    sum = a1 + a2;
    a1 = a2;
    a2 = sum;
    }
```

循环的利用：设计更好的用户交互

```
// 利用while与scanf的结合，让用户输入正确的数据，如

while(scanf("%d", &i) == 1)
/*如果用户输入的不是合法的数字，那程序将会跳出*/

while((ch = getchar()) != \n)
/*在这种情况下，输入一串字符，该循环会不断执行并且把这串字符逐个传输给ch
#当读取到最后一个换行符时，循环停止执行。换行符也被ch当作垃圾字符"吸取"了，这就有效避免了换行符污染后面的输入。*/

while(getchar() != '\n')
    continue;
/*输入一行字符，while会逐个吸取这些字符，只要吸取的字符不是换行符，
#这个循环就会不断执行下去，直到一整行的字符被完全吸取。这一语句非常实用。
#例如说：某程序要求用户输入一长串字母，但用户却在其中输入了数字，
#如果此时程序要求用户重新输入的话，就必须用这个循环，将用户已经输入过的垃圾字符都"吸取"掉，才能让用户重新输入正确的字符。*/
```

数组

定义数组

数组是由数据类型相同的数据组成的。
int a[7]代表一个含有7个整数的数组
a[0]~a[6]分别代表了数组中的7个元素
int a[7] = {1,2,3,4,5,6,7}, 一个数组便被初始化完成了。
double a[3][10]代表了一个3行、10列的二维数组。

用嵌套循环输出一个二维数组

```
int m, n;
for (m = 0; m < 3; m++)
    {
for (n = 0; n < 10; n++)
        printf(“%d”, a[m][n]);
    printf(“\n”);
    }
```

多维数组

本质上来说，不存在多维数组。但是可以将n维数组视为由n-1维数组组成的数组。

函数

定义函数

int f(double n)
代表一个函数，这个函数接受一个双精度浮点数，经过一系列操作后输出一个整数。
int main(void)
主函数，显然，主函数不接受输入值，但返回一个整数。
函数的返回值类型，就是函数的类型。

函数的声明

两种使用函数的方法：
a.将函数整体语句写在main函数之前，不推荐。
b.声明函数，并将函数语句写在main函数之后。

声明函数需要注意什么？编译器是从上到下编译的，声明的意义就是告诉编译器，这是一个函数。例如说，一段程序中自定义了f（）和g（）两个函数，但g（）调用了f（），那么f（）的声明就必须在g（）之前出现，这样才能让编译器在g（）中读到f（）的时候能理解这是一个函数。

为什么要使用函数?

模块化的编程思想
为什么说C语言是面向过程的语言？因为函数是C语言的基本单位。
一个庞大的C源代码文件，就是由无数个函数组成的。每一个函数都只具有最普通、最基本的功能，这就是模块化的编程思想。每个函数的功能越简单、越单一，模块化程度就越高，代码的重复利用率就越高，main函数内的语局就越简洁，重复累赘就越少。

形参和实参

int a(int b, int c)
这个函数接受两个参数b和c，b和c就称为函数的形参。
在调用函数时，写成a(5, 7) 5和7就是函数的实参。

注意：不同的函数中可能使用同样的字母来定义变量，在这种情况下，这些变量只在自己所在的函数中起作用，这种变量被称为局部变量。
但是，有时main函数开始之前会声明全局变量，局部变量和全局变量也有可能使用了同一个字母。在这种情况下，函数将会采取局部变量的值。

return

return代表函数的结束，return后面的语句将不会被执行。
return返回的值应该与函数的类型保持一致。
如果int型函数return了浮点数，那么该浮点数就会被强制转化为整数。

递归

递归就是函数自己调用自己。
注意：递归是在被称为栈的结构里完成的。栈就像一个杯子，先进去的数据在杯子的底部，后进去的数据在杯子的顶部，因此越晚进去的数据越先出来，越先进去的数据越晚出来。

```
# 例如定义函数
void u(int n)
{
    if (n < 4)
        up_and_down(n + 1)
    printf(“%d”, n);
}
```

将n = 1输入时，u先调用了一次自己，也就是执行了u（2）；
u（2）又调用自己，执行了u（3）；
u（3）又调用自己，执行了u（4）；
u（4）不会再调用u（5），而是输出了4, 接着又输出了3，接着又输出了2和1；
这一函数充分表明了栈的原理：先进去的后出来，后进去的先出来。

指针

定义指针

```
int * p
/* *p是一个指针，它指向的是某一个整型变量的地址。一般来说，地址是用十六进制的数表示的
# p是一个变量，变量的类型是什么？是地址变量也就是指针变量，int * 就是p的变量类型，说明这是整型变量的地址变量。*/

printf("%p,", p);
// 将这个指针代表的地址输出，也就是p所代表的十六进制数

p = &i

// p作为一个指针指向整型变量i
// 一旦p指向了整型变量i，那么*p就和i完全等同
```

指针的重要性

表示复杂的数据结构：链表、图、树与二叉树;

快速地传递数据;

使函数传回多个值;

访问并分配内存

处理字符串

地址

内存单元的编号，一般是从0开始的非负整数。

CPU对内存进行控制的原理：控制线控制CPU与内存的关系是只读只写还是读写。数据线用于传输数组。地址线控制具体读写哪一个地址的数据。

假设现在cpu通过一根地址线控制内存，那么它只能控制2个地址，因为一根地址线只能通过高电位和低电位传递两种代码，也就是1和0。增加一根地址线后，它就能控制4处地址。再增加一根地址线，它就能控制8处地址。当操作系统是64位时，就意味着有64根地址线，一共控制着内存中的2的64次方个字节，也就是2的67次方个0/1代码。，2的64次方就是地址的范围。

指针带来的错误

```
int main (void)
{
    int i = 5;
    int * p;
    int * q;

    p = &i;
    *q = p; /*错误，因为p已经指向了i，*p也就等同于i，*p是整型变量。
    #但是p却是地址变量，地址变量不可能被赋给整型变量。*/
    p = q; /*错误，q是垃圾值，p也变成了垃圾值。
    #p本来指向的是i，但是这样的话p就指向了q原本的垃圾值所代表的一个地址。
    #这个地址本来就没有被分配给这个程序，这个程序没有权限对这个地址进行读写，error*/

    return 0;
}
```

用指针向main函数传递数据

```
# include <stdio.h>

void exchange (double, double);

int main (void)
{
    double a, b;
    printf("Please enter 2 numbers.\n");
    scanf_s("%d, %d", &a, &b);

    exchange (&a, &b);

    printf("%d, %d", a, b);
```

```
    return 0;
}

void exchange (double * c, double * d)
{
    double p

    p = * c;
    * c = * d;
    * d = p;
}
// 这样写是正确的，就是把c指针指向的变量的值和d指针指向的变量的值作了交换。

void exchange(double c, double d)
{

    double p;

    p = c;
    c = d;
    d = p;

}
// 这是错的，c和d都是局部变量，它们不可能把交换以后的结果传回main函数。
```

* 的含义

*代表乘法

*代表指针变量

*指针运算符。

该运算符放在已经定义好的指针变量的前面，如果p是一个定义好的指针，则* p表示以p的内容为地址的变量，也就是p指向的变量i。

通过被调函数修改主调函数变量的值

实参必须为该普通变量的地址。(&a, &b)

形参必须为指针变量。(int * p, int * q)

在被调函数中修改* p 和* q，也就改变了a和b的值

数组和指针

指针和一维数组

数组名：一维数组名是指针常量，存放的是一维数组第一个元素的位置。

例如对于数组a[6]来说，a 就是指针常量，a = &a[0], 也就是说* a能与a[0]等价替代。

下标和指针的关系

确定一个一维数组需要的参数

现在需要编写一个函数，这个函数能接收整个数组，并把这个函数输出。

- 怎么把数组发送给函数？先把数组的首地址发送给函数，显然数组的首地址就是数组名。函数能由此找到数组的起点。再把数组的长度发送给函数，函数就能逐个读取数组。

```
void Printarray (* arr, int length)// 把数组的首地址和数组长度发送给函数
{
    int i;

    for (i = 0; i < length; i++)
        printf("%lf", * (arr + i))// 逐个处理数组的元素
    printf("\n");
}
```

- 数组在内存里，就像坐成一排一样，逐个地分配下去。

定义数组int a[5], 那么a[0], a[1], a[2], a[3], a[4]就逐个排列下来

此时，指针* a是指向a[0]的，而指针* (a+1)就将指向a[1]，以此类推。

即

- *(数组名 + x) == 数组名[x]

数组中数的类型决定了每两个指针之间的距离是多少。

- 例如：int型整数是由4个字节来存储的。如果* a指向的是编码为1000的一个字节，那么* (a + 1)就将指向编码为1004的那个字节。

```
void f (int * pArr, int leng)
{
    pArr[3] = 88
}

int main (void)
{
    int a[6] = {1, 2, 3, 4, 5, 6};

    printf("%d\n", a[3]); //输出4
    f(a, 6)
    printf("%d\n", a[3]);//输出88
}
```

为什么？当f(a, 6)被调用时，指针pArr便指向数组a的第一个元素，即* pArr能够与a[0]等价代换了。恒等式 * (数组名 + x) == 数组名[x] 成立。pArr[3]也就能等价代换成* (pArr + 3)，这一指针比* pArr向后指了3个元素，到达第四个元素处，而第四个元素原本存放的就是a[3].由此，pArr[3]指向了a[3],并修改了a[3]的值。

事实上，这时候指针* pArr和数组a也就建立起了一种对应关系，在f函数内部对pArr进行任意操作，都会直接作用到数组a上，仅凭一个函数，就完成了对大量数据的迅速处理。

指针变量的运算

指针变量只能相减。

- 如果两个指针变量指向的是同一块连续空间中的不同的存储单元，则这两个指针变量才能相减。例如p指向a[1], q指向a[5],那么q - p = 4

思考：指针变量到底是什么?

```
double i = 1.8766
double * p;
p = &i;
```

i 是一个双精度浮点数，它在内存中占据一段长达8个字节的“片段”。

指针变量p指向了i，p本质上是一个变量，那么这个变量的值是多少?

p的值就是变量i所占据的片段的第一个字节的地址编码。

每一个指针变量的值，本质上都是它指向的那个变量所占据的片段的第一个字节的编码。而指针变量的类型就决定了指针会从那第一个字节开始往下读取到第几个字节。当指针变量是double时，指针就会往下读取八个字节。

指针变量的值是一般是用十六进制表示的整数，指针变量本身也需要4个字节来存储。就好比指针指向的变量是酒店的各个房间，指针变量是电话号码，那电话号码本身也应该写在电话本上才能被读写。

动态内存分配

传统数组的缺点

- 数组长度必须事先制定，且只能是常整数，不能是变量。
- 传统数组占据的内存不能被程序员释放。如果一个函数里使用了一个传统数组，那么该数据所占据的内存将不能被释放，除非数组所在的函数结束。
- 数组的长度不能动态地扩张和缩小
- A函数定义的数组，在A函数结束后就消失了。不能继续用在别的函数中。

malloc函数

malloc的格式