郝斌老师的 C 语言：课堂讲解全程动手敲代码，讲解细致，对于重要知识点的讲解不厌其烦，是一个难得的 C 语言入门教程。在这里对老师的辛勤付出表示感谢。

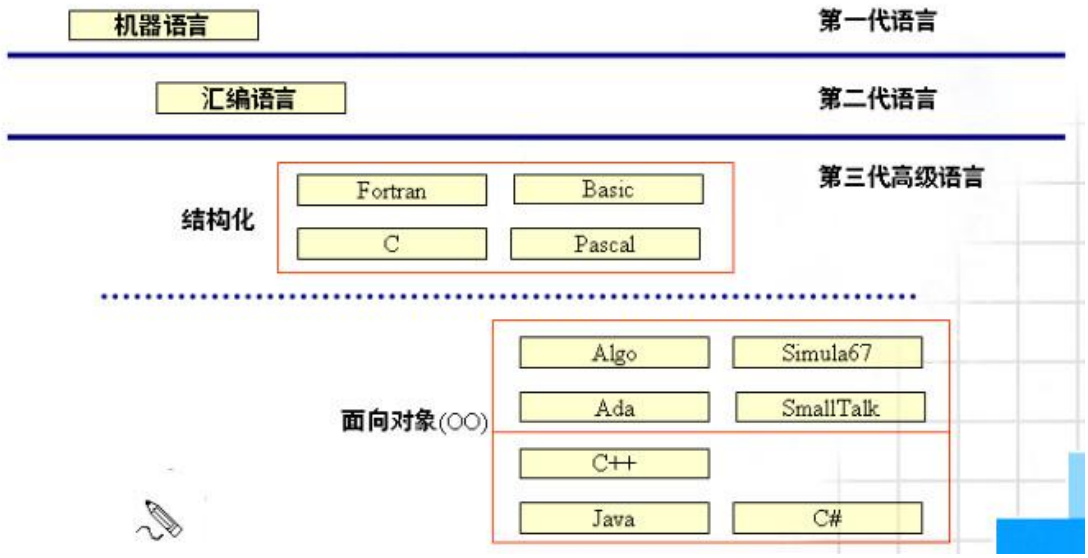
郝斌 c 语言视频教程

•

概述：

课程计划

为什么学习 c 语 言：

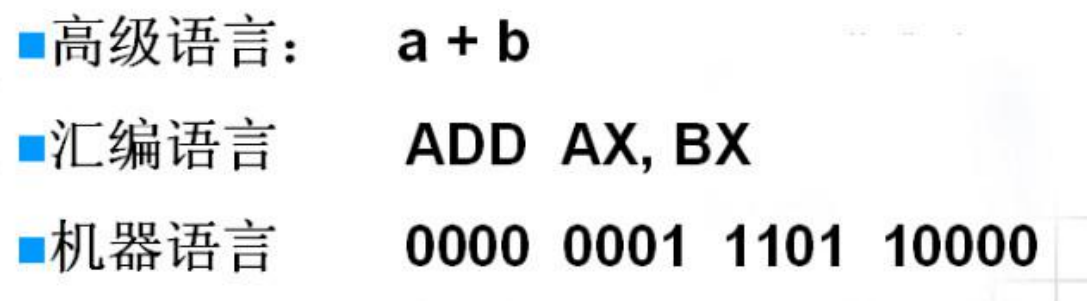


Fortran 语言主要用于科学计算，在第三代语言中，以 1980 年为分水岭，分为结构化和面向对象语言。

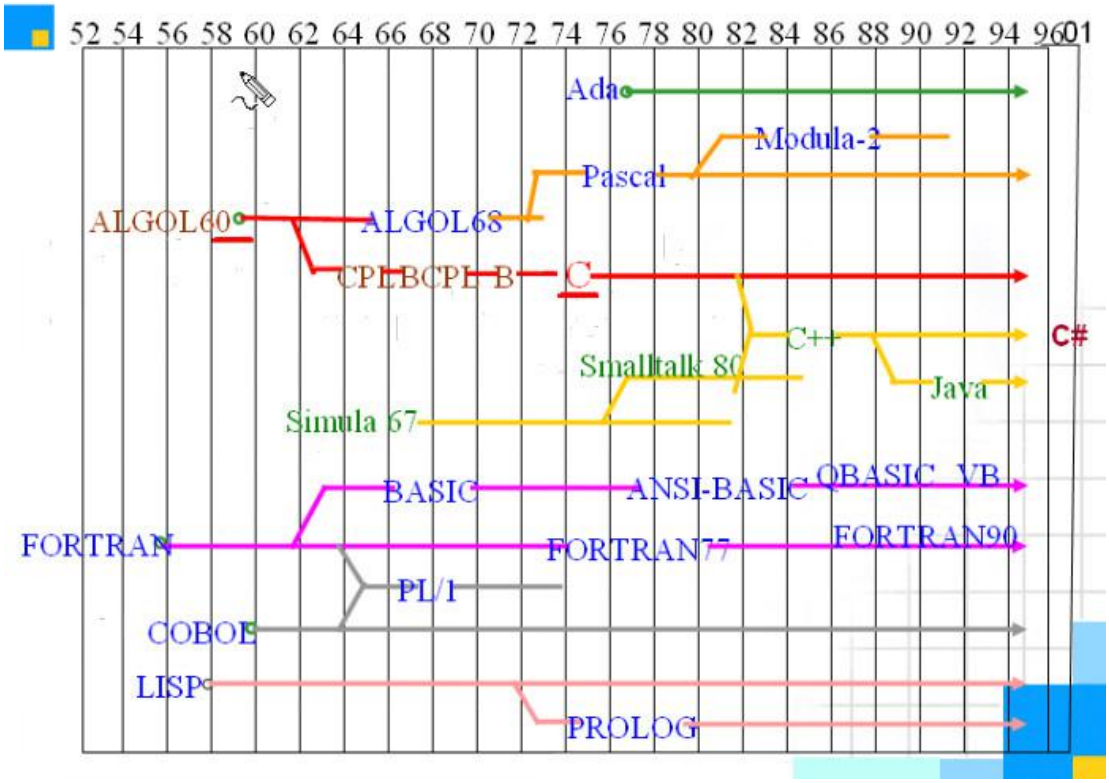
Basic 语言是 vb 的前生，pascal 语言一般是用于教学。

C 语言是最重要的，其他的语言一般很少用了。结构化的代表语言是 c 语言。结构化语言的数据和操作是分离的，导致在写大项目的时候，会出现各种各样莫名其妙的问题。

在面向对象的语言中 c++是最复杂的语言。由于 c++语言太复杂，sun 公司对 c++进行了改装，产生了 java 语言。而 c#是由微软开发的，和 java 相似，几乎一模一样。



在高级语言的执行速度上，c 是最快的，c++其次，而 java 和 c#是最后的。Java 和 c#流行，主要的一个原因是可以跨平台。



C 语言的发展和过程：

- 产生时间：1972-1973
- 产生地点：美国贝尔实验室(Bell)
- 创始人：Dennis.M.Ritchie和 Ken.Thompson
- 目的：改写UNIX操作系统

- C语言发展过程
  - 1983年 ANSI C
  - 1987年 ANSI C 87
  - 1994年 C99

C 语言的特点：

• 优点：代码量小，速度快，功能强大。

• 缺点：危险性高，开发周期长，可移植性弱。

危险性高：写同一个程序，在 java 中会报错，而在 c 中不会报错，为什么呢，因为 c 认为程序你想怎么写就怎么写，c 语言认为你写的程序不是很离谱，他都认为你写的这个程序有特殊的含义。可以直接通过，而 java 则不可以。

开发周期长：c 语言是面向过程的语言，面向过程的语言的特点就是在开发大项目的时候，很容易崩溃，好比盖大楼，C 语言还要造大量的砖块、钢筋等结构原

材料，而 C++ C# JAVA 则进行了一定的继承封装等操作，相当于原材料直接给你，你只需要用它盖楼即可。
现在市场上的语言分三块
C/c++:单纯的学习 c 是什么都做不了的。
Java
C#
可移植性不强：这是针对 java 来说的，因为 java 的可移植性太强了，所以就感觉说 c 的可移植性不强。
金山公司最主要是靠 wps 办公软件来发展的。Wps 是 c 语言开发的，其安装包比 Office 少了 10 多倍。
三大操作系统：windows，unix，linux
Windows 内核是 c 语言写的，而外壳是 c++写的。Java 永远不可能写操作系统。因为 java 运行速度太慢了。
而 linux 和 unix 都是纯 c 写的。
操作系统控制了硬件，如果说操作系统的运行速度慢，那么当我们在运行软件的时候，运行速度会更慢。
为什么使用 c 语言写操作系统呢，首先是因为 c 的运行速度快，然后是因为 c 可以直接控制硬件，而其他语言不可以。没有指针的语言是不能直接访问硬件的。
C 语言的应用领域：

- 系统软件开发
  - 操作系统：Windows、Linux、Unix
  - 驱动程序：主板驱动、显卡驱动、摄像头驱动
  - 数据库：DB2、Oracle、Sql Server
- 应用软件开发
  - 办公软件：Wps
  - 图形图像多媒体：ACDSee Photoshop MediaPlayer
  - 嵌入式软件开发：智能手机、掌上电脑
  - 游戏开发：2D、3D游戏

驱动一般是用 c 和汇编来写的。
数据库一般是用 c 和 c++来写的
C 语言的重要性：

- 有史以来最重要语言
- 所有大学工科和理科学生必修课程
- 最重要系统软件：windows、linux、unix均使用c开发
- 一名合格黑客必须掌握的语言
- 任何一个想终身从事程序设计和开发人员必须熟练掌握的语言
- 大企业、外企招聘程序员必考的语言
- 为学习数据结构、 C++、Java、C#奠定基础

虽然应用场合相对较窄，但贴近系统内核，较底层。病毒最基本的是要感染系统，
数据结构，c，c++这三门语言是必须要学习的。
牛人牛语：

> 入门最基本的方法就是从C语言入手。
>
> 当你成为C语言的高手，那么就你很容易进入到操作系统的平台里面去；当你进入到操作系统的平台里去实际做程序时，就会懂得进行调试；当你懂得调试的时候，你就会发现能轻而易举地了解整个平台的架构。这时候，计算机基本上一切都在你的掌握之中了，没有什么东西能逃得出你的手掌心
>
> ------《编程箴言》梁肇新

怎样学习 c 语言
要将编程当成一项事业来经营，而不是糊口的工具。
多思考，多上机。 不能光看，光听，而要排错，调试。
在犯错误中成长。
参考资料

- 谭浩强 《C语言程序设计》清华
- 《The C programming language》 机械工业
- 《C Primer Plus》60元 人名邮电
- 《C和指针》 65元 人名邮电
- 《C专家编程》绝版
- 《C陷阱与缺陷》人名邮电30
- 《C科学与艺术》 机械工业

王爽写的 c++也很不错

学习的目标：

掌握简单的算法--解决问题的方法和步骤。

熟悉语法规则。

能看懂程序并调试程序。

C 语言的关键字：

32个关键字：(由系统定义，不能重作其它定义)

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	unsigned	union	void
volatile	while			

C 语言程序的格式：

```
# include <stdio.h>

int main(void)
{
    I

    return 0;
}
```

一定要养成良好的习惯：代码规范
边写边保存 ，括号成对出现，应用空格
VC6.0 软件操作：
新建 保存 关闭（关闭空间）
.cpp 是原始文件，可单独拷贝到其它电脑。

第二讲：（14）c 语言编程必备知识

1. Cpu，内存条，硬盘，显卡，主板，显示器之间关系。
   Cpu 不能直接处理硬盘上的数据，必须要先调入内存
2. Hello word 程序是如何运行起来的。
3. 什么是数据类型

数据类型--数据的分类，对编程而言，首要考虑问题是数据的输入和存储。可以分为
A：基本数据类型：
整型
整型 int --4 字节 一字节 byte = 8 位 bit
短整型 short int -2
长整型 long int -8
浮点型
单精度浮点数 float：存储范围小 -4
双精度浮点数 double：存储范围大 -8
Float 和 Double 都不能保证将小数完全准确保存。
字符 char：c 语言中是没有字符串 string -1
（区别于 JAVA、C#中有 string 且 C#中 char 为 2 字节）
B：复合类型：就是把基本类型拼凑在一起
结构体
枚举 --- 实用
共用体—基本淘汰

4．什么是变量
变量的本质是内存中一段存储空间。
Int I; i=5; I 是变量，程序向系统申请了一个内存单元，在程序运行中，i 的值可以改变，但程序结束后，其所占的空间不是释放，而是被系统收回权限。

5 Cpu，内存条，vc++6.0，操作系统之间的关系。

6 变量为什么必须初始（即赋值）
软件运行与内存关系（垃圾数据-9868598658）

1. 软件在运行前需要向操作系统申请存储空间，在内存空间足够空闲时，操作系统将分配一段内存空间并将该外存中软件拷贝一份存入该内存空间中，并启动该软件运行。
2. 在软件运行期间，该软件所占内存空间不再分配给其他软件。
3. 当该软件运行完毕后，操作系统将回收该内存空间（注意：操作系统并不清空该内存空间遗留下来的数据），以便再次分配给其他软件使用。《操作系统》一门课中系统分配表中会讲到，用 1 标记表示内在是被占用的，用 0 标记表示是空闲的。
   综上所述，一个软件所分配到的空间中极可能存在着以前其他软件使用过

后的残留数据，这些数据被称之为垃圾数据，所以通常情况下我们为一个变量，为一个数组，分配好存储空间之前都要对该内存空间初始化。

7 如何定义变量

数据类型 变量名称 = 赋予的值；

等价于

数据类型 变量名；

变量名 = 要赋予的值；

举例子：

int i = 3; 等价于 int i； i = 3；

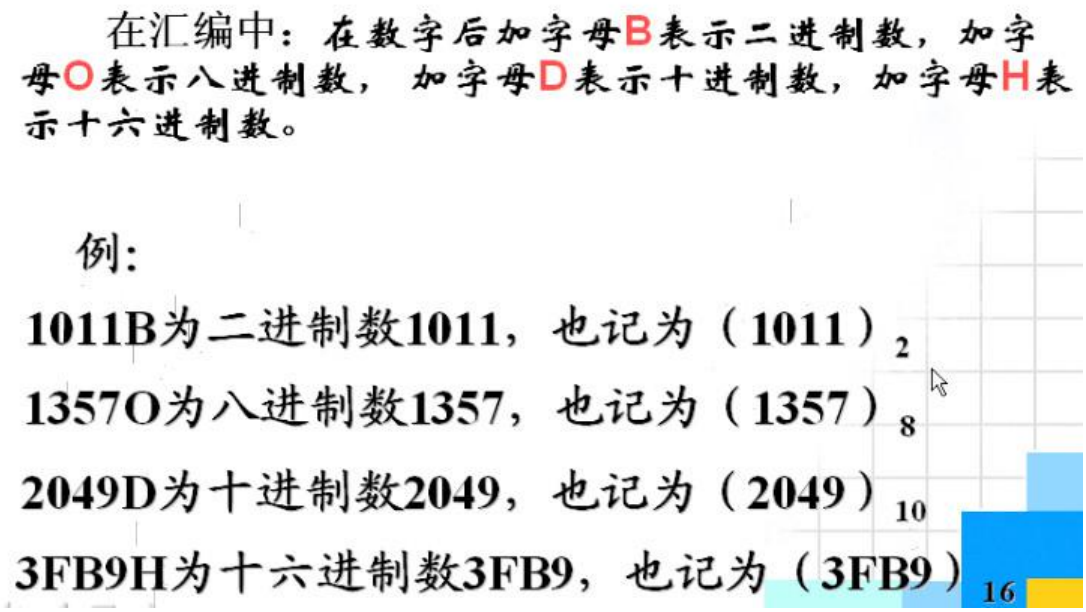
Int i，j；等价于 int i；int j；

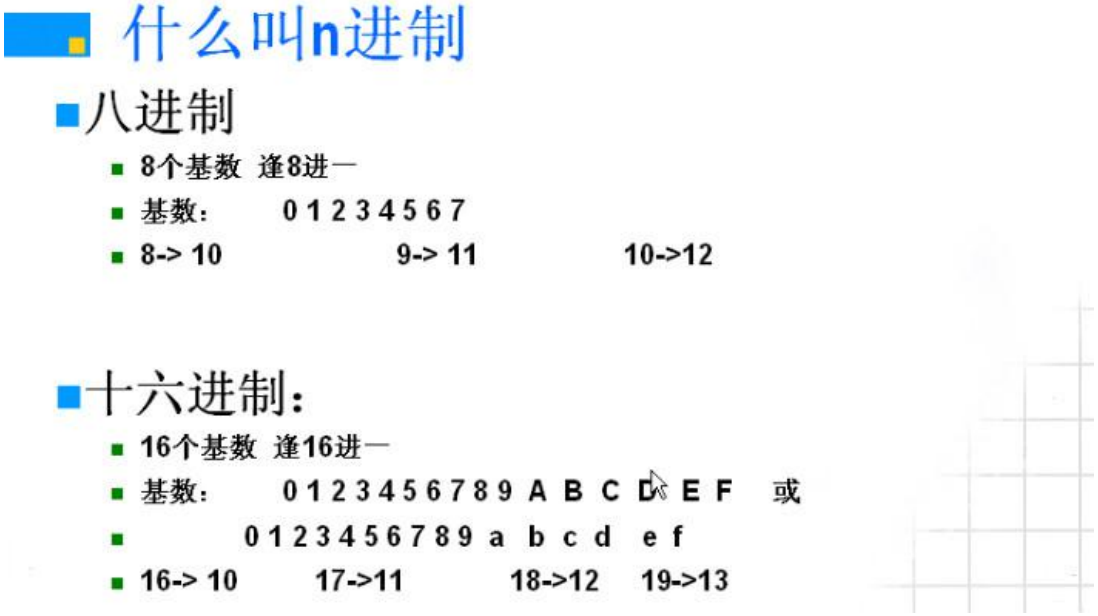
Int i，j=3 等价于 int i； int j；j=3;

Int I =3， j = 5；等价于 int i； int j； I = 3；j = 5；

8 什么是进制 –逢几进一

我们规定八进制前面加 0（零），十六进制前面加 0x。





常用计数制对照表：

常用计数制对照表

十进制(D)	二进制(B)	八进制(O)	十六进制(H)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f

Printf 的基本用法：

```
printf("i = %d\n", i);
/*
    printf的用法
    %d表示以十进制输出
    %x或%X表示以十六进制输出
    %o表示以八进制输出
*/
```

9 常量在 c 中是如何表示的

整数

十进制：　　传统的写法

十六进制：　　前面加0x或0X

八进制：　　前面0　注意是数字零不是字母o

浮点数

传统的写法

```
float x = 3.2;//传统
```

科学计数法

```
float x = 3.2e3;   //x的值是 3200
float x = 123.45e-2;   //x的值是1.2345
```

字符

当个字符使用单引号括起来，多个字符串使用双引号括起来（指针、数组）。

```
int main(void)
{
    float x = 123.45e-2F;

    printf("%f\n", x);

    return 0;
}
```

在 c 中，默认是 double 类型的。在后面加 F 表示当做 float 来处理，否则会有警告提示 --丢失部分字节。

10 常量以什么样的二进制代码存储在计算机中？

编码：

整数是以补码的形式转换为二进制代码存储在计算机

浮点数是以 ieee754 标准转换为二进制代码存储

字符本质实际是与整数的存储方式相同，ASII 码标准。

第三次课：

代码规范化

- 可以参考林锐《高质量 c/c++编程》
- 代码的规范化非常的重要，是学习一门编程语言的基础，代码可以允许错误，但不能 不规范。

例如：

成对敲括号{} （）

加空格于 运算符和数字之间 I = 1 + 2;

加缩进 分清上下级地位。

换行--进行功能区域分隔 or { }括号单独成一行。

- 代码规范化的好处

1：整齐，别人和自己都容易看懂。

2：代码规范了，代码不容易出错。

3：一般的程序可以分为三块:

a: 定义变量

b：对变量进行操作

c: 输出值

什么是字节

- 存储数据的单位，并且是硬件所能访问的最小单位。

内存中存储的最小单位是位 bit(0 或 1)，但是硬件控制的时候不能精确到位，只能精确到字节（8 位），是通过地址总线来控制的，而精确到位是通过软件来控制的，叫做位运算符来精确到位的。

1 字节 = 8 位 1K = 1024 字节

1M = 1024 K 1G =1024 M 1T = 1024 G

2G 的内存条的总空间：2 *1024 * 1024 *1024 * 8 =4*10^{32}

不同类型数据之间相互赋值的问题

不同数据类型之间最好不要相互转换。

```
int i = 45;
long j = 102345;
i = j;
printf("%ld %d\n", i, j);
float x = 6.6;
double y = 8.8;
printf("%f %lf\n , x, y");

int i = ;
```

如果需要明白这个知识点，那么需要明白补码。

什么是 ASCII 码

以 char 定义变量的时候，只能使用单引号括起一个字符才是正确的。

```c
# include <stdio.h>

int main(void)
{
    char ch = 'A';  //4行    OK  等价  char ch;  ch = 'A';
//  char ch = "AB";  //error 因为"AB"是字符串，  我们不能把字符串赋给单个字符
//  char ch = "A";   //error

//  char ch = 'AB';  //'AB'是错误的

//  char ch = 'B';  //error, 因为ch变量已经在4行定义了，  这样会导致变量名被

    ch = 'C';
    printf("%c\n", ch);
```

在上图中注释的最后一样是重复定义了 ch 的值，是错误的，而下面的 ch = ‘c’ 是指把 c 赋值给 ch，是正确的。

```c
# include <stdio.h>

int main(void)
{
    char ch = 'b';

    printf("%d\n", ch);

    return 0;
}
```

上图中输出的值是 98(将字符以整数%d 的形式输出)

Ascll 码规定了 ch 是以哪个值去保存，ascii 码不是一个值，而是一种规定，规定了不同的字符是以哪个整数值去表示。其它规定还有 GB 2312 UTF-8 等。

14. 什么是ASCII
ASCII不是一个值，而是一种规定，
ASCII规定了不同的字符是使用哪个整数值去表示
它规定了
'A' -- 65
'B' -- 66
'a' -- 97
'b' -- 98
'0' -- 48

字符本质上与整数的存储方式相同【字符的存储】

基本的输入和输出函数的用法：

第三次课

Printf（）

将变量的内容输出到显示器上。

四种用法

1. printf("字符串");
2. printf("输出控制符", 输出参数);
3. printf("输出控制符1 输出控制符2。。。", 输出参数1,
   输出控制符和输出参数的个数必须一一对应
4. printf("输出控制符 非输出控制符", 输出参数);

输什么是输出控制符，什么是非输出控制符
输出控制符包含如下：

%d	--	int
%ld	--	long int
%c	--	char
%f	--	float
%lf	--	double
%x(或者%X后者%#X)	--	int 或 long int 或 short int
%o	--	同上
%s	--	字符串

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。必须要有输出控制符告诉他怎么去解读。
- 如果 01 组成的代码表示的是数据的话，那么同样的 01 代码组合以不同的格式输出就会有不同的输出结果，所以必须要有输出控制符。

```
# include <stdio.h>

int main(void)
{
    int x = 47;   //100是十进制

    printf("%x\n", x);   //输出结果是:  2f
    printf("%X\n", x);   //输出结果是:  2F
    printf("%#X\n", x);  //输出结果是:  0X2F
    printf("%#x\n", x);  //输出结果是:  0x2f


    return 0;
}
```

在上图中，int x =47，如果前面加 0（零）048 表示的是八进制，如果前面加 0x（零 x）0X47 则表示的是十六进制，而在输出的时候，则是 o(字母 o)表示八进制，ox（字母 o，x）表示十六进制。
非输出控制符：非输出控制符在输出的时候会原样输出。

```
printf("i = %d, j = %d\n", j, k);
```

Scanf（）通过键盘将数据输入到变量中
有两种用法：

用法一： scanf("输入控制符", 输入参数);
功能： 将从键盘输入的字符转化为输入控制符所规定格式的数据，然后存入以输入参数的值为地址的变量中

示例：

```
# include <stdio.h>

int main(void)
{
    int i;

    scanf("%d", &i); //&i 表示i的地址 &是一个取地址符
    printf("i = %d\n", i);

    return 0;
}
```

非输入控制符：在输入的时候也会原样输入。

用法二： scanf("非输入控制符 输入控制符", 输入参数);
功能： 将从键盘输入的字符转化为输入控制符所规定格式的数据，然后存入以输入参数的值为地址的变量中
非输入控制符必须原样输入

```
# include <stdio.h>

int main(void)
{
    int i;

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

    return 0;
}
```

但是强烈建议：在使用 scanf 的时候，不使用非输入控制符。
给多个变量赋值：

```
# include <stdio.h>

int main(void)
{
    int i, j;

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

    return 0;
}
```

需要记住，非控制符需要原样输入。
如何使用 scanf 编写出高质量代码

如何使用scanf编写出高质量代码

1. 使用scanf之前最好先使用printf提示用户以什么样的方式来输入
2. scanf中尽量不要使用非输入控制符，尤其是不要用\n

```
# include <stdio.h>

int main(void)
{
    int i;

    scanf("%d\n", &i);  //非常不好的格式, 不要加 \n
    printf("i = %d\n", i);

    return 0;
}
```

3. 应该编写代码对用户的非法输入做适当的处理【非重点】

```
while ( (ch=getchar()) != '\n')
    continue;
```

```
# include <stdio.h>

int main(void)
{
    int i;
    char ch;

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

    //.......
    while ( (ch=getchar()) != '\n')
//      continue;
    int j;
    scanf("%d", &j);
    printf("j = %d\n", j);

    return 0;
}
```

运算符：

算术运算符：

加（+），减（—）乘（*）除（/）取余（%）

关系运算符：

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

逻辑运算符：

!（非），&&（且），||（或）

!(非)　&&(并且)　||(或)

!真	假
! 假	真
真&&真	真
真&&假	假
假&&真	假
假&&假	假
真||假	真
假||真	真
真||真	真
假||假	假

C语言对真假的处理

非零是真

零是假

真是1表示

假是0表示

&&左边的表达式为假 右边的表达式肯定不会执行

||左边的表达式为真 右边的表达式肯定不会执行

赋值运算符：

=， +=，*=, /=

例如：a+=3 是等价于 a=a+3，a/=3 等价于 a=a/3

其优先级别是算术>关系>逻辑>赋值。

除法与取模运算符

- 除法/的运算结果和运算对象的数据类型有关，两个数都是int，则商就是int，若商有小数，则截取小数部分；被除数和除数中只要有一个或两个都是浮点型数据，则商也是浮点型，不截取小数部分。
  - 如： 16/5 == 3　　16/5.0 == 3.20000　　-13/4 == -4
  - -13/-3 == 4　　3/5 == 0　　5/3 == 1
  - 最典型的例题就是求s=1+1/2+1/3+1/4+1/5。。。。+1/100的值(具体程序我们以后再讲)。
- 取余%的运算对象必须是整数，结果是整除后的余数，其余数的符号与被除数相同
  - 如： 13%3 == 1　　13%-3 == 1　　-13%3 == -1
  - -13%23 == -13　　3%5 ==3

测试取模运算符的例子

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

2.  int main(void)
3.  {
4.      printf("%d  %d  %d  %d  %d  %d \n",3%3, 13%-3,  -13%3, -13%-3,-13%23, 3%5);
5.      return 0;
6.  }
7.  /*
8.      输出结果是：

9.  ****************************************
10. 0  1  -1  -1  -13  3
11. Press any key to continue
12. ****************************************

13.   总结：取余%的运算对象必须是整数，结果是整除后的余数，其余数的符号与被除数相同

14. */
```

取余的结果的正负只和被除数有关。

第四节

流程控制（第一个重点）：

1. 什么是流程控制

   程序代码执行的顺序。

2. 流程控制的分类

顺序执行

选择执行

定义：某些代码可能执行，可能不执行，有选择的执行某些代码。

分类：if

1. if最简单的用法
2. if的范围问题
3. if..else...的用法
4. if..else if...else...的用法
5. C语言对真假的处理
6. if举例--求分数的等级
7. if的常见问题解析

If 最简单的用法：

1. if最简单的用法

格式：

if (表达式)

语句

功能：

如果表达式为真，执行语句

如果表达式为假，语句不执行

```c
# include <stdio.h>

int main(void)
{
    if (3)
        printf("AAAA\n");   //会输出

    if (0)
        printf("BBBB\n");   //不会输出

    if (0 == 0)
        printf("CCCC\n");   //会输出

    return 0;
}
```

2. if的范围问题

1.

```
if (表达式)
    语句A;
    语句B;
```

解释：if默认只能控制语句A的执行或不执行
if无法控制语句B的执行或不执行
或者讲：语句B一定会执行

if默认的只能控制一个语句的执行或不执行

2.

```
if (表达式)
{
    语句A;
    语句B;
}
```

此时if可以控制语句A和语句B

由此可见：if默认只能控制一个语句的执行或不执行
如果想控制多个语句的执行或不执行
就必须把这些语句用{}括起来

如果想控制多个语句的执行或者不执行，那么需要使用{}括起来。

3.if…else…的用法：

```
# include <stdio.h>

int main(void)
{
    int i, j;

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

    if (i > j)
        printf("i大于j\n");
    else
        printf("i小于j\n");

    return 0;
}
```

```c
# include <stdio.h>

int main(void)
{
    if (1 > 2)
        printf("AAAA\n");
    else
        printf("BBBB\n");
        printf("CCCC\n");

    return 0;
}
```

if…else if…else 的用法：

```c
if (1)
    A;
else if (2)
    B;
else if (3)
    C;
else
    D;
```

```c
# include <stdio.h>

int main(void)
{
    double delta = 3;

    if (delta > 0)
        printf("有两个解!\n");
    else if (delta == 0)
        printf("有一个唯一解!\n");
    else
        printf("无解!\n");

    return 0;
}
```

C 错误的 if…else if…else 语句：

```c
# include <stdio.h>

int main(void)
{
    double delta = -1;

    if (delta > 0)
        printf("有两个解!\n");
        printf("哈哈!\n");
    else if (delta == 0)
        printf("有一个唯一解!\n");
    else
        printf("无解!\n");

    return 0;
}
```

在上图中，当执行到哈哈那句时，下面的 else 将会被算作另外一个语句来执行，而在我们的 c 语言中，没有以 else 开头的语句。所以会出错。

If 实例：

```c
# include <stdio.h>

int main(void)
{
    float score; //score分数

    printf("请输入您的考试成绩: ");
    scanf("%f", &score);

    if (score > 100)
        printf("这是做梦!\n");
    else if (score>=90 && score<=100) //不能写成 90<=score<=100
        printf("优秀!\n");
    else if (score>=80 && score<90)
        printf("良好!\n");
    else if (score>=60 && score<80)
        printf("及格!\n");
    else if (score>=0 && score<60)
        printf("不及格! 继续努力!\n");
```

If 常见的问题：

变量的替换：

```
    int t;  //定义临时变量

//6和7行代码无法完成i和j的互换
//  i = j;  // 6行        i = 5; j = 5;
//  j = i;  // 7行        i = 5; j = 5;

    //正确的互换i和j的方法
    t = i;
    i = j;
    j = t;

    printf("i = %d, j = %d\n", i, j);

    return 0;
```

求三个数字的大小：

```
int a, b, c;  //等价于:  int a; int b; int c;
int t;

printf("请输入三个整数(中间以空格分隔): ");
scanf("%d %d %d", &a, &b, &c);

//编写代码完成a是最大值  b是中间值  c是最小值

if (a < b)
{
    t = a;
    a = b;
    b = t;
}

if (a < c)
{
    t = a;
    a = c;
    c = t;
}

if (b < c)
{
    t = b;
    b = c;
    c = t;
}

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

return 0;
```

C 语言常见误区：纸老虎

素数：只能被 1 和自己整除的数，如 1,5,9 等。

回文数：正着写和倒着写一样的数。如 1221,121，等

编程实现求一个十进制数字的二进制形式:
求一个数字的每位是奇数的数字取出来组合形成的新数字。
求一个数字到过来的数字。
1：如果不懂，那么就看答案。看懂答案在敲。没错误了，在尝试改。
如何看懂一个程序：

1. 流程：
2. 每个语句的功能：
3. 试数：

对一些小算法的程序：

1. 尝试自己编程结局。
2. 解决不了，看答案。
3. 关键是把答案看懂。
4. 看懂之后尝试自己修改程序，且知道修改之后程序的不同输出结果的含义。
5. 照着答案去敲
6. 调试错误
7. 不看答案，自己独立把程序编出
8. 如果程序实在是彻底无法了解，就把他背会。

空语句的问题：

7. if的常见问题解析
   1. 空语句的问题

```
if (3 > 2);
等价于
if (3 > 2)
    ;  //这是一个空语句
```

```
int main(void)
{
    if (1 > 2)
        printf("AAAA\n");
    printf("BBBB\n");

    return 0;
}
```

在上图中，最终的结果会是 AAAA,BBBB，程序也不会报错，为什么呢，因为在程序执行的时候，会在；哪里认为是一个空语句。也就是说，如果 if 成立，那么执行空语句。
If 常见错误解析（重点）

```c
# include <stdio.h>

int main(void)
{
    if (3 > 2) //4行  如果这里加分号，则会导致程序编译到6行时就会出错
        printf("哈哈!\n");
    else  //6行
        printf("嘿嘿!\n");

    return 0;
}
```

上面这个程序是错误的，为什么呢，在该程序中，总的有 4 个语句，而在以 else 开头的那个语句中是有错误的，因为在 c 语言中是没有以 else 开头的这种语法。

```
if (表达式1)
    A;
else
    B;
是正确的


if (表达式1);
    A;
else
    B;
是错误的
```

```c
# include <stdio.h>

int main(void)
{
    if (3 > 2)
        printf("AAAA\n");
    else if (3 > 1)
        printf("BBBB\n");
    else
        printf("CCCC\n");

    return 0;
}
```

在上面这个程序中，最终的值是 AAAA,虽说后面的 3>1 也满足条件，但是当 3>2 满足条件后，该 if 语句就会终止，后面的语句是不会在执行的。

```
if (表达式1)
    A;
else if (表达式2)
    B;
else if (表达式3)
    C;
```

这样写语法不会出错，但逻辑上有漏洞

```
if (表达式1)
    A;
else if (表达式2)
    B;
else if (表达式3)
    C;
else (表达式4)  //7行
    D;
```

这样写是不对的，正确的写法是：

要么去掉7行的(表达式4)

要么在else 后面加if

既然 7 行要写表达式，就要写 if。

```
if (表达式1)
    A;
else if (表达式2)
    B;
else if (表达式3)
    C;
else (表达式4);
    D;
```

这样写语法不会出错，但逻辑上是错误的

```
else (表达式4);
     D;
等价于
else
     (表达式4);
D;
```

循环的定义、分类。
定义：某些代码会被重复执行。
分类：for　while　do……while

```
# include <stdio.h>

int main(void)
{
    int i;
    int sum = 0;

    for (i=1; i<=4; ++i)
        sum = sum + i;

    printf("sum = %d\n", sum);

    return 0;
```

在上图中，先执行 1，在执行 2，2 如果成立，标志着循环成立，那么在执行 4，最后在执行 3，3 执行完后代表一次循环完成，然后在执行 2.以此类推。1 永远只执行一次。
++I 等价于 i+1
求 1-10 的所有奇数的和：

```
# include <stdio.h>

int main(void)
{
    int i;
    int sum = 0;

    for (i=1; i<10; i+=2) //i+=2; 等价于 i = i + 2;
    {
        sum = sum + i;
    }

    return 0;
}
```

求 1-12 之间的所有能被 3 整除的数字之和：

```
    int sum = 0;

    for (i=3; i<=12; ++i)
    {
        if (i%3 == 0)//如果 i能被3整除
            sum = sum + i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

For 所控制的语句：

```
# include <stdio.h>

int main(void)
{
    int i;

    for (i=1; i<4; ++i)
    {
        printf("AAAA\n");
        printf("BBBB\n");
    }
    return 0;
}
```

在上图中，for 默认只能控制一个语句，但是如果要控制多个语句时候，那么需要使用{}把语句括起来。

求 1+1/2+1/3….1/100 的和

```
# include <stdio.h>

int main(void)
{
    int i;
    float sum = 0;

    for (i=1; i<=100; ++i)
    {
        sum = sum + 1 / (float)(i);
    }
    printf("sum = %f\n", sum);  //float必须用%f输出

    return 0;
```

在上图中，重点是强制数据类型转换也就是(float)(i)那句：

强制类型转化

格式：

(数据类型)(表达式)

功能：

把表达式的值强制转化为前面所执行的数据类型

例子：

(int)(4.5+2.2)　最终值是 6

(float)(5)　最终值是5.000000

如果把 print 那句换为下面这句会怎么样呢：

```
//sum = sum +(float)(1/i);
```

也是错的，为什么呢，因为 i 是整型，1 也是整型，所以不管你怎么转换也是整型啊，如果想要这样写的话，那么我们需要把 1 改成 1.0 也可以的。也就是：

```
//更简单的写法是: sum = sum + 1.0/i;
```

试数详细步骤举例：

```
1-> i=1 1<=100 成立
    sum=0+1/1.00=1   ++i   i = 2
2-> i=2 2<=100   成立
    sum=1+1/2.0      ++i   i = 3
3-> i=3 3<=100
    sum=1+1/2.0+1/3.0 ++i i=4
.................
.................
```

浮点数存取：

浮点数的存错所带来的问题

float和double都不能保证可以精确的存储一个小数

举例：

有一个浮点型变量x，　如何判断x的值是否是零

```
if (|x-0.000001| < 0.000001)
    是零
else
    不是零
```

为什么循环中更新的变量不能定义成浮点型

求 1-100 之间所有奇数的和：

```c
# include <stdio.h>

int main(void)
{
    int i;
    int sum = 0;

    for (i=1; i<101; ++i)
    {
        if (i%2 == 1)
            sum += i;// sum = sum + i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

求 1-100 之间的奇数的个数：

```c
# include <stdio.h>

int main(void)
{
    int i;
    int cnt = 0;  //个数一般用cnt表示

    for (i=1; i<101; ++i)
    {
        if (i%2 == 1)
            ++cnt;
    }
    printf("cnt = %d\n", cnt);
```

求 1-100 之间奇数的平均值：

```
    int i;
    int sum = 0;
    int cnt = 0;
    float avg; //average 的缩写

    for (i=1; i<101; ++i)
    {
        if (i%2 == 1)
        {
            sum += i;
            ++cnt;
        }
    }
    avg = 1.0*sum / cnt;  //1.0默认是double类型

    printf("sum = %d\n", sum);
    printf("cnt = %d\n", cnt);
    printf("avg = %f\n", avg);
```

求 1-100 之间的奇数之和，在求 1-100 之间的偶数之和：

```
# include <stdio.h>

int main(void)
{
    int i;
    int sum1 = 0;  //奇数和
    int sum2 = 0;  //偶数和

    for (i=1; i<101; ++i)
    {
        if (i%2 == 1)
        {
            sum1 += i;
        }
        else
        {
            sum2 += i;
        }
    }

   printf("奇数和 = %d\n", sum1);
   printf("偶数和 = %d\n", sum2);
```

多个 for 循环的嵌套使用：

多个for循环的嵌套使用

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

整体是两个语句， 1 2 3 是第一个语句

4 是第二个语句

整体是两个语句。

上图中，先执行 1，在执行 2，如果 2 成立，执行 4，在执行 5，如果 5 成立执行 A,在执行 6，在执行 5，如果 5 不成立，意味着里面的循环结束，然后执行 3，在执行 2，如果 2 成立又执行 4，在执行 5，如果 5 成立在执行 6，在执行 5，如果 5 不成立，在执行 3，在执行 2，如果 2 不成立，意味着本次循环结束，在执行 B，在上图中，需要注意的是，如果 2 成立的话，那么每次 4 都需要执行。

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

整体是一个语句

3.

```
for (7; 8; 9)
    for (1; 2; 3)
    {
        A;
        B;
        for (4; 5; 6)
            C;
    }
```

整体是一个语句

进制之间的转换：

如 234 为 5 进制，那么转换成 10 进制是多少：

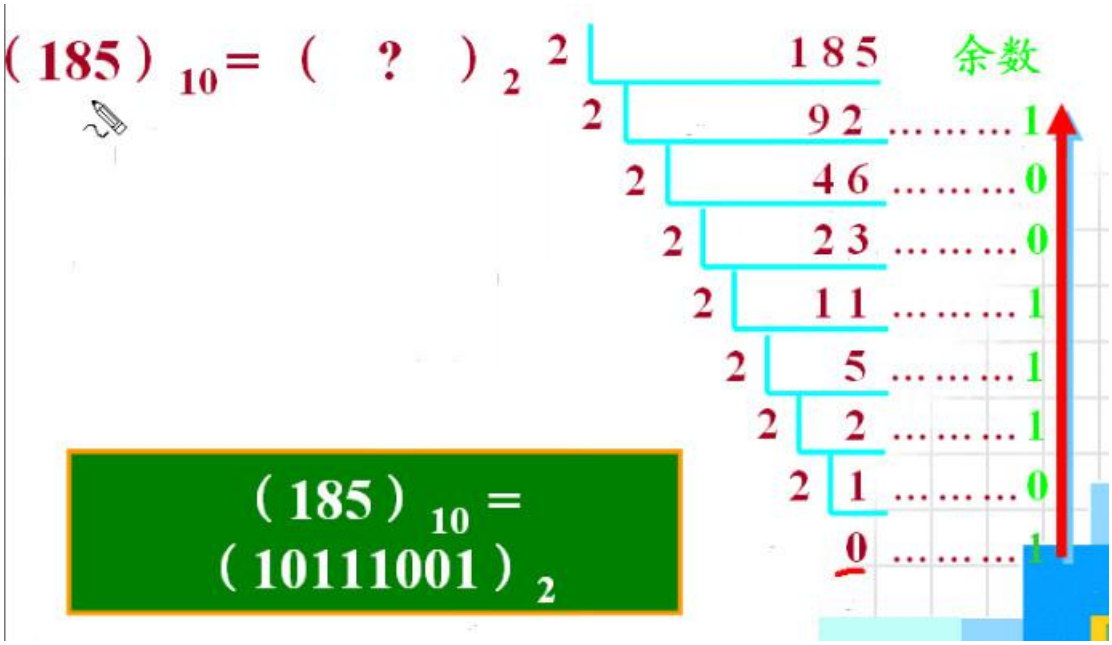
2x5x5+3x5+4 的值就是转换成的 10 进制。

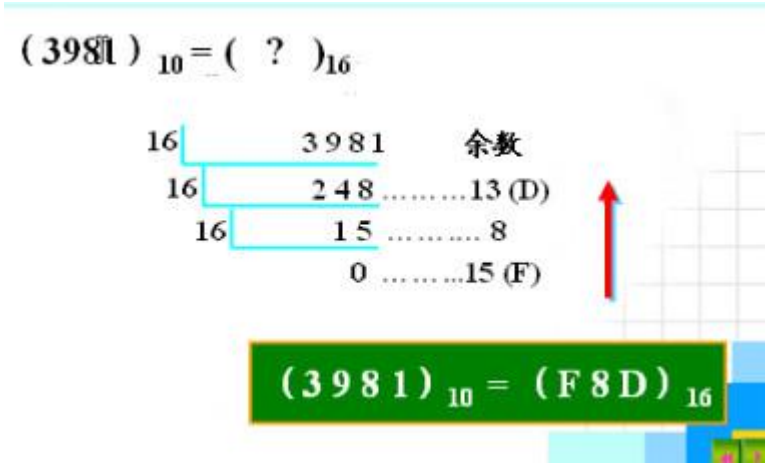
234e 是 16 进制，转换成 2 进制是多少：

2x16x16x16+3x16x16+4x16+12 的值就是转换成 10 进制的值。

注意上面的规律。

那么把十进制转换成 r 进制呢，其实很简单，就是把 10 进制数除以 r，直到商是 0 的时候。然后取余数，余数倒序排列：





琐碎的运算符：

自增：

自增[或者自减]

分类：

前自增　--　++i

后自增　--　i++

前自增和后自增的异同：

相同：

最终都使i的值加1

不同

前自增整体表达式的值是i加1之后的值

后自增整体表达式的值是i加1之前的值

```
# include <stdio.h>

int main(void)
{
    int i;
    int j;
    int k;
    int m;

    i = j = 3; //等价于 i = 3;  j = 3;
    k = i++;
    m = ++j;

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

    return 0;
}

/*
    在Vc++6.0中的输出结果是：
----------------------
i = 4, j = 4, k = 3, m = 4
----------------------
    总结：
        前自增整体表达式的值是i加1之后的值
        后自增整体表达式的值是i加1之前的值
*/
```

学习自增要明白的几个问题

1. 我们编程时应该尽量屏蔽掉前自增和后自增的差别
2. 自增表达式最好不要作为一个更大的表达式的一部分来使
   或者说
   i++ 和 ++i 单独成一个语句，不要把它作为一个完整复合语句的一部分来使用

自减：和自增一样。
三目运算符：

```
A ? B : C
等价于
    if (A)
        B;
    else
        C;
```

```c
# include <stdio.h>

int main(void)
{
    int i;

    i = (0>2 ? 5 : 1);
    printf("%d\n", i);

    return 0;
}
```

最终的输出结果是 1.
逗号表达式：

```
格式
    (A, B, C, D)
功能：
    从左到右执行
    最终表达式的值是最后一项的值
```

```c
# include <stdio.h>

int main(void)
{
    int i;

    i = (3, 2, 5, 6);
    printf("%d\n", i);

    return 0;
}
```

最终结果是 6.

```c
# include <stdio.h>

int main(void)
{
    int i;
    int j = 2;

    i = (j++, ++j, j+2, j-3);
    printf("%d\n", i);

    return 0;
}
```

上图中，逗号是个顺序点，即所有的副作用必须在下个语句前生效，其最后结果为 1，j+2 只是产生临时值，并没有把 j+2 的值赋个 j。如果写成 j+=2，那最后的值则变为 5.

For 的嵌套使用举例：

```c
# include <stdio.h>

int main(void)
{
    int i, j;

    for (i=0; i<3; ++i)
        for (j=2; j<5; ++j)
            printf("哈哈!\n");
        printf("嘻嘻!\n");

    return 0;
}
```

上例中输出的结果是 9 个哈哈，1 个嘻嘻。

```c
# include <stdio.h>

int main(void)
{
    int i, j;

    for (i=0; i<3; ++i)
        printf("嘿嘿!\n");
        for (j=2; j<5; ++j)
            printf("哈哈!\n");
        printf("嘻嘻!\n");

    return 0;
}
```

在上图中，整个程序分成 3 个语句，输出的结果是 3 个嘿嘿，3 个哈哈，1 个嘻嘻。

```
# include <stdio.h>

int main(void)
{
    int i, j;

    for (i=0; i<3; ++i)
    {
        printf("111!\n");
        for (j=2; j<5; ++j)
        {
            printf("222!\n");
            printf("333!\n");
        }
        printf("444!\n");
    }


    return 0;
```

其结果是：

```
111!
222!
333!
222!
333!
222!
333!
444!
111!
222!
333!
222!
333!
222!
333!
444!
111!
222!
333!
222!
333!
222!
333!
444!
```

While（先付钱后吃饭）

1：执行的顺序：

格式：

```
while (表达式)
    语句;
```

2：与 for 的相互比较：

用 for 来求 1-100 之和：

```
# include <stdio.h>

int main(void)
{
    int sum = 0;
    int i;

    for (i=1; i<101; ++i)
    {
        sum = sum + i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

用 while 实现 1-100 之和。只需要把 for 语句替换为：

```
i = 1;
while (i < 101)
{
    sum = sum + i;
    ++i;
}
```

For 和 while 是可以相互转换的，可以用下面的表达式来表示：

```
for (1; 2; 3)
    A;

等价于

1;
while (2)
{
    A;
    3;
}
```

While 和 for 在逻辑上完全等价，但是 for 在逻辑上更强。更容易理解，更不容易出错。推荐多使用 for。

3：while 举例：

从键盘输入一个数字，如果该数字是回文数，
则返回yes，否则返回no
回文数：正着写和倒着写都一样
比如：121　12321

```
# include <stdio.h>

int main(void)
{
    int val; //存放待判断的数字
    int m;
    int sum = 0;

    printf("请输入您需要判断的数字：  ");
    scanf("%d", &val);

    m = val;
    while (m)
    {
        sum = sum * 10 + m%10;
        m /= 10;
    }

    if (sum == val)
        printf("Yes!\n");
    else
        printf("No!\n");

    return 0;
}
```

试数：

```
1> m=1234 成立
   sum=0*10+1234%10=4
   m=m/10=123;
2> m=123 成立
   sum=4*10+123%10=43
   m=123/10=12
3> m=12 成立
   sum=43*10+12%10=432
   m=12/10=1
4> m=1
   sum=432*10+1%10=4321
   m=1/10=0
5> m=0  不成立

最终sum = 4321
```

通过上面的试数，应该能很快的理解回文数的算法。

```c
/*
    菲波拉契序列
    1  2  3  5  8  13  21  34
*/

# include <stdio.h>

int main(void)
{
    int n;
    int f1, f2, f3;
    int i;

    f1 = 1;
    f2 = 2;

    printf("请输入您需要求的想的序列: ");
    scanf("%d", &n);

    if (1 == n)
    {
        f3 = 1;
    }
    else if (2 == n)
    {
        f3 = 2;
    }
    else
    {
        for (i=3; i<=n; ++i)
        {
            f3 = f1 + f2;
            f1 = f2;
            f2 = f3;
        }

     }

     printf("%d\n", f3);

     return 0;
}
```

```
1> i=3 3<=6 成立
   f3=1+2=3  f1=f2=2 f2=f3=3 ++i i=4
2> i=4 4<=6 成立
   f3=2+3=5  f1=3 f2=5 i=5
3> i=5 5<=6 成立
   f3=3+5=8  f1=5 f2=8 i=6
4> i=6 6<=6 成立
   f3=5+8=13 f1=8 f2=13 i=7
5> i=7 7<=6 不成立
```

4：什么时候使用 while，什么时候使用 for：

没法说，用多了就自然而然知道了

Do...while（先吃饭后付钱）

格式

```
do
{
  .....
} while (表达式);
```

do...while.并不等价于for,当然也不等价于while

主要用于人机交互

一元二次方程：

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

int main(void)
{
    double a, b, c;
    double delta;
    double x1, x2;
    char ch;

    do
    {
        printf("请输入一元二次方程的三个系数:\n");
        printf("a = ");
        scanf("%lf", &a);

        printf("b = ");
        scanf("%lf", &b);

        printf("c = ");
        scanf("%lf", &c);

        delta = b*b - 4*a*c;

        if (delta > 0)
        {
            x1 = (-b + sqrt(delta)) / (2*a);
            x2 = (-b - sqrt(delta)) / (2*a);
            printf("有两个解, x1 = %lf, x2 = %lf\n", x1, x2);
        }
        else if (0 == delta)
        {
            x1 = x2 = (-b) / (2*a);
            printf("有唯一解, x1 = x2 = %lf\n", x1, x2);
        }
        else
        {
            printf("无实数解!\n");
        }

        printf("您想继续么(Y/N): ");
        scanf(" %c", &ch);  //%c前面必须得加一个空格 原因略
    } while ('y'==ch || 'Y'==ch);

    return 0;
}
```

Switch 的用法：

电梯程序：

```c
# include <stdio.h>

int main(void)
{
    int val;

    printf("请输入您要进入的楼层: ");
    scanf("%d", &val);

    switch (val)
    {
    case 1:
        printf("1层开!\n");
        break;
    case 2:
        printf("2层开!\n");
        //break;
    case 3:
        printf("3层开!\n");
        break;
    default:
        printf("没有盖到这一层!\n");
        break;
    }

    return 0;
}
```

Case 是程序的入口，当进入程序后，程序会从上往下执行，如果有 break，那么会中断程序，如果没有，那么会一直执行。

```
switch(表达式)
{
    case常量表达式1: 语句1;
    case常量表达式2: 语句2;
    ...
    case常量表达式n: 语句n;
    default      : 语句n+1;
}
```

Break 的用法：

break和continue

break

break如果用于循环是用来终止循环

break如果用于switch，则是用于终止switch

break不能直接用于if，除非if属于循环内部的一个子句

```
# include <stdio.h>

int main(void)
{
        int i;
/*
        switch (2)
        {
        case 2:
                printf("哈哈!\n");
                break;  //OK, break可以用于switch
        }

*/

        for (i=0; i<3; ++i )
        {
                if (3 > 2)
                        break;  //break虽然是if内部的语句，
                                //但break终止的确是外部的for循环
                printf("嘿嘿!\n");
        }

        return 0;
}
```

```
# include <stdio.h>

int main(void)
{
    int i, j;

    for (i=0; i<3; ++i)
    {
        for (j=1; j<4; ++j)
            break; //break只能终止距离它最近的循环
        printf("同志们好!\n");
    }
```

在多层循环中，Break 只能终止他最近的循环。

在多层 switch 中，break 也是只能终止距离他最近的 switch。

```c
# include <stdio.h>

int main()
{
        int x=1, y=0, a=0, b=0;
        switch(x) // 第一个switch
        {
        case 1:
                switch(y)  // 第二个switch
                {
                case 0:
                        a++;
                        break; //终止的是第二个switch
                case 1:
                        b++;
                        break;
                }
                b = 100;
                break;   //终止的是第一个switch
        case 2:
                a++;
                b++;
                break;
        }
        printf("%d %d\n", a, b); //26行

        return 0;
}
```

Break 只能用于循环和 switch，不能用于 if。如果用于 if，必须要当循环中嵌套 if 的时候。

Continue 的用法：

```
continue
    用于跳过本次循环余下的语句，
    转去判断是否需要执行下次循环
```

```
1.
for (1; 2; 3)
{
    A;
    B;
    continue; //如果执行该语句，则执行完该语句
    C;
    D;
}
```

上图中，如果执行 continue，那么 C,D 将不会被执行，会执行 3.

```
while (表达式)
{
    A;
    B;
    continue;
    C;
    D;
}
```

在上图中，如果执行了 continue，那么后面的 C,D 将不再执行，而会去执行表达式。

```
# include <stdio.h>

int main(void)
{
        int i;
        char ch;

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


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

        return 0;
}
```

数组：--非重点

数组的使用：

```
# include <stdio.h>

int main(void)
{
        int a[5] = {1, 2, 3, 4, 5};
                //a是数组的名字， 5表示数组元素的个数 ，
                //并且这5过热元素分别用a[0] a[1] ...a[4]
        int i;

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

        return 0;
}
```

为什么需要数组

1：为了解决大量同类型数据的存储和使用问题。

2：用数组可以模拟现实世界。

Int a[25]:一维数组，可以当做一个线性结构。

Int a[8][6]:可以当做一个平面，意思是 8 行 6 列。有 48 个元素。

Int a[3][4][5]:可以当做一个三维 立体。

Int a[3][4][5][6]:可以当做一个四维空间。

数组的分类

一维数组

怎样定义一维数组：

• 为 n 个变量分配存储空间：数组内存空间是连续的。

• 所有的变量类型必须相同：数组不可能第一个元素是整形，第二个元素是浮点型。

• 所有变量所占用的字节必须相等。

例子： int [5]

数组不是学习重点的原因？

数组一旦定义，其长度是死的。

有关一维数组的操作 --都需要自己另外编程序实现

而我们通常用第三方软件（工具）如数据库等方便直接地实现。

对数组的操作：

初始化 赋值 排序 求最大/小值 倒置 查找 插入 删除

• 初始化：

```
初始化
    完全初始化
        int a[5] = {1, 2, 3, 4, 5};
    不完全初始化，未被初始化的元素自动为零
        int a[5] = {1, 2, 3};
    不初始化，所有元素是垃圾值
        int a[5];
    清零
        int a[5] = {0};
```

```
错误写法：
    int a[5];
    a[5] = {1, 2, 3, 4, 5}; //错误
    只有在定义数组的同时才可以整体赋值，
    其他情况下整体赋值都是错误的

    int a[5] = {1, 2, 3, 4, 5};
    a[5] = 100; //error 因为没有a[5]这个元素，
```

上图中 a[5]前面如果没有加上数据类型，那么这里的 a[5]不是指一个数组，其中的 5 只的是下标。

```
int a[5] = {1, 2, 3, 4, 5};
int b[5];
如果要把a数组中的值全部复制给b数组
    错误的写法：
        b = a;// error
    正确的写法
        for (i=0; i<5; ++i)
            b[i] = a[i];
```

上图中，数组的 5 个元素不是用 a 来代表的，是用 a0,a1…a4 来代表的，所以说数组名 a 代表的不是数组的 5 个元素，数组名代表的是数组的第一个元素的地址。

- 赋值

```c
# include <stdio.h>

int main(void)
{
    int a[5];
    int i;

    scanf("%d", &a[0]);
    printf("a[0] = %d\n", a[0]);

    scanf("%d", &a[3]);
    printf("a[3] = %d\n", a[3]);

    for (i=0; i<5; ++i)
        printf("%d ", a[i]);

    return 0;
}
```

把一个数组元素给全部倒过来：

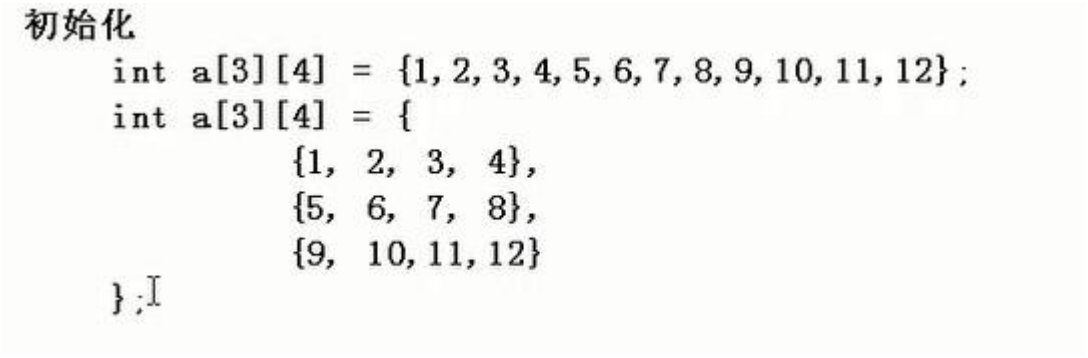
```c
# include <stdio.h>

int main(void)
{
        int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
        int i, j;
        int t;

        i = 0;
        j = 7;
        while (i < j)
        {
                t = a[i];
                a[i] = a[j];
                a[j] = t;

                i++;
                --j;
        }

        for (i=0; i<8; ++i)
                printf("%d\n", a[i]);

        return 0;

}
```

- 排序

- 求最大/小值
- 倒置
- 查找
- 插入
- 删除

二维数组：

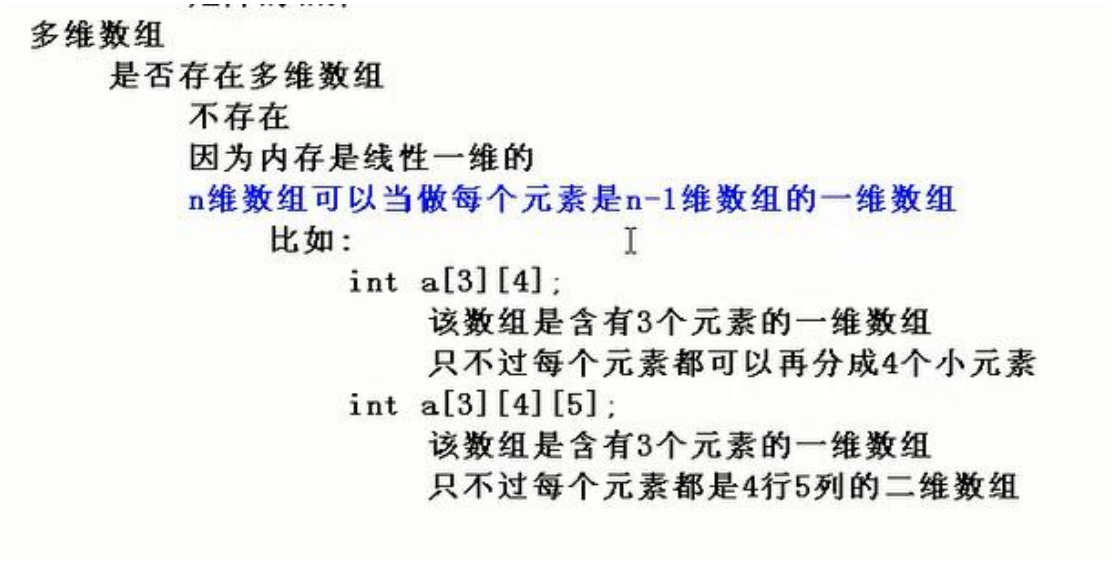
二维数组的初始化：



输出二维数组内容：

```
# include <stdio.h>

int main(void)
{
    int a[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    int i, j;

    //输出数组内容
    for (i=0; i<3; ++i)
    {
        for (j=0; j<4; ++j)
            printf("%d  ", a[i][j]);
        printf("\n");
    }


    return 0;
```

多维数组:
是否存在多维数组：不存在
因为内存是线性一维的，在内存中是不分行不分列的。
N 维数组可以当做每个元素是 n-1 维数组的 一维数组。



函数（第二个重点）：

```c
# include <stdio.h>


//max是函数的名字，i和j是形式参数，简称形参 void表示函数没有返回值
void max(int i, int j)
{
        if (i > j)
                printf("%d\n", i);
        else
                printf("%d\n", j);
}

int main(void)
{
        int a, b, c, d, e, f;

        a = 1, b = 2; c = 3, d = 9, e = -5, f = 100;
        max(a, b);
        max(c, d);
        max(e, f);
/*
        if (a > b)
                printf("%d\n", a);
        else
                printf("%d\n", b);

        if (c > d)
                printf("%d\n", c);
        else
                printf("%d\n", d);

        if (e > f)
                printf("%d\n", e);
        else
                printf("%d\n", f);
*/
```

为什么需要函数：

- 避免了重复性操作。
- 有利于程序的模块化。

（自上而下，逐步细化，大问题分解成小问题）

用它作为参照，可以对比 JAVA 和 C#面向对象的思想。

C 语言基本单位是函数，C#、C++和 JAVA 基本单位是类。

什么叫做函数

```
# include <stdio.h>

int f(void)  //括号中的void表示该函数不能接受数据  int表示函数返回值是int类型的数据
{
        return 10; //向主调函数返回10
}

void g(void)  //8行  函数名前面的void表示该函数没有返回值
{
//      return 10; //error 与8行行首的void相矛盾
}

int main(void)
{
        int j = 88;

        j = f();
        printf("%d\n", j);

//      j = g(); //error 因为g函数没有返回值

        return 0;
}
```

- 逻辑上：能够完成特定功能的独立的代码块。
  物理上：能够接收数据【也可以不接收数据】，能够对接收的数据进行处理【也可以不对数据进行处理】，能够将数据处理的结果返【也可以没有返回值】。
- 总结：函数是个工具，他是为了解决大量类似问题而设计的，函数可以当做黑匣子（内部原理不用管）。

如何定义函数

- 函数的返回值，函数的名字（函数的形参列表）

```
{
        函数的执行体
}
```

- 函数定义的本质：详细描述函数之所以能够实现某个特定功能的具体方法。

函数中的变量叫做形参；数组中的变量叫元素。

一旦函数执行完毕，其内部的形参所占空间就被收回。

- return 表达式的含义：

Return 是终止被调函数，向主调函数返回表达式的值，如果表达式为空，则只终止函数，不向被主函数返回任何值。

例子：

```
void f()
{
    return;  //return只用来终止函数，不向主调函数返回任何值
}

int f()
{
    return 10;  //第一：  终止函数，  第二：向主调函数返回10
}
```

Break 是用来终止（就近的）循环和 switch 语句。而 return 是用来终止被调函数的。

```c
# include <stdio.h>

void f(void)
{
        int i;

        for (i=0; i<5; ++i)
        {
                printf("大家辛苦了!\n");
                return;
        }
        printf("同志们好!\n");
}

int main(void)
{
        f();

        return 0;
}
```

• 函数返回值的类型，也称为函数的类型，因为如果函数名前的返回值类型和函数执行体中的 return 表达式中表达式的类型不同的话，则最终函数返回值的类型以函数名前的返回值类型为准。

例：

```c
# include <stdio.h>

int f()
{
        return 10.5;   //因为函数的返回值类型是int
                       //所以最终f返回的是10而不是10.5
}

int main(void)
{
        int i = 99;
        double x = 6.6;

        x = f();
        printf("%lf\n", x);

        return 0;
}
```

在上图中，函数的返回值以函数前的数值类型为准。

函数的分类

• 有参函数和无参函数。

• 有返回值和无返回值。

• 库函数和用户自定义函数。
• 普通函数和主函数（main 函数）
1：一个程序有且只有一个主函数。
2：主函数可以调用普通函数，普通不能调用主函数。
3：普通函数可以相互调用。
4：主函数是程序的入口，也是函数的出口。
5：值传递函数和地址传递函数。

```
# include <stdio.h>


//max是函数的名字, i和j是形式参数，简称形参 void表示函数没有返回值
void max1(int i, int j)
{
        if (i > j)
                printf("%d\n", i);
        else
                printf("%d\n", j);
}

int max2(int i, int j)
{
        if (i > j)
                return i;
        else
                return j;
}

int main(void)
{
        int a, b, c, d, e, f;
        a = 1, b = 2; c = 3, d = 9, e = -5, f = 100;

        printf("%d\n", max2(a, b));
        printf("%d\n", max2(c, d));
        printf("%d\n", max2(e, f));

/*
        max1(a, b);
        max1(c, d);
        max1(e, f);
*/
        return 0;
}
```

判断一个数是否是素数：

```c
# include <stdio.h>

int main(void)
{
    int val;
    int i;

    scanf("%d", &val);
    for (i=2; i<val; ++i)
    {
        if (val%i == 0)
            break;
    }
    if (i == val)
        printf("Yes!\n");
    else
        printf("No!\n");

    return 0;
}
```

使用函数判断一个数是否是素数：

```
/*
        2009年11月4日11:18:51
        判断一个数字是否是素数
        用单独的函数来实现，代码的可重用性提高
*/

# include <stdio.h>

bool IsPrime(int val)
{
        int i;

        for (i=2; i<val; ++i)
        {
                if (0 == val%i)
                        break;
        }
        if (i == val)
                return true;
        else
                return false;
}

int main(void)
{
        int val;
        int i;

        scanf("%d", &val);
        if ( IsPrime(val) )
                printf("Yes!\n");
        else
                printf("No!\n");

        return 0;
}
```

函数和程序的调用应该注意的地方：

```c
# include <stdio.h>

int f(int i)
{
    return 10;
}

int main(void)
{
    int i = 99;

    printf("%d\n", i);
    i = f(5);
    printf("%d\n", i);

    return 0;
}
```

函数的声明：

```c
# include <stdio.h>

void f(void)
{
    printf("哈哈!\n");
}

int main(void)
{
    f();

    return 0;
}
```

当函数没有返回值时，那么规范的写法是要在函数中写明 void 的。
在上图中，第一个 void 表示没有返回值，而第二个 void 表示不接收形参，也就是函数不接收数据。
如果想把函数写在程序的后面，那么需要写函数声明：

```c
# include <stdio.h>

void f(void); //函数声明， 分号不能丢掉

int main(void)
{
    f();

    return 0;
}

void f(void)
{
    printf("哈哈!\n");
}
```

函数声明的含义是告诉编译器 f()是个函数名。如果不加函数声明，那么编译器在编译到 f 的时候，不知道 f 是个什么，如果加了函数声明，那么编译器编译到 f 的时候，就知道 f 是个函数。

• 需要注意的是，调用语句需要放在定义语句的后面，也就是说，定义函数的语句要放在调用语句的前面。

```c
/*
        2009年11月4日11:01:35
        一定要明白该程序为什么是错误的
        一定要明白该程序第9行生效之后程序为什么就正确了
*/

# include <stdio.h>

//void f(void);  //9行

void g(void)
{
        f();  //因为函数f的定义放在了调用f语句的后面，所有语法出错
}

void f(void)
{
        printf("哈哈!\n");
}

int main(void)
{
        g();

        return 0;
}
```

如果函数调用写在了函数定义的前面，则必须加函数前置声明，函数前置声明的作用是：

1：告诉编译器即将可能出现的若干个字母代表的是一个函数。“打招呼”

2：告诉编译器即将可能出现的若干个字母所代表的函数的形参和返回值的具体情况。

3：函数声明必须是一个语句，也就是在函数声明后需加分号。
4：对库函数的声明也就是系统函数。是通过#include<库函数所在的文件的名字.h>来实现的。如 stdio.h

形参和实参要求：

1：形参和实参个数是一一对应的。
2：形参和实参的位置也是一一对应的。
3:形参和实参的数据类型需要相互兼容。

• 如何在软件开发中合理的设计函数来解决实际问题。

求 1 到某个数字之间的数是否是素数，并将他输出：

合理设计函数 1

```
/*
        2009年11月4日11:18:51
        判断一个数字是否是素数
        只用一个函数实现,不好，代码的利用率不高
*/

# include <stdio.h>

int main(void)
{
        int val;
        int i;

        scanf("%d", &val);
        for (i=2; i<val; ++i)
        {
                if (0 == val%i)
                        break;
        }
        if (i == val)
                printf("Yes!\n");
        else
                printf("No!\n");

        return 0;
}
```

合理设计函数 2：

```
/*
        2009年11月4日11:18:51
        判断一个数字是否是素数
        用单独的函数来实现，代码的可重用性提高
*/

# include <stdio.h>

bool IsPrime(int val)
{
        int i;

        for (i=2; i<val; ++i)
        {
                if (0 == val%i)
                        break;
        }
        if (i == val)
                return true;
        else
                return false;
}

int main(void)
{
        int val;
        int i;

        scanf("%d", &val);
        if ( IsPrime(val) )
                printf("Yes!\n");
        else
                printf("No!\n");

        return 0;
}
```

合理设计函数 3：

```
/*
        2009年11月4日11:18:51
        求1到某个数字之间(包括该数字)所有的素数，并将其输出
        只用main函数实现，有局限性：
                1. 代码的重用性不高
                2. 代码不容易理解
*/

# include <stdio.h>

int main(void)
{
        int val;
        int i;
        int j;

        scanf("%d", &val);
        for (i=2; i<=val; ++i)
        {
                //判断i是否是素数，是输出，不是不输出
                for (j=2; j<i; ++j)
                {
                        if (0 == i%j)
                                break;
                }
                if (j == i)
                        printf("%d\n", i);
        }

        return 0;
}
```

合理的设计函数 4：

```
        2009年11月4日11:18:51
        求1到某个数字之间(包括该数字)所有的素数，并将其输出
        用1个函数来判断一个数字是否是素数
                优点:
                        代码比   如何设计函数_3.cpp   更容易理解
                        代码的可重用性比   如何设计函数_3.cpp 高
                缺点:
                        可重用性仍然不是非常高，
                        比如有1000个数字，求它们每个数字从1到它本身的素数
                        则
                                        for (i=2; i<=val; ++i)
                                        {
                                                if ( IsPrime(i) )
                                                        printf("%d\n", i);
                                        }
                        要写1000次

*/

# include <stdio.h>

bool IsPrime(int m)
{
        int i;

        for (i=2; i<m; ++i)
        {
                if (0 == m%i)
                        break;
        }
        if (i == m)
                return true;
        else
                return false;

}

int main(void)
{
        int val;
        int i;

        scanf("%d", &val);
        for (i=2; i<=val; ++i)
        {
                if ( IsPrime(i) )
                        printf("%d\n", i);
        }

        return 0;
}
```

合理设计函数 5：

```
/*
        2009年11月4日11:56:29
        用两个函数来实现求1到某个数字之间所有的素数，并将其输出
        本程序 和 如何合理设计函数_4.cpp 相比较
        代码量更少，可重用性更高
*/


# include <stdio.h>

//本函数的功能是：判断m是否是素数，是返回true，不是返回false
bool IsPrime(int m)
{
        int i;

        for (i=2; i<m; ++i)
        {
                if (0 == m%i)
                        break;
        }
        if (i == m)
                return true;
        else
                return false;

}

//本函数的功能是把1到n之间所有的素数在显示器上输出
void TraverseVal(int n)
{
        int i;

        for (i=2; i<=n; ++i)
        {
                if ( IsPrime(i) )
                        printf("%d\n", i);
        }
}

int main(void)
{
        int val;

        scanf("%d", &val);
        TraverseVal(val);

        return 0;
}
```

常用的系统函数和如何通过书籍来学习函数：

Turboc2.0 实用大全—机械工业出版社

常用的系统函数

```
double sqrt(double x);
    求的x的平方根
int abs(int x)
    求x的绝对值
double fabs(double x)
    求x的绝对值
```

递归：（略）

栈：相当于一个杯子（容器）

变量的作用域和存储方式：

```
变量的作用域和存储方式:
    按作用域分:
        全局变量

        局部变量

    按变量的存储方式
        静态变量
        自动变量
        寄存器变量
```

全局变量和局部变量：

局部变量：

```
局部变量
    在一个函数内部定义的变量或者函数的形参 都统称为局部变量
        void f(int i)
        {
            int j = 20;
        }
        i和j都属于局部变量
```

局部变量的使用范围只能在本函数内部使用。

全部变量：

```
全局变量
    在所有函数外部定义的变量叫全局变量
    全局变量使用范围: 从定义位置开始到整个程序结束
```

```
/*
        2009年11月6日9:33:02
        一定要明白该程序为什么是错的，
        也要明白把9到12行代码放在14行后面，为什么程
序就OK了
*/

# include <stdio.h>

void g()  //9
{
        printf("k = %d\n", k);
} //12

int k = 1000;  //14行

void f(void)
{
        g();
        printf("k = %d\n", k);
}

int main(void)
{
        f();
        return 0;
}
```

全局变量和局部变量命名冲突的问题：
在同一个范围之内不能定义两个一样的局部变量：

```
# include <stdio.h>


void f(int i)
{
    int i;
    printf("i = %d\n", i);
}

int main(void)
{
    f(8);
```

在一个函数内部，如果定义的局部函数的名字和全局变量名一样时，局部变量会屏蔽掉全局变量：

```c
# include <stdio.h>

int i = 99;

void f(int i)
{
    printf("i = %d\n", i);
}

int main(void)
{
    f(8);

    return 0;
}
```

上例中最终的输出结果是 8，因为局部变量把全局变量给屏蔽掉了。

指针：（C 语言的灵魂）

内存的存储是以一个字节为一个编号，也就是 8 位合在一起给一个编号，不是 0,1 就给编号。

内存分为很多个单元，每个单元就会分配一个编号。

地址：内存单元的一个编号。而指针和地址一个概念的。也就是说指针就是地址。

```c
# include <stdio.h>

int main(void)
{
        int * p; //p是变量的名字, int * 表示p变量存放的是int类型变量的地址
        int i = 3;

        p = &i;  //OK
        //p = i; //error, 因为类型不一致, p只能存放int类型变量的地址, 不能存
        //放int类型变量的值
        //p = 55; //error 原因同上

        return 0;
}
```

普通变量：只能存放一个值。

指针变量：同样是一个变量，但是指针变量存放其他变量的地址。

```c
# include <stdio.h>

int main(void)
{
	int * p; //p是变量的名字, int * 表示p变量存放的是int类型变量的地址
			 //int * p; 不表示定义了一个名字叫做*p的变量
			// int * p; 应该这样理解: p是变量名, p变量的数据类型是 int *类型
			//          所谓int * 类型 实际就是存放int变量地址的类型
	int i = 3;
	int j;

	p = &i;
		/*
			1. p保存了i的地址, 因此p指向i
			2. p不是i, i也不是p, 更准确的说:  修改p的值不影响i的值,
			修改i的值也不会影响p的值
			3. 如果一个指针变量指向了某个普通变量, 则
					*指针变量  就完全等同于  普通变量
			   例子:
					如果p是个指针变量, 并且p存放了普通变量i的地址
					则p指向了普通变量i
					*p      就完全等同于  i
					或者说:  在所有出现*p的地方都可以替换成i
							在所有出现i的地方都可以替换成*p

				*p 就是以p的内容为地址的变量
		*/
	j = *p;  //等价于 j = i;
	printf("i = %d, j = %d\n", i, j);

	return 0;
}
```

*p 代表的是 p 所指向的那个变量。在上图中*p 和 i 是同一个东西，但是*p 和 p 不是同一个东西。

在上图中，int * p 是一个声明，开头的 int * 是他的数据类型。P 是变量的名字。不能理解我定义了一个整形变量，这个整形变量的名字叫做*p。所谓 int *类型，实际就是存放 int 变量地址的类型。

*p 代表的是以 p 的内容为地址的变量。

解析：p 的内容是一个地址，在上图中，p 的内容就是 i 的地址，*p 其指向的变量当然就是 i 变量了。

指针和指针变量：

指针就是地址，地址就是指针。

地址就是内存单元的编号。

指针变量：存放地址的变量。而指针只是一个值，这个值是内存单元的一个编号。

指针变量才是一个变量，他里面才可以存放数据。

指针和指针变量是两个不同的概念，但是需要注意的是，通常我们在叙述时会把指针变量简称为指针，实际他们含义并不一样。

指针的重要性：

指针：

表示一些复杂的数据结构
快速的传递数据
使函数返回一个以上的值
能直接访问硬件
能够方便的处理字符串
是理解面向对象语言中引用的基础

总结：指针是C语言的灵魂

指针的分类：

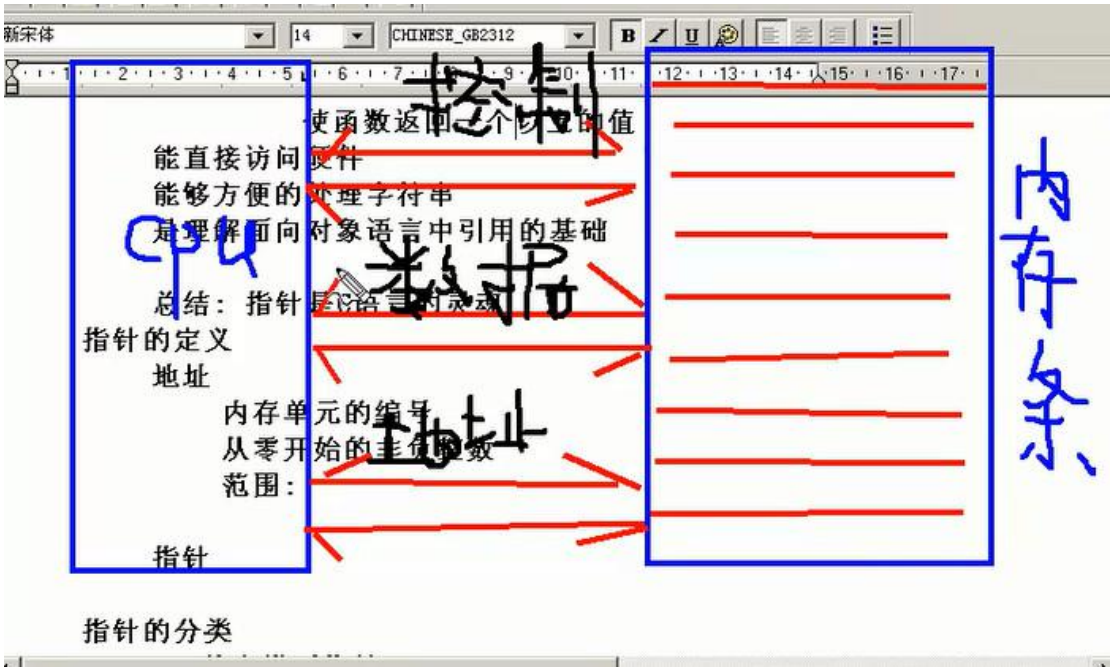
指针的分类

1. 基本类型指针
2. 指针和数组
3. 指针和函数
4. 指针和结构体
5. 多级指针

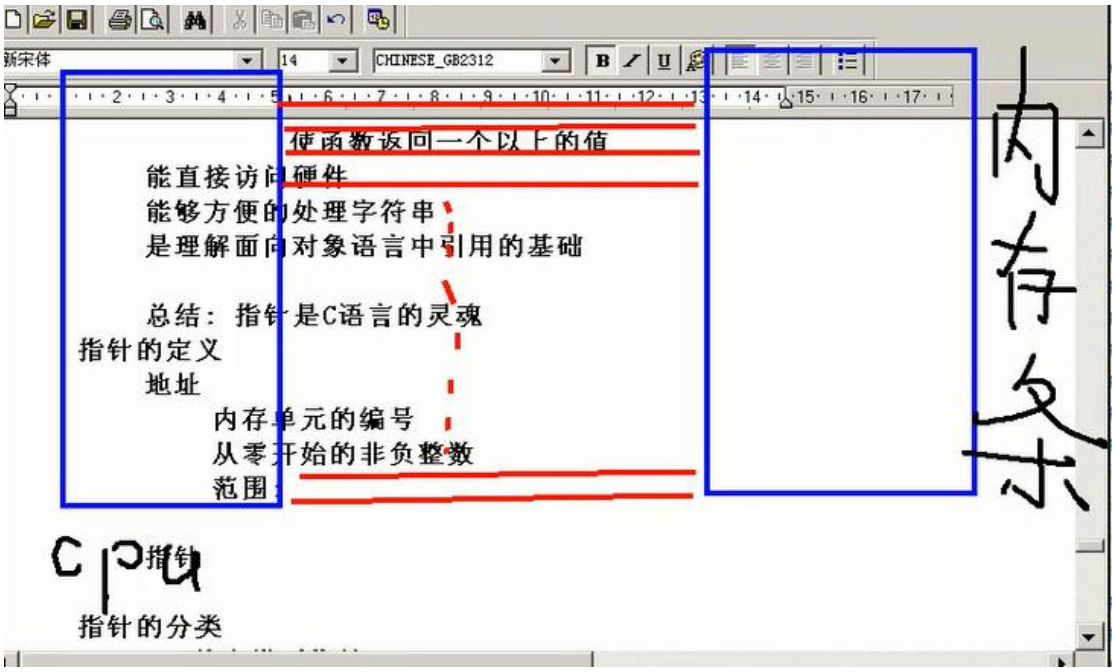
指针的定义：

• 地址：内存单元的编号，是一个从 0 开始的非负整数。

范围：cpu 对内存是通过控制、数据、地址三条总线来进行控制的。



控制：cup 会先把内存中的数据读入，进行处理后，在返回给内存，然后内存在把数据写入硬盘。
数据：用于数据的传输，不管是把内存中的数据发送给 cpu，还是把 cpu 的数据写如内存条，都是由数据线来完成的，但是数据传输的方向则是由控制线来控制的。
地址：地址线则是确定数据要写入内存中的那个单元。
所谓的一个单元就是一个字节。



一条地址总线能控制 2 的 1 次方，一般的机器有 32 个地址线，最终能够控制 2 的 32 个单元，而每个单元是八位，而最终我们的内存能够存储 2 的 32 次方*8 位。
则换算为 G 的话，最终大小为 4G.那么地址总线的范围则是 4G 大。
指针：指针就是地址，地址就是指针。
指针变量就是存放内存单元编号的变量。
指针变量和指针是两个不同的概念。
指针的本质就是一个操作受限的非负整数。
指针不能进行算术运算-相加 乘 除。但是能相减。
如果两个指针变量指向的是同一块连续空间的不同存储单元，则这两个指针变量才可以相减。类似于同一个小区同一楼层六牌号相减表示两房间隔。这时才有现实意义。
基本类型的指针：

```c
# include <stdio.h>

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

    *p = i;
    printf("%d\n", *p);

    return 0;
}
```

Int *p：p 只能存放 int 类型的地址。

P = & i：把 i 的地址赋给 p。然后 p 就指向了 i，*p 就等于 i。其实就是

1：该语句保存了 i 的地址。

2：p 保存了 i 的地址，所以 p 指向 i。

3：p 既然指向 i，*p 就是 i。

*p：表示以 p 的内容为地址的变量。

*p：p 是有指向的，p 里面是个垃圾值，*p 则是说以 p 的内容为地址的变量。因为不知道 p 的值是多少，所以不知道*p 到底代表的是那个变量。而*p = i，i=5，最终的结果就是把 5 赋给了一个所不知道的单元。

```c
# include <stdio.h>

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

        p = &i;
        //*q = p; //error 语法编译会出错
        //*q = *p;  //error
        p = q;  //q是垃圾值，q赋给p，p也变成垃圾值
        printf("%d\n", *q);    //13行
                /*
                        q的空间是属于本程序的，所以本程序可以读写q的内容，
                        但是如果q内部是垃圾值，则本程序不能读写*q的内容
                        因为此时*q所代表的内存单元的控制权限并没有分配给本程序
                        所以本程序运行到13行时就会立即出错
                */
        return 0;
}
```

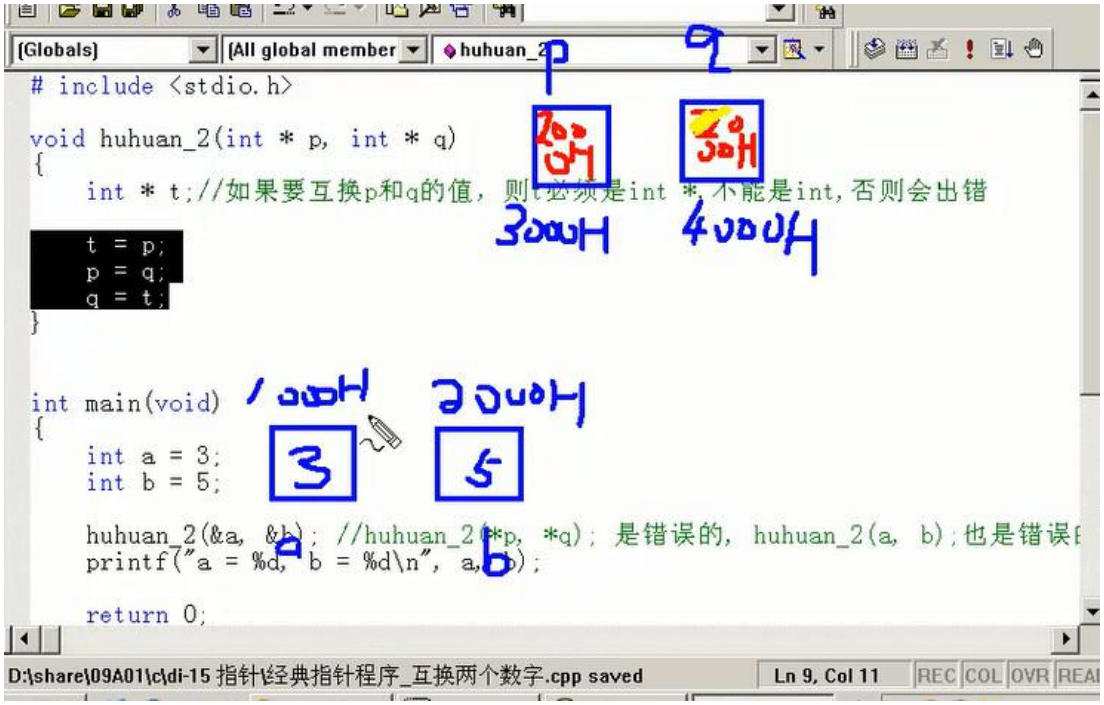
上图中，第一个 error 是数据类型不符合，不能相互转换。*q 代表的是整形，因为*q 代表的是以 q 的地址为内容的变量。而 p 是地址（int *）类型。第二个 error 同样有错，因为 q 没有赋值。

经典指针程序-互换两个数字：

1：先用函数来互换：

```c
# include <stdio.h>

void huhuan(int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;

    return;
}

int main(void)
{
    int a  = 3;
    int b = 5;

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

最终的输出结果没有互换，因为函数的 a，b 已经执行完成，分配给内存的空间已经释放了，所以最终 a，b 的值还是主函数 a，b 的值。互换的是形参的 a，b。和主函数没有关系。



在上图中，输出的值也是没有互换的，输出的同样是 3,5，需要注意的是，互换的只是 p、q 的内容，局部函数变化了，但是主函数是没有变化的。

最终正确的程序：

```c
# include <stdio.h>

void huhuan_1(int , int);
void huhuan_2(int *, int *);
void huhuan_3(int *, int *);

int main(void)
{
	int a = 3;
	int b = 5;

	huhuan_3(&a, &b); //huhuan_2(*p, *q); 是错误的,
			  //huhuan_2(a, b);也是错误的
	printf("a = %d, b = %d\n", a, b);

	return 0;
}

//不能完成互换功能
void huhuan_1(int a, int b)
{
	int t;

	t = a;
	a = b;
	b = t;

	return;
}

//不能完成互换功能
void huhuan_2(int * p, int * q)
{
	int * t;//如果要互换p和q的值，则t必须是int *,
		//不能是int,否则会出错

	t = p;
	p = q;
	q = t;
}

//可以完成互换功能
void huhuan_3(int * p, int * q)
{
	int t; //如果要互换*p和*q的值，则t必须定义成int,
		//不能定义成int *, 否则语法出错

	t = *p;  //p是int *,  *p是int
	*p = *q;
	*q = t;
}
```

*号的三种含义：

1：乘法

2：定义指针变量。Int　* p，定义了一个名字叫 p 的变量，int *表示 p 只能存放 int 变量的地址。

3：指针运算符。该运算符是放在已经定义好的指针变量的前面。如果 p 是一个已经定义好的指针变量，则* P 表示以 p 的内容为地址的变量。

注意理解形参，实参，和局部变量的关系。

```
# include <stdio.h>

int main(void)
{
    int * p;//等价于 int *p; 也等价于 int* p;
    int i = 5;
    char ch = 'A';

    p = &i;  // *p 以p的内容为地址的变量
    *p = 99;
    printf("i = %d, *p = %d\n", i, *p);

    //p = &ch;
    //p = ch; //error
    //p = 5;  //error

    return 0;
```

指针可以是函数返回一个以上的值：

```
# include <stdio.h>

int f(int i, int j)
{
    return 100;
//  return 88;
}


int main(void)
{
    int a = 3, b = 5;

    a = f(a, b);
    b = f(a, b);
```

不使用指针的话，只能使用用 return 来返回一个值。

如何通过被调函数修改主调函数普通变量的值

如何通过被调函数修改主调函数普通变量的值

1. 实参必须为该普通变量的地址
2. 形参必须为指针变量
3. 在被调函数中通过

*形参名 = ...

的方式就可以修改主调函数相关变量的值

```
void g(int * p, int * q)
{
    *p = 1;
    *q = 2;
}

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

指针和数组：

指针和一维数组：

（数组名　下标与指针关系 指针变量的运算）

• 一维数组名：一维数组名是个指针常量，他存放的是一维数组第一个元素的地址。

```
# include <stdio.h>

int main(void)
{
        int a[5]; //a是数组名  5是数组元素的个数 元素就是变量  a[0]  -- a[4]
//      int a[3][4]; //3行4列 a[0][0]是第一个元素 a[i][j]第i+1行j+1列
        int b[5];

        //a = b;//error a是常量
        printf("%#X\n", &a[0]);
        printf("%#X\n", a);

        return 0;
}
/*
        在Vc++6.0中的输出结果是:
--------------
0X12FF6C
0X12FF6C
Press any key to continue
--------------
        总结:
                一维数组名
                                        一维数组名是个指针常量
                                        它存放的是一维数组第一个元素的地址

*/
```

常量是不能被改变的，也就是说，一维数组名是不能被改变的。

数组名 a 存放的是一维数组第一个元素的地址，也就是
a = &a。
printf(“%#X\n”,&a[0]); ===== printf(“%#X\n”,a);
指针和二维数组：
下标和指针的关系：
• 如果 p 是个指针变量，则 p[i]永远等价于*(p+i)

```
# include <stdio.h>

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

    for (i=0; i<5; ++i)
        printf("%d\n", a[i]);  //a[i] == *(a+i)

    return 0;
}
```

确定一个一维数组需要几个参数，【如果一个函数要处理一个一维数组，则形参需要接收该数组的哪些信息。】
确定一个一维数组需要两个参数，
1：数组名，从数组的名称就可以知道数组的第一个值，因为一维数组的名称就是数组的第一个元素的地址。
2：是数组的个数。来计算该数组有多少个值。
区别于 字符串（只需要一个参数—首地址）
因为字符串默认其后面都有一个“/0”作为结束标志。而数组并没有相关约定。

```
# include <stdio.h>

//f函数可以输出任何一个一维数组的内容
void f(int * pArr, int len)
{
    *pArr
    *(pArr+1)
}

int main(void)
{
    int a[5] = {1, 2, 3, 4, 5};
    int b[6] = {-1, -2, -3, 4, 5, -6};
    int c[100] = {1, 99, 22, 33};

    f(a, 5);  //a是 int *

    return 0;
}
```

在上图中，a 是个指针变量，所以上面局部函数 f 的 pArr 则要定义成指针函数才可以，而 len 则是 int 类型。代表接收的是整型的数字。

```
   1# include <stdio.h>

//f函数可以输出任何一个一维数组的内容
void f(int * pArr, int len)
{
        int i;

        for (i=0; i<len; ++i)
                printf("%d  ", *(pArr+i) );  //*pArr *(pArr+1) *(pArr+2)
        printf("\n");
}

int main(void)
{
        int a[5] = {1,2,3,4,5};
        int b[6] = {-1,-2,-3,4,5,-6};
        int c[100] = {1, 99, 22, 33};

        f(a, 5);  //a是 int *
        f(b, 6);
        f(c, 100);

        return 0;
}

/*
        2009年11月14日10:45:51
        一定要明白 10行的pArr[3] 和17行 19行的a[3] 是同一个变量
*/

# include <stdio.h>

void f(int * pArr, int len)
{
        pArr[3] = 88;  //10行
}

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

        printf("%d\n", a[3]);  //17行
        f(a, 6);
        printf("%d\n", a[3]);  //  19行

        return 0;
}
/*
        在Vc++6.0中的输出结果是:
----------------------------
4
88
Press any key to continue
----------------------------
*/
```

在上图中因为数组 a 的名称代表的是 a 的第一个元素的地址，所以在函数 f 中所定义的指针变量 pArr 和 a 是相同的，因为 a 也是指针类型。也就是说 pArr=a=a[0]，

pArr[1]=a[1]=*(pArr+1)=*(a+1),pArr[2]=a[2]=*(pArr+2) =*(a+2).所以在 f 函数中 pArr[3]=a[3],所以第二个 printf 输出的结果是 88.
总结：pArr[i] = a[i] = *(pArr+i) = *(a+i)
在没有学习指针时，可将 a[3]认为是数组中第 4 个元素，但现在应该对其内部原理有更深刻认识。
这里下标也当成指针了，从首元素开始向后移动 3 个，即指向第 4 个元素。

```
# include <stdio.h>

void f(int * pArr, int len)
{
        int i;
        for (i=0; i<len; ++i)
                printf("%d  ",  pArr[i]);  //*(pArr+i) 等价于 pArr[i]
                                           //也等价于 b[i] 也等价于 *(b+i)

        printf("\n");
}

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

        f(b, 6);

        b[i]

        return 0;
}
```

在上图中因为数组 a 的名称代表的是 a 的第一个元素的地址，所以在函数 f 中所定义的指针变量 pArr 和 a 是相同的，因为 a 也是指针变量类型。也就是说 pArr=a=a[0]，pArr[1]=a[1]=*(pArr+1)=*(a+1),pArr[2]=a[2]=*(pArr+2) =*(a+2).
通过上图，我们知道，我们在 f 函数中修改数组的值，相当于修改主函数中相对应的值。

何谓变量地址 / 一个指针占几个字节

Sizeof(变量名/数据类型) 其返回值就是该变量或数据类型所占字节数。
一个指针变量无论其指向变量占几个字节，其本身所占大小都是 4 字节。
*p 具体指向几个字节，要靠前面类型确定，如果为 int 则为 4 字节，如果 double 则占 8 字节。
CPU 与 内存 交互时 有 32 根线，每根线只能是 1 或 0 两个状态，所有总共有 2^{32} 个状态。
1 个状态 对应 一个单元。如全为 0 全为 1 等。
内存中第一个单元，即 32 根线状态全为 0。
0000 0000 0000 0000 0000 0000 0000 0000
其大小为 4 字节
所有每个地址（硬件所能访问）的用 4 个字节保存（而不是一 位 bit）
一个变量的地址—用该变量首字节的地址表示。这也就是为什么指针变量始终只占 4 字节的原因。

接下来是：138 课 动态分配内存（很重要）

专题：138讲

动态内存分配（所有高级语言，没有C里深刻，对JAVA、C#理解有益）

传统数组的缺点：

1.数组长度必须事先指定，而且只能是常整数，不能是变量

例子　int a[5]; //必须事先指定，而且只能是常整数

int len = 5; int a[len];//error

2.传统形式定义的数组，该数组的内存程序员无法手动释放

数组一旦定义，系统为数组分配的内存空间就会一直存在，除非数组所在的函数运行终止。

在一个函数运行期间，系统为该函数中的数组分配的空间会一直存在。直到该函数运行完毕时，数组的空间才会被系统自动释放（不是清零）。

例子：void f(void){ int a[5]={1,2,3,4,5};....}

//数组a 占20个字节的内存空间，程序员无法手动编程释放它，数组a只能在f()函数结束被系统释放

3. 数组的长度一旦定义，数组长度就不能再更改。

数组的长度不能在函数运行的过程中动态的扩充或缩小

4. 传统方式定义的数组不能跨函数使用

A函数定义的数组，只有在A函数运行期间才可以被其他函数使用，

但A函数运行完毕后，A函数中的数组将无法在被其他函数使用。

```
#include<stdio.h>
void g(int * pArr, int len)
{
    pArr[2] = 88; //parr[2]==a[2] 等价于
}
void f(void)
{
    int a[5] = {1,2,3,4,5}; //数组a 只在f()执行时有效
    g(a,5);
    printf("%d\n", a[2]);
}
int main(void)
{
    f(); // 结果： 88
    //printf("a[0] = %d\n", a[0]); // error
    return 0;
}
```

为什么需要动态分配内存

很好的解决的了传统数组的4个缺陷

动态内存分配举例_动态数组的构造 难点

```
/*2011-05-01
malloc是memory(内存) allocate(分配)的缩写 动态内存空间是怎么造出来的?
*/#include <stdio.h>
#include <malloc.h>

int main(void)
{
    int i = 5; //分配了4个字节，静态分配
    int * p = (int *)malloc(100);
    /*
        1. 要使用malloc函数，必须要添加malloc.h头文件
        2. malloc函数只有一个形参，并且形参是整型
        3. 100表示请求系统为本程序分配100个字节
        4. malloc函数只能返回第一个字节的地址，但此时并不能确定该变量的类型，只有将这个地址被强制类型转化成存放整型变量的地址，这时才传达出指向整型变量的信息。
        5. 系统总共分配了104个字节的内存空间,p变量本身占4个字节(静态分配)，p所指向的内存占100个字节（动态分配） 若为int 则可存25个，若为char则可存100个变量。
        6. p本身所占的内存是静态分配的，p所指向的内存是动态分配的
    */
    free(p);
    //free(p)表示把p说指向的内存空间给释放掉，
    //p本身的内存不能释放，只有main函数终止时，由系统自动释放

    *p = 5;
    //*p代表的就是一个这int变量，*p这个整型变量的内存分配方式和int i =5;不同。
    //*p是内存是动态分配的， int i是静态的。

    printf("同志们好！\n");

    return 0;
}
```

```
/*
    2011-05-01
    目的： malloc使用_2
*/
```

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

void f(int * q) //q是p的拷贝或副本 q等价于p *q等价于*p *q=200则*p=200
{
    //*p = 200;     //error f()没有p变量，p是在main()函数定义的
    // q = 200;  //error q是指针变量（地址），200是整数int
    *q = 200;   //OK! 类型一致
    //  * *q  语法错误 ！*q整型变量，  只有指针变量前可以加*
    //free(q);//把q指向的内存释放掉
}


int main(void)
{
    int * p = (int *)malloc(sizeof(int)); //sizeof(int)=4;
    *p = 10;

    printf("%d\n", *p); //10
    f(p);
    printf("%d\n", *p); //200
              // f()函数中 free(q)作用后,则输出 -572662307（垃圾值）
    return 0;
}

/*
    2011-05-02
    目的：动态一维数组示例
    realloc(pArr, 100)
    //扩充动态内存空间 (原来50变100；原来150变100)
    //保留原来动态内存中未被截取的内容
*/

#include <stdio.h>
#include <malloc.h>

int main(void)
{
    //int a[5]; //系统静态地分配20个字节的空间给数组a

    int len;
    int *pArr;
```

```
    printf("请输入你要存放的元素个数：" );
    scanf ("%d", &len);  //5
    pArr = (int *)malloc(4*len);    //pArr指向这20个字节动态空间的前4个字节
    /*
    动态的构造了一个一维数组，该数组的长度len， 数组名是pArr, 数组元素类型是int
    类似与int pArr[len]; len可以根据需要变化
    */

    //对一维数组进行操作，如：对动态一维数组进行赋值
    for (int i=0; i<len; ++i)
        scanf("%d", &pArr[i]);

    printf("动态数组元素为: \n");

    //对一维数组进行输出
    for (i=0; i<len; ++i)
        printf("%d\n", pArr[i]);

    free(pArr);   //动态空间被释放

    printf("%d\n", *(pArr+1));
    //动态空间被释放，原来动态数组数元素内容为垃圾值-572662307

    return 0;
}

/*------------在VC++6.0输出结果：
请输入你要存放的元素个数：4
4 6 8 10
动态数组元素为:
4
6
8
10
*/
```

使用动态数组的优点:

1. 动态数组长度不需要事先给定；
2. 内存空间可以手动释放；
3. 在程序运行中，动态内存空间大小可以通过realloc函数手动扩充或缩小

静态内存和动态内存的比较

静态内存是由系统自动分配，有系统自动释放

静态内存是在栈分配的

动态内存是由程序员手动分配、手动释放

动态内存是在堆分配的

```
/*
   2011-05-02
   目的：  多级指针  --  自己画几个示意图  就会豁然开朗。
/

#include <stdio.h>

int main(void)
{
   int i = 10;       // i
   int * p = &i;     // 最终 *p就是 i;
   int* *q = &p;  // q 只能存放 int *类型的地址 即p 的地址&p

   int** *r = &q;  // r 只能存放  int **类型的地址 即q 的地址&q

   //r = &p; //error! 因为r是int***类型，只能存放int **类型变量的
地址
   printf("i = %d\n", ***r); *r = q; **r = *q =p  ***r = **q = *p =
i;
   printf("i = %d\n", **q); *q = p ;  **q = *p  === i
   printf("i = %d\n", *p); *p = i;
   printf("i = %d\n", i);
   return 0;
}

/*-----------在VC++6.0输出结果：
i = 10
i = 10
i = 10
i = 10
*/

#include <stdio.h>

//多级指针在函数中的应用
void f(int ** q)
{
   **q = 100; //*q就是p
```

```
}

void g()
{
  int i = 10;
  int * p = &i;
  printf("i = %d  *p = %d\n", i, *p);

  f(&p); //p是int *类型    &p就是int ** 类型

  printf("i = %d  *p = %d\n", i, *p);
}

  int main(void)
{
  g();
  return 0;
}

/*
  -----------在VC++6.0输出结果：
  i = 10   *p = 10
  i = 100  *p = 100
*/


 #include <stdio.h>
#include <malloc.h>
void f(int * q) //q是p的拷贝副本
{
  *q = 1;
}

void g(int **r)
{
  **r = 2;
}

void h(int ***s)
{
  ***s = 3;
}

void i(int ****t)
```

```
{
  ****t = 4;
}
```

//要想修改函数变量的值，只能发送该变量的地址,修改一个以上的值，必须用指针

```
int main(void)
{
  int *p = (int *)malloc(4);
  printf("*p = %d\n", *p);//垃圾值

  f(p); //调用的是指针
  printf("*p = %d\n", *p);//1

  g(&p);//调用的是指针变量的地址
  printf("*p = %d\n", *p);//2

  //h(&(&p));// error C2102: '&' requires l-value
  int **pp = &p; //pp是存放p地址的指针， int ** 整型指针的指针类型
  h(&pp); //调用的是存放p指针的指针的地址   int ***整型指针的指针的指针类型
  printf("*p = %d\n", *p);//3

  int ***ppp = &pp;
  i(&ppp);//调用的是一个三级指针的指针的地址，int **** 整型四级指针
  printf("*p = %d\n", *p);//4
  return 0;
}
```

跨函数使用内存的问题 难点

```
/*
  2011-05-02
  目的：跨函数使用内存
        函数内的静态空间，不能被其他函数调用访问
*/
  #include <stdio.h>

   void f(int **q) //理解为 int* *q
   {
     int i =5;
```

```
    //*q等价于p       *p和**q都不等价于p
    //*q = i;     //error *q等价于p 推出 p=i; 错!
    *q = &i;      //**q = *p = i;
}
int main(void)
{
    int *p;

    f(&p);
    printf("%d\n", *p);
return 0;
}
/*结果：5
本语句语法没有问题，但逻辑上有问题
内存越界：程序访问了一个不该被访问的内存
函数内的静态空间，不能被其他函数调用访问
函数中的内存空间，随函数终止而被释放。
内存空间释放后的内容不属于其他函数，其他函数无权限访问。
但释放后的内存空间的地址是可以被其他函数读取的。
但指针变量可以存贮任何函数中静态内存空间的地址，p都能存垃圾，p想存谁存谁。
只是它此时已经没有权限读取（访问） i这个地址的数据了，出错。
*/

/*
2011-05-02
目的：动态内存可以跨函数访问
程序运行在栈顶进行
```

静态空间是在栈里面分配的，函数终止本质叫做出栈，所以静态空间随着函数终止而释放，
动态空间是在堆里面分配的，与栈无关，与函数终止无关，不随着函数终止而释放。

堆和栈相关深入知识就需要《数据结构》和《操作系统》两门课学习，而这两门课难度大，理论性强，短期内收不到立竿见影的成效，属于内功心法，因此大多培训班已经取消了学习。

```
可以用free()释放
*/

#include <stdio.h>
#include <malloc.h>


void f(int ** q)//*q等价p 已经声明了q的类型为int **
```

```
{
  *q = (int *)malloc(sizeof(int));//sizeof(整数类型)
/*
不要用4，因为c语言只规定short int字节数 小于 int字节数 小于 long int字节数，没有规定明确的字节数，无统一硬性规定。
不同软件系统可能出现不同，统一用sizeof(int)来获取实际值
int * p;在p声明的情况下，
构造动态空间也可以写成 p = (int *)malloc(sizeof(int));
*/
  // *q等价p, 等价于 p = (int *)malloc(sizeof(int));
  //  q = 5;//error! q指针
  // *q = 5;//error! p = 5
    **q = 5;//OK! 等价于 *p = 5
}

int main(void)
{
  int * p;
  f(&p); //只有调用变量的地址，才能改变变量的值

  printf("%d\n", *p);
//f函数中，没有free(q);所以动态空间仍然保留，动态空间中的内容可以被访问

  return 0;
}
/*
----------在VC++6.0输出结果：
5
*/
```

枚举

什么是枚举

把一个事物所以可能的取值一一列举出来

```
/*
    日期：2011-05-04
    目的：枚举
*/

#include <stdio.h>

//自定义了一个数据类型，并没有定义变量，该数据类型的名字
enum WeekDay
```

```
enum WeekDay
{
   //MonDay, TuesDay, WednesDay, ThursDay, FriDay,
SaturdDay, Sunday
     MonDay=10, TuesDay, WednesDay, ThursDay, FriDay,
SaturdDay, Sunday

};//分号

int main(void)
{
   //int day;//day定义成int类型范围太大不合适，day的取
值只可能有7个(0-6)，浪费空间
   enum WeekDay day = FriDay; //初始化一个enum WeekDay 类
型变量 day
   printf("%d\n", day);


   return 0;
}

/*
-----------在VC++6.0输出结果：
4

14
*/
```

怎么使用枚举

```
/*
   日期：2011-05-04
   目的：枚举2
*/

#include <stdio.h>
enum weekday
{
   MonDay, TuesDay, WednesDay, ThursDay, FriDay,
SaturdDay, Sunday

};

void f(enum weekday i)//本函数的目的只是期望接受0-6之间
的数字，将形参定义为枚举
{
```

```
    switch (i)
    {
        case 0:
            printf("MonDay !\n");
            break;
        case 1:
            printf("TuesDay !\n");
            break;
        case 2:
            printf("WednesDay !\n");
            break;
        case 3:
            printf("ThrusDay !\n");
            break;
        case 4:
            printf("FriDay !\n");
            break;
        case 5:
            printf("ThursDay !\n");
            break;
        case 6:
            printf("SunDay !\n");
            break;
    }

}

int main(void)
{
    f(FriDay);//虽然FriDay本质上就是5，但直接写出f(5);就是错的，也不可能写成Friday  大小写敏感
    return 0;
}
```

枚举的优缺点

优点：代码更安全（强制输入），比较直观（有意义）

缺点：书写麻烦，不能出错。

总结：当是有限个元素时，用枚举更安全，高效。

位运算符

约翰·冯·诺依曼（JohnVonNouma，1903－1957），美藉匈牙利人

被称为计算机之父：2大贡献

二进制

计算机设备分类：运算器 控制器 存储器 输入设备 输出设备

什么是进制

数字是本质，进制只是不同表现方式
一个十六进制位，要用4个二进制数表示，$(1)_{16}$ = （0001）$_2$ 前面补齐

二进制 逢 二进一
十进制　　逢 十进一 dec
八进制　　逢 八进一　　oct 0数字　int i = 05;
十六进制　逢 十六进一 hex　0x数字　0X数字
　　int i = 0x5;int i = 0X5;

生活中：
七进制　　七天进周
十二进制　十二月进年
二十四进制 二十四小时进日
六十进制　六十分钟进小时
　　六十秒钟进分钟

汇编里
1101B 二进制
1357O 八进制
2049D 十进制
3FB9H 十六进制

十进制(D)	二进制(B)	八进制(O)	十六进制(H)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a (A)
11	1011	13	b (B)
12	1100	14	c (C)
13	1101	15	d (D)
14	1110	16	e (E)
15	1111	17	f (F)
16	10000	20	10

017 = 7 + 1*8 = 15
0x17 = 7 + 1*16 = 25
1234 = 4 + 1*10三次方 + 2*10二次方 + 3*10一次方
0x32c = c + 3*16二次方 + 2*16一次方 = 4354
最高幂数是位数-1

```c
#include<stdio.h>
int main(void)
{
    int i = 0x32c;
    printf("i = %d\n", i);
    printf("i = %x\n", i);
    printf("i = %o\n", i);
    /*
        printf的用法
        %d 以十进制输出
        %x 或 %X  以十六进制输出
        %o 或 %0  以八进制输出
    */
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    int i = 1000;
    print("%X\n", i)  //3E8
    printf("%#X\n", i) //OX3E8 %#X 推荐
    return 0;
}
```

补码：

原码：

也叫符号绝对值

最高位0表示正 1表示负，其余二进制位是该数字的绝对值的二进制位

在计算机中，从未被使用！

反码

反码运行不便，也没有在计算机中应用

移码

表示数值平移n位，n称为移码量

移码主要用于浮点数的阶码的存储

补码

地址是内存单元编号从 0到4G-1

即 2的32次方-1 总线若是32位，则有32个0，1

主要解决整数的存储 int 4字节 32位个0，1

A 已知十进制求二进制

求正整数的二进制

除2取余，直到商为零，余数倒序排列

求负整数的二进制

先求出与该负数相对应的正整数的二进制代码，

然后，将所有位取反末尾加1，不够位数时，左边补一

4字节int -5 先求5的二进制

0000 0000 0000 0000 0000 0000 0000 0101 所有位取反，末尾加1

1111 1111 1111 1111 1111 1111 1111 1011 16进制：FFFFFFFB

2字节 short int(-3) 先求3的二进制

0000 0000 0000 0011 所有位取反，末尾加1

1111 1111 1111 1101 用十六进制表示： FFFD

求零的二进制

全是零

B 已知二进制求十进制

如果首位是0，则表明是正整数，

按普通方法来求

如果首位是1，则表明是负整数，

将所有位取反末尾加1，所得数字就是该负数的绝对值

习题：FFFFFFF5 已知二进制 求其代表的整数是多少？

1111 1111 1111 1111 1111 1111 1111 0101

由于最高位是1，所以最终是负数，先对其所有取反

0000 0000 0000 0000 0000 0000 0000 1010 末尾加1后

0000 0000 0000 0000 0000 0000 0000 1011 该值为11 所以最终结果：-11

如果全是零，则对应的十进制数字就是零

C 二进制 到 十六进制

4位一段 从右到左 分别转化 不够左边初零

$(0010\ 1110)_2$ — $(2E)_{16}$

D 十六进制 到 二进制

一位转化成4位，不够左边补0

$(1)_{16}$-- $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2$

E 二进制 转到 八进制

从右往左，三位一段 不够被0

F 八进制 与 十六进制 不能直接转化。通过中间进制。

进制总结：

进制属于什么知识点，许多人爱不懂装懂。学习好它有利于将来学习底层如汇编等知识，但要往高级语言发展则不需要掌握，另外学好它对后面位运算等有帮助。

学习后应掌握：

在在VC++6.0中一个int类型变量所能存储的数字的范围是多少

32位系统，32个0，1组合表示的内存单元，8个十六进制数组合）

int类型变量所能存储的最大正数用十六进制表示： 7FFFFFFF

int类型变量所能存储的绝对值最大负整数用十六进制表示：

80000000

最小负数的二进制代码是多少 1（0-0 31个0）

最大正数的二进制代码是多少 0（1-1 31个1）

已知一个整数的二进制代码求原始的数字

按“已知二进制求十进制”求

数字超过最大正数会怎样 变成负数 0111 1111=127 1000 0000 -128

不同数据类型转化

丢失最高位.只截留后者类型所占的字节数。

例如： int 类型 转化 为char 则高3位字节被截取。只保留最后一位字节。

结构体 （非重点）

为什么需要结构体

为了表示一些复杂的事物，而普通的基本类型无法满足实际要求。

什么叫结构体

把一些基本类型数据组合在一起形成的一个新的复合数据类型。

如何定义结构体

3种方式：

```
// 第一种  只是定义了一个新的数据类型，并没有定义变量   推荐采用1
struct Student
{   Int age;
    Float score;
    Char sex;
};
//第二种
   struct Student
{   Int age;
    Float score;
    Char sex;
} st;
// 第三种
struct
{   Int age;
    Float score;
```

```
    Char sex;
} st;
```
怎样使用结构体变量

赋值和初始化

定义的同时可以整体赋值。

如果定义完之后，则只能单个赋初值。

如果取出结构体变量中的每一个成员{重点}

1 结构体变量名.成员名

2 指针变量名->成员名 （更常用）

它会在计算机内部转化成 (* 指针变量名).成员名 的方式来执行。

所以两者是等价的。

例子：

```
 Struct Student
{
    Int age;
    Float score;
    Char sex;
};
    Int main(void)
{
    Struct Student st = {80,66.6f,'F'};
//初始化定义的时候赋值  66.6在C语言中默认是double类型，如果希望一个实数是float类型，则必须在末尾加f或F, 因此66.6是double, 66.6f或66.6F是float
      struct Student st2;
    st2.age = 10;
    st2.score = 88;
    st2.sex = 'F';   // 第二种方式 单独赋值


    Struct Student *pst = &st; // &st 不能写成 st
    Pst->age = 88; // 通过结构体指针变量
    printf("%d %f\n", st.age, pst->score);
    Return 0;
}
```
理解：1 pst->age 会在计算机内部转化成 (*pst).age 的方式来执行,没有为什么，这就是->的含义，这也是一种硬性规定。

2 所以 pst->age 等价于 (*pst).age 也等价于 st.age

3 pst->的含义：pst 所指向的那个结构体变量中的age这个成员。

结构体变量的大小略大于其内部成员类型所占字节数之和。试：sizeof(struct)

若想通过函数对主函数结构体变量进行修改，则主函数必须发送地址，外函数定义指针结构体变量，通过外函数内部语句完成对变量的修改。

而仅想输出、读取操作，则不用传地址，定义指针过程。

例：

```
/*
    2009年11月24日9:17:43
    通过函数完成对结构体变量的输入和输出
*/


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

struct Student
{
    int age;
    char sex;
    char name[100];
}; //分号不能省

void InputStudent(struct Student *);
void OutputStudent(struct Student ss);
int main(void)
{
    struct Student st;  //15行

    InputStudent(&st); //对结构体变量输入  必须发送st的地址
     //printf("%d %c %s\n", st.age, st.sex, st.name);  // 此行和 下行输出函数 功能相同
    OutputStudent(st); //对结构体变量输出  可以发送st的地址也可以直接发送st的内容

    return 0;
}

void OutputStudent(struct Student ss)
{
    printf("%d %c %s\n", ss.age, ss.sex, ss.name);
}

void InputStudent(struct Student * pstu) //pstu只占4个字节
{
    (*pstu).age = 10;
  // pstu->name = "张三丰"; 或(*pstu).name = "张三丰";  都是error，提示错误信息： cannot convert from 'char [5]' to 'char [100]'
    strcpy(pstu->name,"张三丰"); // 用字符串拷贝命令解决问题
    pstu->sex = 'F';
```

```
}

/*
//本函数无法修改主函数15行st的值 所以本函数是错误的
void InputStudent(struct Student stu)
{
    stu.age = 10;
    strcpy(stu.name, "张三");  //不能写成 stu.name = "张三";
    stu.sex = 'F';
}
*/
```

结构体：应该发送地址还是内容

设计函数的目的：必须考虑 功能单一， 还要考虑安全因素

C++中指针前可加const 则只能读而不能修改其指向的变量。

指针的优点：

耗用内在小（4字节）

快速传递数据

执行速度快。

因此：推荐使用结构体变量作为函数参数来传递

```
/*
    2009年11月24日9:17:43
    示例:
        发送地址还是发送内容
    目的:
        指针的优点之一:
            快速的传递数据，
            耗用内存小
            执行速度快
*/


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

struct Student
{
    int age;
    char sex;
    char name[100];
}; //分号不能省

void InputStudent(struct Student *);
```

```
void OutputStudent(struct Student *);
int main(void)
{
    struct Student st ;  //15行
    //printf("%d\n", sizeof(st));

    InputStudent(&st); //对结构体变量输入  必须发送st的地址
    OutputStudent(&st); //对结构体变量输出  可以发送st的地址也可以直接发送st的内容 但为了减少内存的耗费，也为了提高执行速度，推荐发送地址

    return 0;
}

void OutputStudent(struct Student *pst)
{
    printf("%d %c %s\n", pst->age, pst->sex, pst->name);
}

void InputStudent(struct Student * pstu) //pstu只占4个字节
{
    (*pstu).age = 10;
    strcpy(pstu->name, "张三");
    pstu->sex = 'F';
}
```

结构体变量的运算

不能加减乘除操作，只能相互赋值。

例如：
```
struct Student
{
    Int age;
    Char sex;
    Char[100];
};
Struct Student str1, str2;
Str1 = str2 /  str2 = str1 ;
```
都是正确的。

举例：动态构造存放学生信息的结构体数组

```
# include <stdio.h>
# include <malloc.h>   // 必须先添加头文件 malloc

struct Student
{
    int age;
    float score;
```

```
    char name[100];
};

int main(void)
{
    int len;
    struct Student * pArr;
    int i, j;
    struct Student t;


    //动态的构造一维数组
    printf("请输入学生的个数:\n");
    printf("len = ");
    scanf("%d", &len);
    pArr = (struct Student *)malloc(len * sizeof(struct Student));
    //printf("%d\n",sizeof(struct Student));   // 108字节

    //输入
    for (i=0; i<len; ++i)
    {
        printf("请输入第%d个学生的信息:\n", i+1);
        printf("age = ");
        scanf("%d", &pArr[i].age);

        printf("name = ");
        scanf("%s", pArr[i].name);  //name是数组名，本身就已经是数组首元素的地址, 所以pArr[i].name 不能改成 &pArr[i].name

        printf("score = ");
        scanf("%f", &pArr[i].score);
    }

    //按学生成绩升序排序 冒泡算法
    for (i=0; i<len-1; ++i)
    {
        for (j=0; j<len-1-i; ++j)
        {
            if (pArr[j].score > pArr[j+1].score)  // >升序 <降序
            {
                t = pArr[j];    // 注意 t 的类型为  Struct Student
                pArr[j] = pArr[j+1];
                pArr[j+1] = t;
            }
```

```
        }
    }

    printf("\n\n学生的信息是:\n");
    //输出
    for (i=0; i<len; ++i)
    {
        printf("第%d个学生的信息是:\n", i+1);
        printf("age = %d\n", pArr[i].age);
        printf("name = %s\n", pArr[i].name);
        printf("score = %f\n", pArr[i].score);

        printf("\n");
    }

    return 0;
}
```

总结：
对于一个人事管理或图书管理项目，分析流程：
第一步；存储
第二步：操作
第三步：输出
前两个过程最难、最核心 是“数据结构”研究的重点，一般都屏蔽了。
数组 和 变量 虽然都可以存储，但都不完美。
比如：人事关系图、交通图等，都不好用数组保存。
从事关系结构只能用“树”还保存，而对于 两个起终点：公交线路查询，实现时间最小/ 距离最短/ 花费最低 等功能，只能用“图”来存。
而图 和 树 都必须有指针知识，它们属于较高深的思想层次的东西。因此要学好数据结构必须要 懂得指针。
若感兴趣：可以继续学习郝斌老师的《数据结构》教学视频。

链表（较难）

C语言和数据结构的连接(过渡)

链表是数据结构第一部分 而是C语言最后一章内容，由此可以比较两者难度

算法：

通俗定义：

解题的方法和步骤

狭义定义：

对存储数据的操作

对不同的存储结构，要完成某一个功能所执行的操作是不一样

比如：

要输出数组中所有的元素的操作 和

要输出链表中所有的元素的操作 是不一样的

这说明：

算法是依附于存储结构的

不同的存储结构，所执行的算法是不一样的

广义定义：

广义的算法也叫**泛型** C++

无论数据是如何存储的，对该数据的操作都是一样的

分层思想，站在更高的层次看，把内部的实现给屏蔽

数组和链表都是线性的，都是先输出一个元素后，再输出下一个元素

我们至少可以通过两种结构来存储数据

数组

优点：存取速度快

缺点：需要一整块连续的空间

（对于庞大数据，往往没有一个适合的较大的连续的空间如a[30000000000000]）

插入和删除元素效率很低

(插入和删除中间某个元素，其后的所有都要前后移动)

链表

优点：插入删除元素效率高

缺点：查找某个位置的元素效率低

（由于不是连续的，不同由下标直接找，必须由头至尾逐一比对查找）

两者各有所长，至今没有出现一个更优的存储方式，可集数组、链表优点于一身。

链表专业术语：

首结点：

存放**第一个有效数据**的结点

尾结点：

存放**最后一个有效数据**的结点，指针域的指针为NULL，尾结点的标志

头结点：

头结点的数据类型和首结点的类型是一模一样的

头结点是首结点前面的那个节点

头结点并**不存在有效数据**

设置头结点的目的是为了方便对链表的操作

头指针：

存放头结点地址的指针变量

确定一个链表需要一个参数，头指针

对于每个链表元素，分为左右两部分，左边为数据单元，右边为下一元素地址。
例：

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

struct Node
{
    int data; //数据域
    struct Node * pNext; //指针域
};

//函数声明
struct Node * create_list(void);
void traverse_list(struct Node *);

int main(void)
{
    struct Node * pHead = NULL;

    pHead = create_list();
//create_list()：创建一个非循环单链表，并将该链表的头结点的地址付给
pHead
    traverse_list(pHead);

    return 0;
}

struct Node * create_list(void)
{
    int len;  //用来存放有效节点的个数
    int i;
    int val; //用来临时存放用户输入的结点的值

    //分配了一个不存放有效数据的头结点
    struct Node * pHead = (struct Node *)malloc(sizeof(struct Node));
    if (NULL == pHead)
    {
        printf("分配失败，程序终止!\n");
        exit(-1);
    }
    struct Node * pTail = pHead;
    pTail->pNext = NULL;
```

```
    printf("请输入您需要生成的链表节点的个数: len = ");
    scanf("%d", &len);

    for (i=0; i<len; ++i)
    {
        printf("请输入第%d个节点的值: ", i+1);
        scanf("%d", &val);

        struct Node * pNew = (struct Node *)malloc(sizeof(struct Node));
        if (NULL == pNew)
        {
            printf("分配失败, 程序终止!\n");
            exit(-1);  //终止程序
        }
        pNew->data = val;
        pTail->pNext = pNew;
        pNew->pNext = NULL;
        pTail = pNew;
    }

    return pHead;
}

void traverse_list(struct Node * pHead)
{
    struct Node * p = pHead->pNext;

    while (NULL != p)
    {
        printf("%d  ", p->data);
        p = p->pNext;
    }
    printf("\n");

    return;
}
```

对于以上例题：不要求逐行敲出，但要能看懂。

字符串的处理

两种： 字符数组　字符指针

位运算

& 按位与　--每一位都按位与 （区别&j取地址）

1&1 = 1
1&0 = 0
0&0 = 0
0&1 = 0

| 按位或 -- 每一位都按位与

~ 取反 -- 每一位取反

^ 按位异或 -- 相同为零 不同为1
1^0=1
0^1=1
1^1=0
0^0=0

<< 按位左移 -- 左移n位相当于乘以2的n次方
i<<3 表示把i的所有二进制位左移动3位，右边补零
面试题：
A) i = i*8;
B) i = i<<3;
请问上述两个语句，哪个语句执行的速度快
答案：B快
乘法在运算器里，运行原理比较复杂
按位左移，简单！

>> 按位右移 -- 右移n位相当于除以2的n次方，首位为0补0，首位是1补1

i>>3 表示把i的所有二进制位右移动3位，左边补零
防止过度右移，容易丧失精度和意义

位运算的现实意义：
通过位运算符，我们可以对数据的操作精确到每一位。

NULL 二进制全部为零的含义：
---0000000000 的含义
1.数值零
2.字符串结束标记 '\0'
3.空指针NULL
NULL表示零，而这个零不代表数字零，而表示的内存单元的编号零

我们计算机规定了，以零为编号的存储单元的内容不可读，不可写
free(p);
p = NULL;
*p = 0; 错！把0号单元改写！0单元是非常重要的数据。程序员不可能

读写出0号单元信息

纯C的知识(略)

文件

不是用'流'的思想，用函数实现,于java C++没联系

宏

typedef

期末考试

1.什么叫分配内存，什么叫释放内存

分配内存：操作系统把某一块内存空间的使用权力分配给该程序

内存释放：操作系统把分配给该程序的内存空间的使用权力收回，
该程序就不能使用这块内存空间

附注：释放内存不是把该内存的数据清零

2.变量为什么必须初始化

不初始化，变量通常是垃圾值，很可能是上次程序结束遗留下来的数据。

3.详细说明系统如何执行：int i = 5; 这条语句的

1> Vc++6.0软件请求操作系统为i分配存储空间
2> 操作系统会在内存中寻找一块空闲的区域，把该区域当作i来使用
3> Vc++6.0会把i和这块空间区域关联起来，今后对字母i操作就是对这块空闲的区域操作。
4> 把5存储到字母i所关联的内存区域中

附注：所谓内存区域也就是内存的一块存储单元

4.详细列出C语言所有基本类型

int long int short int char float double

5.在printf函数中，int用%d输出，
请问：long int char double float分别用什么输出？

%ld %c %lf %f

6.函数的优点

1>避免重复操作
2>有利于程序的模块化

7.谈谈你对函数的理解

... ...

8.什么是指针，什么是地址，什么是指针变量，三者之间的关系？

地址是内存单元的编号 指针就是地址 指针和地址是同一个概念
指针变量是存放内存单元编号的变量

指针变量和指针是两个完全不同的概念，只不过人们通常把指针变量称作指针

9.请写出静态变量和动态变量的异同

相同点:
　　都需要分配内存
不同点：
　　静态变量是由系统自动分配，自动释放，程序员无法在程序运行的过程当中手动分配，
　　也无法在程序运行的过程中手动释放。
　　静态变量是在栈中分配的，
　　只有在函数终止之后，静态变量的存储空间会被系统自动释放。

　　动态内存是由程序员手动分配，程序员可以在程序运行的过程当中手动分配，手动释放。
　　动态变量是在堆中分配的，
　　程序员可以在行是执行过程中的任何时刻手动释放动态变量的空间
　　不需要等到函数终止才释放。

10.C语言中哪些知识点是我们学习的重点，请一一列举出来

流程控制 函数 指针 静态内存和动态内存

11.for(1;2;3)

```
    A;
    B;
```

1> 2成立，会继续执行哪条语句：A
2> 3执行完毕后，会继续执行哪条语句: 2
3> A执行完毕后，会继续执行哪个语句:3
4> 1总共会执行几次：1次

12.for(1;2;3)

```
    for(4;5;6)
    {
        A;
        B;
    }
```

```
    C;
```
1> 6执行完毕后，会继续执行哪个语句：5
2> 5成立，会继续执行哪个语句：A
3> 5不成立，会继续执行哪个语句：3
4> 2不成立，会继续执行哪个语句：C
5> 2成立，会继续执行哪个语句：4
6> A 和 B语句是否一定会被执行 不会 (2或5不成)
7> C语句是否一定会执行 是

13.
```
for(1;2;3)
    {
        while(4)
            5;
            6;
            break;
        if(7)
            8;
            9;
    }
    10;
```
1> 5执行完毕后，会继续执行哪个语句：4
2> break终止什么？ 终止for， break终止最里层包裹它的循环
3> 如果8是break语句，则8执行完毕之后会继续执行哪个语句 10
4> 如果7不成立，会继续执行哪条语句 9

方法：调整清楚
```
    for(1;2;3)
    {
        while(4)
        {
            5;
        }
            6;
            break;
        if(7)
        {
            8;
        }
            9;
    }
    10;
```

14 判断下列程序语法上是否有错误，说出错误原因
A) int *p; *p = 10;　　错 p没有指向，*p数据不可读和操作

B) char *p; char ch=A; p=&ch; 错 A改成'A' A非法无意义
C) int i,j; i=j=0; int *p; p=&i;对 i=j=0; 从右向左
D) int *p; int **q; q = &p; 对 指针的指针是int ** 类型
E) int *p; int i=5; p=&i; *p=10;对 p指向i，*p=5;把10赋值给*p

15 编程实现：如果x大于0，则y为1.如果x小于0，则y为-1，如果x等于0，则y为0，
以下程序段中，不能根据x值正确计算出y值的是 CD

A)
```
if(x>0) y=1;
else if(x==0) y=0;
else y=-1;
```
B)
```
y=0;
if(x>0) y=1;
else if(x<0) y=-1;
```
C)
```
y=0;
if(x>=0);(;空语句)
if(x>0) y=1;
```
D)
```
    if(x>=0)
if(x>0) y=1;
else y=0;
```

16.若变量已正确定义，有以下程序段
```
int a=3, b=5, c=7;
if (a>b)
    a=b;
c=a;
if(c!=a)
c=b;
printf("%d, %d, %d\n", a, b, c);
```
输出结果是：B
A）程序段有语法错误 B）3，5，3
C）3，5，5 D）3,5,7

17.执行以下程序后，输出'#'的个数是:6
```
#include <stdio.h>
int main(void)
{
    int i,j;
    for(i=1; i<5; i++)
        for(j=2; j<=i; j++)
            printf("%c\n",'#');
    return 0;
}
```

18.有以下程序

```
#include <stdio.h>
int main(void)
{
    int i,s=0;
    for(i=1; i<10; i+=2)
        s+=i+1;  //s = s + (i + 1);
    printf("%d\n", s);
    return 0;
}
```

程序执行后的结果是： D

A） 自然数1～9的累加和

B） 自然数1～10的累加和

C） 自然数1～9中的奇数之和

D） 自然数1～10中偶数之和

19.若有一些定义和语句

```
int a = 4;
int b = 3;
int *p;
int *q;
int *w;
p=&a;
q=&b;
w = q;
q = NULL;
```

则以下选项中错误的语句是 A

A)*q=0;　B)w=q;　C)*p=88;　D)*p=*w;

20 以下程序

```
#include <stdio.h>
void fun(char *a, char *b)
{
    a=b; //指针ab相互指向，则他们的值相同
    (*a)++;
}
int main(void)
{
    char c1='A', c2='a', *p1, *p2;
    p1=&c1;
    p2=&c2;
    fun(p1,p2);
    printf("%c%c\n", c1, c2);
}
```

程序运行后的输出结果是：Ab
实验程序

```
/*
#include <stdio.h>
void fun(char *a, char *b)
{
    printf("%c %c\n", *a, *b);  // A a
    a=b;
    printf("%c %c\n", *a, *b);  // a a
    (*a)++;
    printf("%c %c\n", *a, *b);  // b b
}
int main(void)
{
    char c1='A', c2='a', *p1, *p2;
    p1=&c1;
    p2=&c2;
    fun(p1,p2);
    printf("%c %c\n", c1, c2);  // A b
}
*/


#include <stdio.h>
int main(void)
{
    int i = 0;
    int j = 1;
    int *a = &i;
    int *b = &j;
    a=b;   //OK 互为指向 *a=*b=1;
    printf("%d %d\n", *a, *b);  //1 1

    */
    *a+=1;
    printf("%d %d\n", *a, *b);
    //2 2  指针a和指针互为指向，改变*a的值，就是改变*b的值

    /*
    a=a+2; //把指针a的址编号加2，b不指向a，
        //b仍然指向a原来的地址无法知道与a地址相隔2个单元的内容*a.
    printf("%d %d\n", *a, *b);  //1245088 2
    return 0;
}
```

21. 若有定义：int k;
    以下程序段的输出结果是：##2##4

```
for(int k=2; k<6; K++, K++) printf("##%d", k);
```

K++, K++ = 1个循环因子

22. break
23. 进制转化
24. C的一些认识问题。

此致：郝斌老师的C语言教程全部结束。课堂讲解全程动手敲代码，讲解细致，对于重要知识点的讲解不厌其烦，是一个难得的C语言入门教程。在这里对老师的辛勤付出表示感谢。

2014-3-8