C 语言入门

Nisy 著

导读

有不少人问，学 C 语言需要什么基础？我想说，你有兴趣吗？你有时间吗？如果两个都准备好了，就可以开始了。数学不太好能学好 C 语言吗？关系不大，就跟婴儿学说话前是否需要具备数学知识一个道理。

本书是写给 C 语言初学者的。文中对 C 语言的语法部分没有过多阐述，因为其他 C 语言书上已经对 C 标准讲的很详细了。而是侧重对 C 语言中一些司空见惯的内容重新进行了剖析，如对 C 语言的思考、程序是什么的思考、教学顺序的思考、对变量是什么的思考、对模块化程序设计的思考、对递归函数的思考、对指针的思考等。虽没有太多的语法内容，但本书还是拥有一个完整的 C 语言框架的，对 C 基础知识有一些了解的朋友可能读起来会更轻松一些。

写这个东西的目的很单纯，就是把自己对 C 的理解以及 C 教学方法的一些心得和大家做一个交流。C 语言的书籍很多，大都将“Hello,World!”作为见面礼，本书中的第一节内容是先让学习者建立一个内存环境模型，因为我觉得一个 C 程序员脑海中若没有对内存环境建模是一件很荒唐的事情，C 的精髓在于指针，连空间模型都模糊，谈指针又何必。由于时间等原因，书中对一些知识点没有详细讲述，如 switch…case…和一维数组的应用等，几句话很难讲透彻，但又不影响初学者对 C 语言的理解，故本书没有详细阐述。

在本书的阅读上，建议大家还是顺序来读，从第一章开始读，就如同看一幅画，只看局部是没有什么意思的。写这个东西就是一个纯交流，若大家在阅读时发现文中的错误，还望不吝赐教。关于致谢部分，能完成这个文档，我需要感谢的人很多，由于担心文章质量可能会影响到他们的声誉，故这里不再一一写明，只在心中默默感谢。

目录

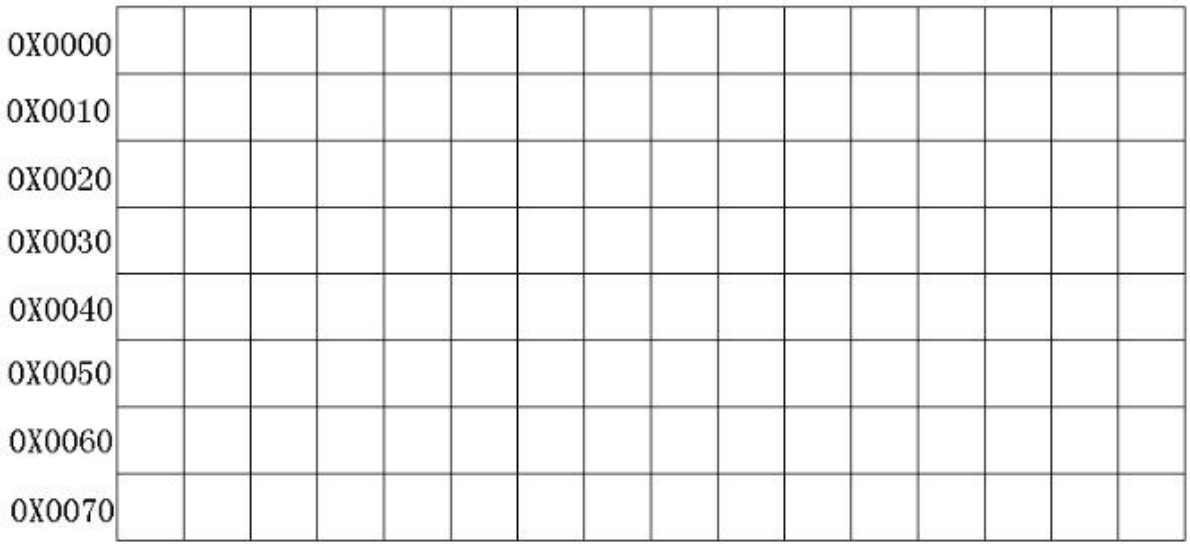
1 变量、判断、循环

1.1 变量的本质

1.1.1计算两个整数的和

C 语言的运行平台是 16 位的 DOS 平台，学语言前，我们最好对程序的运行环境做一些了解，这里先用一个图片来介绍一下内存情况：

内存模型图



程序运行要加载到内存中。我们可以把内存想象成一个方格纸，内存中存储数据的最小单位是字节，我们就把每个字节看做方格纸中的一个方格，内存中，每个方格都有一个对应的地址（我们可以试想一下，如果这些方格没有相对应的地址，程序将如何获取内存中所需要的数据）。

了解内存的模型后，我们先来设计这样一个程序，输入两个整数并计算出其和。输入两个整数，那我们输入的这两个整数应该放在哪？肯定最终得放到内存中，那又放在内存的什么地方，我们如何才能找到我们存放的数据？首先我们需要在内存中申请一块空间，还要找到该空间的这个地址。我们再思考一个问题，在内存中，这些方格如果光有地址行吗？可以找到完整的数据吗？不行，因为我们无法知道该地址对应的数据有多长，是一个字节呢，还是两个字节，或者是更多的字节。

所以我们的逻辑第一步就是要申请空间来存放我们输入的整数，我们需要为申请到的空间起一个名字，我们通过这个名字可以获取该空间的地址。因为我们一会还需要用到该空间存放的数据，如果我们找不到该空间的地址，我们就没有办法找到我们需要的数据了。

下面我们来看 C 语言语法中申请空间的格式，C 语言中有以下语法支持我们申请空间，他们是：

char　空间名 （可以申请 1 个字节的内存 可存放一个字符）
int　空间名 （可以申请 2 个字节的内存 可存放一个整数）
long　空间名 （可以申请 4 个字节的内存 可存放一个整数）
double 空间名 （可以申请 4 个字节的内存 可存放一个浮点数）

我们称 char、int、long、double 为数据的类型，申请空间的格式为：类型 该空间的名字。我们先申请两个一个字节的空间：

```
char a;  /* C 语言规定以‘;’来代表每条指令的结束 */
char b;
```

也可以写成这样 char a,b; 因为我们开辟的是相同大小的两个空间。我们定义的空间名自身指代的是内存中的数据，取该内存对应的地址的指令为：&空间名，即在变量名前加“&”取地址符号。

下面我们介绍 C 语言中接受数据的指令：scanf("数据存储类型",内存地址);我们要保存的数据类型是整形，该数据类型对应的格式是“%d”，我们申请了两个空间，获取所申请空间地址的语法是：&空间名。所以我们直接套用该语法即可：

scanf("%d %d",&a,&b); 说明：接收多个数据，数据存储类型间可用空格隔开（只要和输入格式相一致即可）。地址间用‘,’隔开。然后就是计算并显示两个数的和，C 语言中显示函数格式为：printf("显示格式",内存数据);

C 语言中计算两个数据加减乘除的语法和数学中相同，分别是：+、-、*、/ 。最后介绍 C 语言程序的语法：

```
main()
{
    我们程序的代码
}
```

有了以上的知识，该程序的代码就有了：

```
main()
{
  char a,b;
  scanf("%d %d",&a,&b);
  printf("%d",a+b);
}
```

注意：空间名等于该空间存所存储的数据。该指令中 printf("",); 若显示格式非“%s”，逗号后一定跟的是内存数据。

我们 tcc 编译连接后运行程序：
输入：12 23
输出：35

需要说明的是，由于我们每个变量申请的内存空间为一个字节，故能存储的整数大小为-128~127 之间，所以我们输入的数值及两数值之和需控制在该范围以内，如果需要输入更大的整数，请将申请空间扩展为两个或四个字节。

1.1.2如何编译连接一个程序

如果您之前有写过 C 程序，可以使用自己熟悉的编译环境，或者参看附录。
注：该书中的代码均在 TC2 中编译通过，在 VC6 中编译请遵循 C99 标准。

1.1.3申请连续的空间

上文中我们提到的申请空间，或者叫做在内存中开辟一个空间，在其他 C 语言书籍中叫做定义变量，我们给空间起的名字就叫做变量，因为存储的数据不是固定值。比如我们输入一个整数，可能每个人输入的数值都不相同。

我们既然知道如何在内存中申请空间了，那是否我们可以申请一个连续的空间，这样在计算该题时是否更方便些。C 语言中申请连续空间的方法其实很简单，就是在空间名后面加上一个‘[n]’，我们开辟一个双字的空间用 int a; 即可，我们想要开辟十个 int 型的连续空间，只需 int a[10]; 就可以了，该连续空间的地址就等于这个连续空间的空间名，即 &a == a 。

下面我们申请一个连续的空间来解决两数和的问题：

```
main()
{
   char a[4];
   scanf("%d %d",(int *)a,(int *)a+1);
   printf("%d \n",*(int *)a + *((int *)a+1));
}
```

我们逐条来解释：

1. char a[4]; 申请 4 个 char 型的连续的空间

2. a 为所开辟连续空间的名称，只要开辟连续空间时，空间名才等于该空间的地址，我们需要让该地址指向的数据为两个字节，而我们在开辟空间时告诉编译器该地址对应的单元为一个字节， char 型的，而我们现在想使用两个字节的空间来存储数据，所以就需要再次告诉编译器该地址指向的数据为两个字节，C 语言中提供了该语法支持，我们成为强制转化，其格式为：(变量类型) 需转化对象，其中需转化对象可以为内存地址，也可以为内存数据。(*)表示被修饰对象将被转化为地址型，该地址对应的类型就是‘*’之前的类型。如(int *)a 就表示将数值 a 强制转化为地址型，该地址指向的内存数据为 int 型，即两个字节。“(int *)a+1”中数据 a 被强制转化为指向 2 个字节的地址，a+1 表示 a 地址之后第 1 个该类型(int)型数据的地址。

3. 在地址前加‘*’，表示该地址指向的内存数据，所以 *(int *)a 就表示我们输入的第一个整数，该空间占两个字节，可表示的范围-32768~32767 之间。

当然我们也可以申请空间时就申请为 int 型的空间，这样我们在使用中就无需再次向编译器转化数据类型了。

```
main()
{
   int a[2];
   scanf("%d %d",a,a+1);
   printf("%d \n",*a + *(a+1));
}
```

这里 a+n 就表示地址 a 指向的数据之后的第 n 个该类型数据的地址，*(a+n) 就表示地址 a 指向的数据之后的第 n 个该类型的数据。其中 *(a+0) 可以写成 a[0]，*(a+n) 可以写成 a[n]。则该程序还可以写为以下形式：

```
main()
{
   int a[2];
   scanf("%d %d",a,a+1);
   printf("%d \n",a[0] + a[1]);
}
```

上文中我们指出，**定义变量就是申请或开辟空间，我们开辟的空间如何使用全在于我们使用前对编译器所做的声明（或强制转化）。**

上文中我们静态申请空间的方法来实现了“输入两数并输入其和”的问题，下面我们用动态申请空间的方法来实现这一个程序。

动态申请空间需用到一个申请空间函数：malloc(空间长度); 该函数返回所申请到空间的地址，该地址类型为 void *型，所以使用该空间前我们还需对该地址进行强制转化。释放空间的函数名为：free(空间地址); 该函数可释放该地址指向的空间。

我们先申请两个字节的空间用来接收动态申请空间的地址，C 语言有六种编译模式，默认的为 small 模式，small 模式下指针（地址）为两个字节。紧凑模式以上的模

式为 4 个字节，前两个字节为偏移值，后两个字节为段地址，这一点大家先记住即可，我们在后文中将一起证明该结论。

```
main()
{
   int a;
   (void *)a=(void *)malloc(sizeof(int)*2);
   scanf("%d %d",(int *)a,(int *)a+1);
   printf("%d \n",*(int *)a+*((int *)a+1));
}
```

我们开辟空间的时候告诉编译器 a 为 int 型数据，默认存放的数值为整型，而这里我们想用该空间来存放一个地址（也叫指针），所以此时我们需要强制转化为指针型，void * 仅代表该数据为一个地址。sizeof(); 函数用来求取目标对象所占内存的字节数。

1.1.4对内存中字符串的思考

下面我们看一下 C 语言如何处理内存中的字符串，先看一个例子：

```
main()
{
   char a[]="ChinaPYG!";
   puts(a);
}
```

我们先开辟了一个空间，而空间大小没有直接给出，而是通过后面的字符串（每个字符占一个字节的空间）间接给出的。编译器肯定是通过字符串长度来开辟的这个连续空间。下一行指令 puts(a); 是输出地址 a 指向的字符串。这里就有些奇怪了，在处理一个数据时，只给了数据的地址，而没有确定长度，那该指令应该如何实现呢？

这个问题如果交给我们来处理，我们应该如何来设计呢？

打印一个字符串，肯定是一个字符一个字符的依次打印，而只给了地址没有给打印的长度或次数，该指令应该如何结束呢？除非是在该字符串之后加上一个结束的标志，该当指令读取到该标志时就会自行结束。或在事先可以获取到该字符串的长度。若没有打印次数，那就必须要有结束标志。

这个是我们推理出来的，C 语言是人设计的，设计的时候一定有符合人逻辑的东西在里边，我们用自己分析出的结论来和答案对照，发现确实字符串末尾有一个结束标志，该标志为‘\0’，用 16 进制表示就是 0X00 。

所以 char a[]="ChinaPYG!"; 实际申请的空间就为 9+1 。该问题我们可以用 sizeof(); 函数来验证。

```
main()
{
   char a[]="ChinaPYG!";
   printf("%d \n",sizeof(a));
   puts(a);
}

C:\tc>test
10
ChinaPYG!
```

下面我们再介绍一个函数，memcpy(); 可以将一段内存的数据复制到另一个段内存地址中。

```
main()
{
   double a[2];
   char * str="ChinaPYG!";
   memcpy(a,str,sizeof("ChinaPYG!"));
   puts(a);
}
```

memcpy(); 函数跟三个参数（参数我们在下文中会作详细解释，这里大家只需套用格式即可），第一个参数为目标地址，第二个参数为源地址，第三个参数为字节数长度。该函数可以将源地址指向的数据复制到目标地址，复制的字节数等于第三个参数。

下面我们将该程序修改一下：

```
main()
{
   double a[2];
   char * str="ChinaPYG!";
   memcpy(a,str,sizeof("ChinaPYG!"));
   printf("%f",(double *)a);
}
```

输出结果：

```
C:\tc>test
7.347250443619764660000000000000000000000e+223
```

如果我们将 a 地址对应的 8 个字节按照浮点数格式输入，那这个字符串就可以表示为对应的浮点数数值。可见，**内存中的数据，关键在于我们使用时对其的声明，即变量的类型就是告诉编译器如何存储或如何使用。**

1.1.5 i++中背后的思考

下面我们再来看一个问题：i++

i++; 就相当于 i=i+1;

```
main()
{
   int a=5,b;
   b=a++;
   printf("%d \n",b);
}
```

++ 在 a 后 表示 a 先 +1 后再使用。b=a++; 相当于 b=5，然后 a++，a=6 。
++ 在 a 前 表示 a 先 使用后再 +1。b=++a; 相当于 ++a，a=6 然后 b=6 。

下面我们再来看一个大家喜欢讨论的问题：

```
main()
{
   int a=5,b;
   b= ++a + ++a + ++a;
   printf("%d",b);
}
```

代码在 tc2 中运行结果为：24
该代码在 VC6 中结果为：22

这些结果好像我们都无法接受，一些 C 语言的书籍都对类似的程序做出过精辟的讲解，我们也简单的来分析下该程序究竟是如何进行的运算。

在 TC 中编译得：

```
push    si
push    di
mov     si, 5               ; a=5
```

```
inc     si              ; a=5+1=6
inc     si              ; a=6+1=7
inc     si              ; a=7+1=8
mov     di, si
add     di, si          ; 8+8=16
add     di, si          ; 16+8=24
push    di
mov     ax, 194h
push    ax              ; format
call    _printf
```

在 VC6 中编译得：

```
push    ebp
mov     ebp, esp
sub     esp, 48h
push    ebx
push    esi
push    edi
mov     [ebp+a], 5              ; a = 5
mov     eax, [ebp+a]
add     eax, 1                  ; a = 5+1=6
mov     [ebp+a], eax
mov     ecx, [ebp+a]
add     ecx, 1                  ; a = 6+1=7
mov     [ebp+a], ecx
mov     edx, [ebp+a]
add     edx, [ebp+a]            ; 7+7=14
mov     eax, [ebp+a]
add     eax, 1                  ; 7+1=8
mov     [ebp+a], eax
add     edx, [ebp+a]            ; 14+8=22
mov     [ebp+var_8], edx
mov     ecx, [ebp+var_8]
push    ecx
push    offset aD               ; "%d"
call    printf
```

通过反编译后的汇编指令，我们可以看清该程序是如何进行的运算，貌似终于弄明白了该运算的过程有不少的收获。那下边这个程序你能马上说出结果吗？

```
main()
{
   int a=5,b;
```

```
    b=a++ + a + ++a + a++ + ++a;
    printf("%d",b);
}
```

不能，为什么？因为我们不知道 a 究竟是如何进行运算的，不清楚其中的逻辑。那这里就又有一个问题，连我们这些写代码的人都弄不明白该指令的逻辑，那编译器能猜出我们想要表达的逻辑吗？自然不能。所以运行后的结果我们一定也会让我们迷茫。

那既然在一条指令中若对同一变量进行多次自增自减运算后的结果我们很难猜测，C 语言为何还支持该条语句的编译呢并不报错呢？首先是代码语法格式是 OK 的，所以没有报错。我们再深入想一下其中的奥妙：如果程序员的逻辑混乱，那编译器出的程序就会出错。反过来推理，如果逻辑正确，执行结果自然正确。可能编译器的作者故意留下这个悬疑让我们来参透其中的奥秘：如果你的逻辑正确，程序自然就正确，如果你的逻辑混乱，程序自然有问题。所以这里我给出一个公式：

源代码 = 人的逻辑 + 用语言描述一下

1.2 分支语句

计算机语言是人设计的一种语言，是为了帮助人类解决问题而存在的。代码就是人逻辑上的一个体现。所程序中含有人的逻辑，那么 C 语言中就一定要包含人的逻辑。

生活中我们会常有这样的情况，假如说吃午饭时一个朋友到我家做客，我会说：“如果你还没吃饭；或者你还没吃饱饭；或者你喜欢吃我做的饭，那么就过来一块吃。”。人在生活中处理问题时常会做出这样的假设，而程序也是要解决问题，所以计算机语言中也需要能够对问题的不同分支做出不同的处理，所以就需要有类似“如果”的语句来处理问题中的分支。

C 语言中，“如果”的指令语法为：

```
if( 判断条件 )
{
        执行语句
}
```

上方吃饭的问题用 C 语言描述就是：

```
if( 你还没吃饭 || 你还没吃饱饭 || 你喜欢吃我做的饭 )
{
        过来一块吃;
}
```

if 语句中判断条件的数值为逻辑真或者逻辑假，只有当 if 语句中条件逻辑真为 1 时，才执行{}内的语句。判断条件中，需要同时满足的条件叫做逻辑与，写做“ && ”；满足一个即可的叫做逻辑或，写做“||”。

下面我们来看这样一个问题，输入一个年份，来判断该年份是否为闰年。首先我们要考虑一下闰年的条件，年份要可以被 4 整除，同时不能被 100 整除；或者该年份可以被 400 整除。满足该条件的年份就是闰年。既然知道了判断闰年的条件，那么我们用 C 语言来描述一下：

```
main()
{
   int y;
   scanf("%d",&y);
   if((!(y%4) && y%100) || !(y%400))
   {
      printf("Yes!");
   }
   else printf("No!");
}
```

“!(判断条件)”表示对条件值取反，如 0 取反后变成 1。

1.3 循环指令

1.3.1大小写字母的转化问题

下面我们来思考一个问题，如何将“ChinaPYG!”中的小写字母变成大写字母。我们在处理这个问题的时候，肯定需要逐个的去处理，先判断该字母是否为小写，如果是则进行相应处理变成大写。这个是我们处理这个问题的逻辑。

生活中，太多的事情在处理时都要依次进行处理，比如我们喝水都需要一口一口喝，写一篇字都要一个一个写。所以语言中就需要有支持该逻辑的语法，否则一些问题将无法处理。

C 语言中用于循环的语法是：

```
for()
{
}
```

我们用中文来描述一下处理该字符串的问题就是：

```
main()
{
   char * str="ChinaPYG!";
   int i;
   for(i=0;i<;i++)
   {
      if(*(str+i)为小写字母) 那么将该字母变成大写字母
   }
   puts(str);
}
```

for() 语句的语法就是()中三句话：循环变量赋初值；判断条件（若满足则执行循环体，不满足则退出 for 语句执行 for 之后的语句）；循环变量自增减

变量赋初值也写到 for 语句之前，变量自增减可以写到执行体中，但中间的“;”是必须要有的，例如：

```
变量赋初值
for( ; 判断条件; )
{
        循环体语句
        变量自增减（若变量不能自增减 循环条件将一直成立 导致成为死循环）
}
```

该语句又可以精简为 while()语句：

```
变量赋初值
while(判断条件)
{
        循环体语句
        变量自增减
}
```

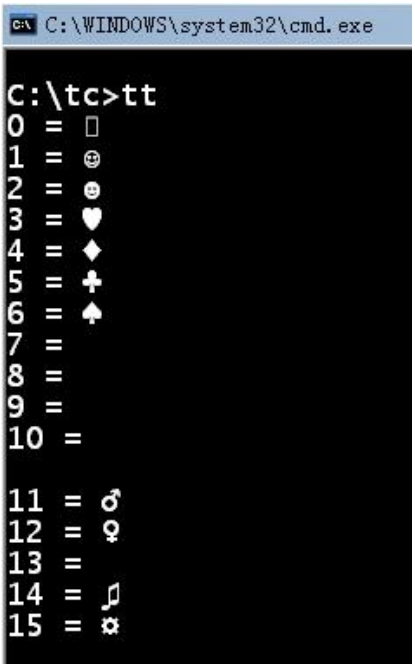
我们处理该问题需要先了解一下数据在内存中的存放模式，先来构建一个 ASCII 表。一个字节有 8 位，可表示的数字的大小为 0~255，正好是一个 16*16 的表格，所以我们将这个表格处理成一个 16*16 的表格：

```
main()
{
   unsigned char i;
   for(i=0;i<255;i++)
```

```
    {
        if(!(i%16))printf("\n");
        printf("%c",i);
    }
}
```

这个时候我们发现表格中并非我们想象的那样整齐，因为 ASCII 表中还有一些转义字符，所以我们将这些转义字符处理为空格，再做一些简单美化。首先处理转义字符的问题，我们发现只有第一行的 ASCII 码有问题，所以我们就单打印一下前 16 个字符看一下那些是转意字符：

```
main()
{
    char i;
    for(i=0;i<16;i++)
    {
        printf("%d = %c\n",i,i);
    }
}
```



效果图如右图所示：

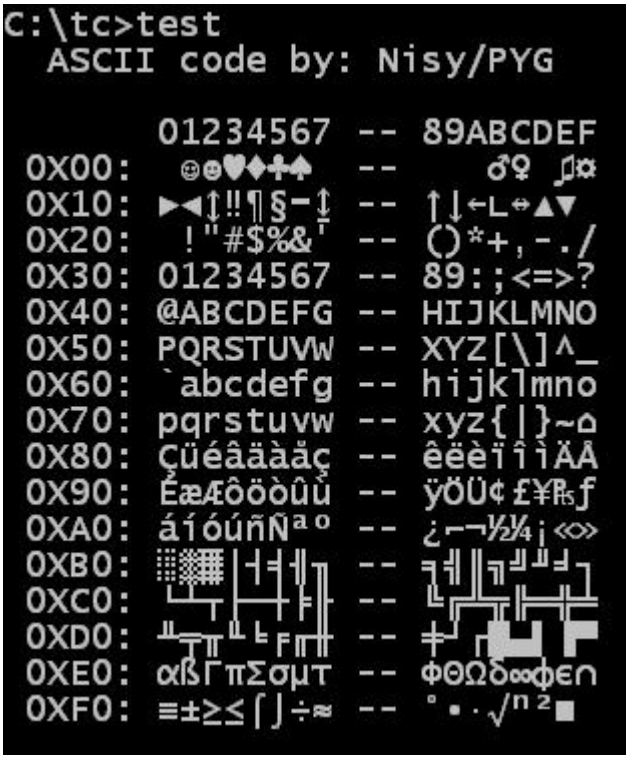
OK，我们找到了转义字符的 ASCII 值，剩下的就好处理了，加上一些优化，最终代码如下：

```
main()
{
        unsigned char i,j=0;
        printf("  ASCII code by: Nisy/PYG\r\n\r\n");
        printf("       01234567 -- 89ABCDEF\r\n");
        for(i=0;i<255;i++)
        {
                if(!(i%8))
                {
                        if(i==0){printf(" 0X%X0:  ",j++);continue;}
                        if(i%16) printf(" -- ");
                        else printf("\n 0X%X0: ",j++);
                }
                if(i==0 || i==7 || i==8 || i==9 || i==10 || i==13)
                {
                        printf(" ");
                        continue;
```

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

效果图如右图所示：

这样我们发现表格就比较整齐了。通过表格我们可以看出字母‘A’在 ASCII 表中的序号为 0X41，字母‘a’在表中的序号为 0X61，差值为 0X20，而且大小写字母间都有这样的对应关系，(*^__^*) 嘻嘻…… 那么我们这个处理大小写字母的程序就可以写出来了。




```
main()
{
   char * str="ChinaPYG!";
   int i;
   puts(str);
   for(i=0;i<strlen(str);i++)
   {
      if( *(str+i)>='a' && *(str+i)<='z')
      *(str+i)-=0X20;
   }
   puts(str);
}
```

因为字符串后面结束符为“0X00”，所以我们也可以通过判断字符是否为 0X00 来检测是否结束循环，如：

```
main()
{
   char * str="ChinaPYG!";
   char * p=str;
   while(*p)
   {
      if(*p >='a' && *p<='z') *p-=0X20;
      p++;
   }
   puts(str);
}
```

下面我们再来看一种处理大小写转化的方法，首先我们来观察一下大小写字母的二进制编码。

‘A’ 的 ASCII 码的二进制：0100 0001

‘a’ 的 ASCII 码的二进制：0110 0001

我们发现两个字符的二进制的异同点就是大写字母的第六位（从低位到高位来数）为 1，而小写字母的第六位为 0。A~Z 有 26 个字母，26 的二进制码为：0001 1010，也不影响该规律。所以我们转化大小写字母就又多了一种方法：处理字母二进制中的第六位。

那 C 语言中有支持处理 bit 的位运算符码？答案的乐观的，C 语言中处理 bit 的指令成为布尔运算符，有这样几种：与运算、或运算、异或运算、求反运算、左移右移运算符，这里我们只讲前三种。

与运算

符号：‘&’。两个运算数据对应的比特位中有一个为零，则结果中对应的比特位就为零，简称一零则零。

例如：

```
        0X41：0010 1001
&       0X37：0010 0101
---------------------------------
=       0X21：0010 0001
```

或运算

符号：‘|’。两个运算数据对应的比特位中有一个为一，则结果中对应的比特位就为一，简称有一则一。

例如：

```
        0X41：0010 1001
|       0X37：0010 0101
---------------------------------
=       0X2D：0010 1101
```

异或运算

符号：‘^’。两个运算数据对应的比特位若相同结果则为零，若不同结果则为一，简称同零异一。（如果不记成同一异零呢？大家只要记住万事万物不能无中生有即可，即 0^0!=1）

例如：

```
        0X41：0010 1001
|       0X37：0010 0101
---------------------------------
=       0X0C：0000 1100
```

有了这个语法知识，我们就可以用布尔运算符来处理大小写的问题了：

```
main()
{
   char str[50];
   char * p=str;
   gets(str);              /* 输入一个字符串，长度不要超过我们申请的空间哦 */
   while(*p)
   {
      if(*p >='a' && *p<='z')
      {
         *p++ &=0XDF;
         continue;
      }
      p++;
   }
   puts(str);
}
```

那大家试着用布尔运算来写一个将大写字母转化为小写字母的程序。

1.3.2for 循环中的思考

当我们掌握了 C 语言的一些基本要素后，我们来看一下这个题目：请计算出从 1 到 102 的数值。大家请先写一下该程序，或者在脑海中思考一下应该如何去写。
可能很多人第一反应就是用 for()循环来计算累加和：

```
main()
{
   int i,sum=0;
   for(i=1;i<=102;i++)sum+=i;
   printf("%d",sum);
}
```

我曾对该题目做过一次调查统计，参与调查的大约有五十几人，只有三个人没有用 for() 循环指令来实现这个累加和，另外有两个用递归来实现的，其他人提交的代码基本上都类似上边的形式。

计算 1~100 的和我们口算就可以得出：5050。计算方法就是(1+100)*100/2。那为何使用 C 语言来计算该题的时候就不用这个简单的方法，而非要 for 循环一下来计算结果呢。程序就是设计者逻辑的一个写照，要先有逻辑才会有程序。一些 C 语言的书籍中在讲 for 循环中用的就是这个求累加和的题目来讲解的，但是作者是为了让大家去了解 for 循环的使用，而不是向我们传达计算整数的累加和还需要 for 一下。我们在看书的时候应该去多思考，不要去死读书，读死书。

作业题：大家试着写一个输入菱形的程序，例如：
输入：9
输出：

```
    *
   * *
  *   *
 *     *
*       *
 *     *
  *   *
   * *
    *
```

大家一定要试着去写一下这个程序，然后再看下一节的内容。

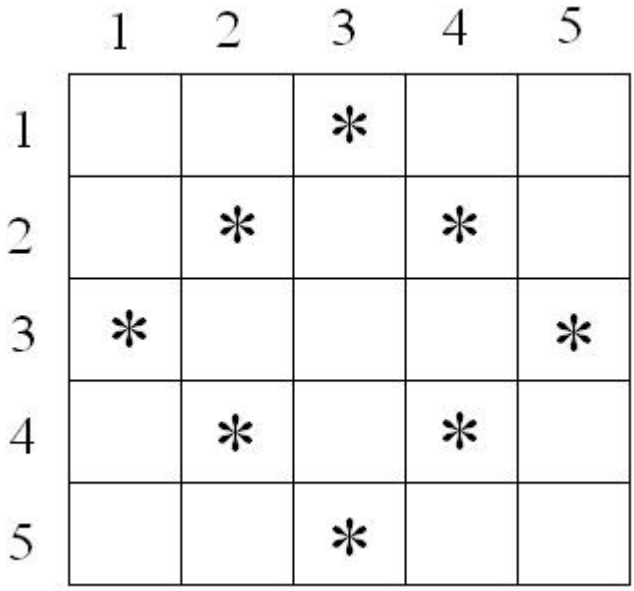
1.3.3关于菱形程序

我们上一节给大家布置的作业是写一个菱形的程序，菱形的概念我们就不介绍了，我们画一张图表来分析其结构，我们首先可以看到菱形沿中轴线上下左右对称。我们再来分析各个“*”在表格中的位置，我们以下图菱形变成=3 为例。

中轴线以上每行“*”的位置 = 边长 – 行数 + 1 。这个规律适合其他行吗？我们来看中轴线以下每行 “*”的位置 = 行数 – 边长 + 1 。那这样我们就可以把规律统一一下了，数轴线左侧“*”的位置 = |边长 – 行数 | + 1

数轴线右侧“*”的位置大家自己分析一下即可。

```
#define L 9
#define H L/2

main()
{
   int i,j;
   for(i=0;i<L;i++)
   {
      for(j=0;j<=L;j++)
      {
         if(j==abs(H-i) || j==L-
abs(H-i)-1) printf("*");
         else if(j==L) printf("\n");
         else printf(" ");
      }
   }
}
```



可能有些人在做这个题目的时候，一开始没有先去思考这个题的逻辑，边写边想，最终导致写了一两个小时还是无法输出菱形。而当大家看到代码后，除了 abs() 求绝对值这个函数没有见过之外，其他的 if() 和 for() 指令大家应该都非常熟悉。有人说我再写程序的时候卡到了绝对值求值上面，那我们看一下绝对值求值的问题：

|5-3|= + (5-3) =2
|3-5|= - (3-5) =2

用 C 语言描述一下就是：

```
int a=5,b=3,num;
if(a>b)num=a-b;
else num=-b+a;
```

也可以写成 num=a>b?a-b:-b+a;

在该问题中我们不使用 abs()这个函数照样可以解答该问题。所以说我们之所以没能够写出这个程序是因为我们脑海中没有实现该程序的一个清晰的逻辑。在写一个程序或函数前，一定是我们先有了这个解题的逻辑，然后再用 C 语言进行表达。于是我们可以得出一个结论，没有逻辑就没有程序，所以可以推出**原程序就是逻辑，然后用语言来描述一下。**

接下来我们看一下 C 语言中最经典的那个 Hello 程序：

```
main()
{
   printf("Hello,HoHo!");
}
```

从该程序中我们可以得到什么？我们的目标是在屏幕上输出一个字符串，那应该如何实现呢？

首先是要有逻辑，逻辑就是要输出一个字符串，然后要了解 C 语言中输入字符串的语法或者格式：使用 printf()函数来实现。我们再深一层的想，该程序就是告诉我们如何学习计算机语言，首先是要有该程序的逻辑，然后用该语言的语法格式描述一下。即：

源程序=人的逻辑+用语言描述一下

下面我们通过一个经典问题来回味一下上文中所定义的程序是什么，百钱买百鸡的问题：有人拿 100 块钱去买 100 只鸡，公鸡 7 元一只，母鸡 5 元一只，小鸡一元 3 只，每种鸡都要买到，问各买多少只鸡。

我们简单的用 C 语言来描述一下：

```
main()
{
   int i,j;
   for(i=1;i<=100-5-1;i++)
   {
      for(j=1;j<=100-7-1;j++)
      {
         if(i*7+j*5+(100-i-j)/3==100 && !((100-i-j)%3))
         printf("%d %d %d \n",i,j,100-i-j);
      }
   }
}
```

语言这个东西是相通的，就象国语中会有“你、我、他”这样的元素，英语中也会有“你、我、他”，法语中也有，德语中也有。如果他缺少这些基本要素，就很难称之为一个语言了。计算机语言亦如此。如果人在解决问题中含有判断和循环，那么计算机语言就需要支持该逻辑，若一种计算机语言中没有，那么就很难去处理需要用人的逻辑来处理的问题。有人会继续深入思考，既然程序是为了解决问题，计算机语言需要支持人在解决问题时用到的逻辑，判断分支有了，循环指令也有了，那人拥有

的联想的能力计算机语言为什么没有呢？我们说，要用我们的逻辑使我们的代码产生联想的能力，而非应该让计算机语言支持联想的指令。

掌握了计算机语言中的变量的本质，如果判断条件和循环指令之后，一个计算机语言的框架我们就搭建起来了，这一章我们介绍了很多内容，除了这些字面上的东西，更重要的要的是要让大家学会去推理分析事物的本质。每一本 C 语言的书籍写的都很好，关键是你是否可以发现文字背后的内容，关键是你只是看了看，还是在研究。传统教育一直是填鸭式的教学模式，然后以考分来论英雄。其实一个人的价值在于你创造了什么，在于你对这个社会做出了哪些贡献。那些所谓考试分数根本就没什么意义，你比别人考试少个几十分很难证明你就是比别人差。因为社会中不需要去检测你的记忆力，更重视你是否具备了深度思考的能力，具备了是否善于发现问题本质的能力。我们一直在提倡创新，创新首先来源于一个人是否拥有深度思考的能力，只有善于去发现问题去探索的人、能够深度思考挖掘问题本质的人才更有创新的能力。而传统教育中很难培养学生深度思考问题的能力，只能是死读书读死书，终日为几个考分而碌碌无为。这就是社会的发展和教育体制中的悲哀。

2 函数及模块化程序设计

2.1 模块化程序设计的必然

我们先来写一个排序的程序，该整形数组中的 10 个数组按从小到大的顺序排列，这里我们使用冒泡法来实现。

```
main()
{
   int i,j,t;
   int a[10]={9,2,3,4,5,6,7,8,1,0};
   for(i=0;i<10;i++)
   {
      printf("%d ",*(a+i));
   }
   printf("\n");
   for(i=0;i<10-1;i++)
   {
      for(j=0;j<10-1-i;j++)
      {
         if(*(a+j)>*(a+j+1))
         {
            t=*(a+j);
            *(a+j)=*(a+j+1);
            *(a+j+1)=t;
         }
      }
   }
   for(i=0;i<10;i++)
   {
      printf("%d ",*(a+i));
   }
   printf("\n");
}
```

我们发现，该程序中输出的这两段代码是相同的：

```
for(i=0;i<10;i++)
{
   printf("%d ",*(a+i));
}
printf("\n");
```

这样编译之后程序中肯定就会有这两段相同的指令，如果一个程序中相同的代码很多，势必程序的体积就会增大，程序加载内存的空间就要增大。如何解决这个问题呢？我们的想法是最好可以共用该段代码。我们最熟悉的函数是 C 语言的 main()函数，先回顾一下 main 函数的模型：

```
main()
{
}
```

其实这里是省略了 main 函数的类型和参数。函数默认的类型为 void 型，可省略不写，main 函数传参我们下文中再做出详细讲解。所以完整的函数模型如下：

```
函数类型 函数名(变量类型 变量名)
{
}
```

我们简单整理下输出数组的代码并借鉴 main 函数来试着写成函数的样子：

```
void Show()
{
   int * a,num,i;
   for(i=0;i<num;i++)
   {
      printf("%d",*(a+i));
   }
   printf("\n");
}
```

分析这个函数，我们只要知道整形数组的地址和长度，就可以输出该数组，即这两个变量是解决该问题的关键所在。我们将这段代码提取出来，目的就在于希望处理该共性的问题“输出整形数组”时，调用此函数便可实现该功能，所以现在问题的关键就是如何将这两个关键数据传递给该函数。

C 语言提供了这种语法支持，通过函数传参来实现数据的传递，调用函数的格式为：函数名（变量）。也可以戏称为变量的函数运算符^_^

定义函数为了突出这些关键变量，同时和调用格式相呼应，我们将这些定义变量的语句提升到括号内，变量间用逗号隔开。

```
void Show(int * a,int num)
{
   int i;
   for(i=0;i<num;i++)
   {
      printf("%d",*(a+i));
   }
   printf("\n");
}
```

需要大家注意的是，虽然格式上是将定义关键变量的语句提升到了括号内，但其本质仍是定义变量，仍是开辟空间。**传参的本质就是赋值语句，就是将参数的数值复制到函数对应的变量新开辟的空间中，以将函数代码补全可以执行。**我们将该排序代码用函数的形式重新描述一下：

```
Show(int * a,int num)
{
   int i;
   for(i=0;i<num;i++)
   {
      printf("%d",*(a+i));
   }
   printf("\n");
}

main()
{
   int i,j,t;
   int a[10]={9,2,3,4,5,6,7,8,1,0};
   Show(a,10);
   for(i=0;i<10-1;i++)
   {
      for(j=0;j<10-1-i;j++)
      {
         if(*(a+j)>*(a+j+1))
         {
            t=*(a+j);
            *(a+j)=*(a+j+1);
            *(a+j+1)=t;
         }
```

```
    }
  }
  Show(a,10);
}
```

这里我们把 Show()函数写在 main 函数上方，当函数写在调用指令所在函数的上方时，我们无需对该函数进行声明，其他情况我们最好在使用前前对其声明。声明该函数的格式为：函数类型 函数名(参数类型); 注意要加“;”号，其中变量名可省略，声明时只需要告诉系统带几个参数各是什么类型即可。

通过上文的分析可以可以得出：**程序模块化设计首先是为了节省程序体积。**

2.2 模块化程序设计的意义

通过上文的分析，我们可以得出程序模块化设计的作用之一就是节约程序空间。我们再来看另一个重要的作用。下面我们将排序的代码提出来也写成一个函数：

```
PaiXu(int * a,int num)
{
  int i,j,t;
  for(i=0;i<num-1;i++)
  {
    for(j=0;j<num-1-i;j++)
    {
      if(*(a+j)>*(a+j+1))
      {
        t=*(a+j);
        *(a+j)=*(a+j+1);
        *(a+j+1)=t;
      }
    }
  }
}
```

观察以下几行代码：

```
if(*(a+j)>*(a+j+1))
{
  t=*(a+j);
  *(a+j)=*(a+j+1);
```

```
    *(a+j+1)=t;
}
```

大括号中的代码就是互换两个数。我们将这几行具有共性的代码提出来进行模块化封装（就是函数一下啦），使我们的程序更有逻辑性。

```
Show(int * a,int num)
{
    int i;
    for(i=0;i<num;i++)
    {
        printf("%d",*(a+i));
    }
    printf("\n");
}

PaiXu(int * a,int num)
{
    int i,j,t;
    for(i=0;i<num-1;i++)
    {
        for(j=0;j<num-1-i;j++)
        {
            Big(a+j,a+j+1);
        }
    }
}

Change(int * a,int * b)
{
    int t;
    t=*b;
    *b=*a;
    *a=t;
}

Big(int * a,int * b)
{
    if(*a>*b)Change(a,b);
}
```

```
main()
{
   int a[10]={9,2,3,4,5,6,7,8,1,0};
   Show(a,10);
   PaiXu(a,10,'b');
   Show(a,10);
}
```

现在再看我们的程序，结构就比较明晰了。我们将从小到大的排序的代码也加进去，在 PaiXu 这个函数中加入一个参数，以来判断排序的种类。

```
Show(int * a,int num)
{
   int i;
   for(i=0;i<num;i++)
   {
      printf("%d",*(a+i));
   }
   printf("\n");
}

PaiXu(int * a,int num,char c)
{
   int i,j,t;
   for(i=0;i<num-1;i++)
   {
      for(j=0;j<num-1-i;j++)
      {
         if(c=='b')Big(a+j,a+j+1);
         if(c=='s')Small(a+j,a+j+1);
      }
   }
}

Change(int * a,int * b)
{
   int t;
   t=*b;
   *b=*a;
```

```
    *a=t;
}

Big(int * a,int * b)
{
    if(*a>*b)Change(a,b);
}

Small(int * a,int * b)
{
    if(*a<*b)Change(a,b);
}

main()
{
    int a[10]={9,2,3,4,5,6,7,8,1,0};
    Show(a,10);
    PaiXu(a,10,'b');
    Show(a,10);
    PaiXu(a,10,'s');
    Show(a,10);
}
```

整个程序就添加了几行代码，貌似没做什么改动，该程序就写完了。因为我们对具有共性的代码进行了封装，使程序的结构更加明晰，代码更具逻辑性，main 函数中的几个函数的堆砌就将程序的整个逻辑清晰的进行了描述，所以模块化程序设计的另一个重要作用就是使我们书写的代码更具有逻辑性、更具可读性。我们设计程序的时候，可以先把大框架搭起来，然后再去处理其中的小细节。

作业：

输入 10 个数，进行排序并输出，顺序依次为：从大到小、从小到大、乱序

例如：0 1 2 3 4 5 6 7 8 9

乱序：5 4 6 3 7 2 8 1 9 0

2.3 如何处理其他函数的局部变量

我们上文中有一个交换两个数的函数：

```
Change(int * a,int * b)
{
```

```
   int t;
   t=*b;
   *b=*a;
   *a=t;
}
```

该函数是用指针来处理的数据，下面我们思考一下，若不用指针可以吗？

```
Change(int a,int b)
{
   int t;
   if(a<b)
   {
      t=b;
      b=a;
      a=t;
   }
}

main()
{
   int a,b;
   scanf("%d %d",&a,&b);
   Change(a,b);
   printf("%d %d",a,b);
}
```

输入：10 20
输出：10 20

我们发现程序并没有按照我们的意图将输入的两个整数按从大到小排序，为什么呢？这个问题就有回归到本书一开始对变量本质的剖析：定义变量就是开辟空间。

我们在 main 函数中定义了变量 a 和 b，相当于在内存中开辟了两个 int 型数据的空间，Change 函数中也定义的这两个变量（参数），开辟了另外两个空间，传参就是对这两个变量赋值。

Change 函数中其实交换的是本函数所开辟空间中的数值，和 main 函数中所定义的变量没有关系。而如果传入的参数是指针，Change 函数中变量存放的数值就是 main 函数中变量 a 和 b 的地址，*a 和 *b 就是 main 函数中的变量 a 和 b 。这样该函数在处理参数时就直接的对外部变量作了处理。

我们处理函数传参时，一定要考虑数据的处理问题。**如果参数非指针类型，该函数就无法处理非全局变量外的其他函数的局部变量。**

2.4 设置屏幕光标

本节我们介绍 C 语言中设置屏幕光标的函数：void gotoxy(int x,int y)。x 和 y 这里相当于屏幕的坐标，我们可以将屏幕想象为 x 坐标和 y 坐标均为正的第四象限。
如：

```
main()
{
   gotoxy(10,15);
   printf("Hello,HoHo!");
}
```

程序先将光标设置到第 15 行第 10 列，即 x=10，y=15。然后在输出字符串。下面我们使用该函数并用模块化程序设计的思想来构造输出菱形的程序：

```
#include "stdio.h"

LingXing(int n)
{
   int y,h;
   h=n/2+1;
   if(n<3 || n>25 || !(n%2))
   {
      printf("Qing Shuru >= 3 qie <= 25 de Qishu!\n");
      return 0;
   }
   clrscr();
   for(y=1;y<=n;y++)
   {
      gotoxy(abs(h-y)+1,y);printf("*");
      gotoxy(n-abs(h-y),y);printf("*");
   }
   printf("\n");
   getchar();
}

main()
```

```
{
   int n;
   char c;
   do
   {
      printf("ShuRu:");
      scanf("%d",&n);
      getchar();
      LingXing(n);
      printf("JiXu?(Y or N):");
      c=getchar();
   }
   while((c|=0x20)=='y');
}
```

2.5 递归函数

现在我们谈一下神秘的“递归函数”。先来看一道阶乘的问题，求 10！。这个问题其实就是求 n 的阶乘，该程序用一个循环就可以实现：

```
for(i=1,sum=1;i<=10;i++)
{
   sum*=i;
}
```

那还有其他方法可以实现吗？我们再来分析一下阶乘的求解：

1!=1

2!=2*1!=2

3!=3*2!=6

……

10!=10*9!=?

当大家看到以上的分析时，一定可以总结出 n!=n*(n-1)!的公式。既然我们有了这个解题的逻辑，我们用 C 语言来描述一下，将这个求解阶乘的公式写成一个函数试试：

```
int f(int n)
{
   if(n==1)return 1;
   return n*f(n-1);
}
```

该函数就是用我们的逻辑用 C 语言做了一下描述。因为程序是逻辑的表达，所以各个函数也是逻辑的表达。我们叫这种**函数内部又调用自身的函数为递归函数。**递归也是函数，所以递归肯定也是人逻辑的表达。

下面我们来看一下汉诺塔的问题。

汉诺塔的故事就是有三个柱子，其中第一个柱子上有规律的上放了一些盘子，小盘子在大盘子的之上。问将这堆盘子通过第二个柱子都搬到最后一个柱子上，要求搬运中每次只能搬一个盘子，且小盘子不能放到大盘子下边，需要搬多少次。

假设三个柱子分别为 A、B、C，要求是从 A 柱通过 B 柱将盘子放到 C 柱上。我们考虑这个问题，首先想到的是当一个盘子时，我们应该如何搬运，直接从 A 搬运到 C。那两个盘子呢，将第一个盘子（通过 C 柱）搬运到 B 柱，然后将最后一个盘子放到 C 柱，最后将第一个盘子搬运到 C 柱。那么 n 个盘子呢？首先将前 n-1 个盘子（通过 C 柱）搬运到 B 柱，然后将最后一个盘子放到 C 柱，最后将这前 n-1 个盘子搬运到 C 柱。

解题方案有了，我们用文字先来描述一下：

```
void mov(int N,char A,char B,char C)
{
  if(N==1)
  {
    将盘子从 A 柱搬到 C 柱
    工作结束
  }
  将前 N-1 盘子从 A 柱通过 C 柱搬到 B 柱
  将前 N-1 盘子从 B 柱通过 A 柱搬到 C 柱
  工作结束
}
```

搬运过程我们用 printf 来打印一下，有了逻辑然后我们用 C 语言来描述一下该函数：

```
mov(int n,char a,char b,char c)
{
  if(n==1){ printf("from %c to %c \n",a,c); return ; }
  mov(n-1,a,c,b);
  printf("from %c to %c \n",a,c);
  mov(n-1,b,a,c);
}

main()
{
  mov(3,'a','b','c');
}
```

汉诺谈的代码还可以继续优化，我们常规考虑该问题的逻辑一般都是先一个盘子，然后是一堆盘子的情况。如果我们出发点是先是 0 个盘子，然后一堆盘子，那代码如何表达呢，大家可以试着写一下。

递归函数和普通的函数没有任何区别，都是用语言来描述一下自己的解题逻辑，如果非要说他特别的话，可能就是他要求程序员对该问题的逻辑把握的更清晰更透彻。而递归所谓的难点可能就是思考一个问题时，应该从哪入手。上文中我们已经给出了每个问题的分析思路了，大多数情况都是从假设变量=1 入手，然后一步一步推出解题的逻辑。

下面我们再来看一题，输入一个字符，输出其回文，如输入：‘C’，输出：“CBABC”。

该题应该如何入手呢？对，从最特殊的情况入手——一个字符时，输入’A’，则输出’A’，用 C 语言描述如下：

```
void f(char c)
{
  if(c=='A'){printf("%c",c);return;}
}
```

那我们继续考虑第二个字符时，输入 B，输出 BAB。要先打印一下该字符’N’，然后打印字符’N-1’（如 A 字符可以用 B-1 来表示），最后再输出字符’N’。我们继续用 C 语言来描述一下：

```
void f(char c)
{
  if(c=='A'){printf("%c",c);return;}
  printf("%c",c);
  f(--c);
  printf("%c",++c);
}
```

两个字符的情况我们解决了，三个、四个字符我们还需要考虑吗？我们用 C 语言描述的是的解题逻辑，这个逻辑解决的就是该问题，所以该函数解决的就是该问题。那又问如果输入’C’的时候我还要一层套一层的去分析一下该函数是如何实现的吗？**逻辑对，函数就对，逻辑错，程序就错。递归函数本身就是一个函数，之所以故作神秘就是因为讲解者没有把他看成一个普通的函数，没有把函数就是逻辑的本质看穿，在讲解时把函数实现递归的过程给层层展开了，而忽视了函数就是逻辑的本质。我们写递归函数时，要把 80%的精力放到逻辑层面，20%的精力编译运行一下程序，看看结果就可以了如果结果不对，肯定是逻辑上出了问题。千万不要去单步去跟踪递归的函数，把精力用到层层嵌套的跟踪上就是背放弃了函数就是逻辑的本质，此时的调试将变得毫无意义。**

如果非要说递归比较复杂，我想难点应该就是在传参上吧，比如有时候我们需要传一个指针的指针，参数却写成了一个指针等等，比如这个回文问题，我们的代码还可以写成如下格式：

```
void f(char c)
 {
   if(c=='A'){printf("%c",c);return;}
   printf("%c",c);
   f(c-1);
   printf("%c",c);
 }
```

这时的 f(c-1); 还和 f(--c); 相同吗？更多时候结果如果错了，而我们大逻辑正确的时候，就应该去注意这些细节的问题了。

还可以写成一行指令，但我们不推荐这种写作方案，代码表达的是我们的逻辑，越容易让自己读懂越好。

```
void f(char c)
{
   (c|0x20)>'a' ? putchar(c), f(c-1), putchar(c) : putchar(c) ;
}
```

至此，递归函数就讲解完了，我想通过对递归函数的分析，大家对“原程序=逻辑 + 语言描述一下”这句话理解更加深刻了。

3 指针详解

3.1 指向二维数组的指针

上文中我们了解到变量的二个属性：地址和长度。下面我们来研究一下指针变量的属性：

```
main()
{
    int  * a;
    char  * b;
    long  * c;
    float * d;
    printf("%d ",sizeof(a));
    printf("%d ",sizeof(b));
    printf("%d ",sizeof(c));
    printf("%d ",sizeof(d));
}
```

输出的结果为（TC2 的编译环境）： 2 2 2 2

通过结果我们推断指针变量在内存中的长度均为 2，情况也确实如此，在 Small 模式下，指针变量开辟的空间长度均为两个字节。既然指针变量存放的都是内存地址，那该地址如何从内存中获取数据呢？获取的长度是多少，是谁来告诉编译器的呢？相信大家猜对了，就是指针变量的类型。指针变量的类型声明了该指针指向数据的类型。

下面我们定义一个二维数组 a[2][5]（数组在内存中是以线性方式依次存放的），并通过一个指针变量来输出字符数据 a[1][1]。

二维数组我们可以将其看做是一维数组，只是一维数组中的每一个元素又是一个一维数组。那如何去定义一个指向二维数组的指针变量呢，我们先来看一下二维数组的格式：char a[ ][ ]; 当一维数组时，我们用 a+i 就可以得到 a[0] 元素的地址，而在二维数组中，a+i 等于 a[i]，相当于数组中第 i 个元素所对应的一维数组的数组名，但此时 a[0]已不再是某一个元素的地址了。对于 char a[2][5]来说 a+1 地址将增加 5（ 增加地址 = 变量类型基本长度 * 第二维长度)，是什么使其地址+1 而实际增长了 5 个字节的地址呢？对，是由于第二维的出现影响了 char 型变量的长度属性，长度属性扩展为：变量类型基本长度 * 第二维长度。

一种语言的语法的命名规范一定是有联系的，既然从变量到数组是用“[ ]”实现的，从一维到二维也是用“[ ]”实现的。我们是否可以效仿其格式来制造一个指向二维数组的指针呢？问题的关键在于如何扩展指针类型的长度属性，我们可以效仿二维数组的扩展方法。在指针变量后加上“[ ]”符号来影响指针的类型属性。即指向二维数组的指针变量的格式为： char (*b)[n]; （ n 为第二维的长度 ）。由于运算符的优先级，在进行对指针变量类型属性扩展前首先要将指针变量用( )进行保护。程序实现如下：

```
main()
{
    char a[2][5]={'C','h','i','n','a','H','e','l','l','o'};
    char (*b)[5]=a;
    printf("%c \n",*(*(b+1)+1));
}
```

输出结果为：e

该程序在内存中的模型图如下：

地址： 数 据 ASCII

XX00: 43 68 69 6E 61 48 65 6C-6C 6F 00 00 00 00 00 00 | ChinaHello.....

我们知道使用指向二维数组的指针变量 b 进行操作时，b+1 相当于地址 + 5 个字节(基本长度 * 第二维长度)，而格式输出中 %c 能访问的只是一个字节，所以在访问被扩展长度后的内存单元时，首先要将其扩展的属性收回，这个操作可以在指向扩展单元后在加 * 来实现，即用之前先赋予特权，使用结束后再收回其特权。 *(b+n) 就可以将扩展的属性回归原本，收回之后的指针属性就由 char (*)[n] 变为 char * 了，此时地址的处理就等同于一维数组了。

注意：这里的属性收回操作是强制的，若不收回扩展属性，我们将无法定位其中的任何数据。如我们如果想输出 a[0][0]这个字符，就需要使用 printf("%c",**b); ，这里显然 **b 不是指向指针的指针。

3.2 指向函数的指针和返回指针的函数

上文中我们介绍了指针变量的类型属性的扩展,我们先做一个总结：

```
int    a;        // 定义   整型     的    变量：
int   *a;        // 定义   整型     的   指针变量：
int   *a[ ];     // 定义   整型     的   指针数组；
int (*a)[ ];     // 定义 指向整型二维数组  的   指针变量：
```

我们可以将指向二维数组的指针变量写成如下格式：

```
 int          ( *a )              [];
```
定义整型　　指针变量　（属性扩展） 指向二维数组

即先定义指针变量，然后对其类型属性进行扩展即可扩充指针变量的属性。下文我们来讨论定义指向函数的指针。

我们先来看一道问题：要求输入三个整形数据，输出最大值，并将这三个数值按从大到小的顺序输出。我们先来看输入两个整数时的情况：

```
main()
{
    int a,b;
    void swap(int *,int *);
    scanf("%d %d",&a,&b);
    swap(&a,&b);
    printf("%d \n",a);
    printf("%d %d\n",a,b);
}

void swap(int *a,int *b)
{
    int c;
    if(*a < *b)
    {
        c=*a;
        *a=*b;
        *b=c;
    }
}
```

我们看到定义一个函数的格式为 void f();，所以我们猜测定义一个指向函数的指针变量只需要先定义一个指针变量，并将其第三属性做相应扩展：void (*a)()；，我们用指向函数的指针变量来实现上方的程序。

```
main()
{
    int a,b;
    int swap(int *,int *);
    int (*p)()=swap;
    scanf("%d %d",&a,&b);
    printf("%d \n",p(&a,&b));        /* p(&a,&b) 函数可以当其返回值来用 */
    printf("%d %d\n",a,b);
}
```

```
int swap(int *a,int *b)
{
    int c;
    if(*a < *b)
    {
        c=*a;
        *a=*b;
        *b=c;
    }
    return *a;          /*  返回最大值  */
}
```

说明两点：
01.函数名等于该函数的首地址,将函数名给指针变量就等于将函数首地址给指针变量。
02.我们调用非 void 型的函数可以其返回值来使用。

了解了定义指向函数的指针变量后，我们再来设计输入三个整型数据的程序：

```
main()
{
    int a,b,c;
    int *swap(int *,int *);
    int *(*p)()=swap;
    scanf("%d %d %d",&a,&b,&c);
    printf("%d\n",*(p(p(&a,&b),&c)));
      /*非 void 型的函数可以当其返回值来用 该语句实现最大值同 a 互换*/
    p(&b,&c);
    printf("%d %d %d",a,b,c);
}

int * swap(int *a,int *b)
{
    int c;
    if(*a < *b)
    {
        c=*a;
        *a=*b;
        *b=c;
    }
```

```
    return a;                    /*  变量 a 是地址 而 *a 为整型数据  */
}
```

我们来看一下这两个指向函数的指针变量：int (*p)(); 和 int *(*p)(); ，貌似有些复杂，我们将其拆开分析或许就会一目了然：

```
        int                    (*p)              ();
函数返回值为整型          指针变量    （属性扩展）指针变量指向函数
        int *                  (*p)              ();
函数返回值为整型指针      指针变量    （属性扩展）指针变量指向函数
```

==> int (*p)();指向返回 整型 变量的 函数的指针
==> int *(*p)();指向返回 整型指针变量的 函数的指针

3.3 指向指针的指针

我们先看一个输出字符串的代码：

```
main()
{
  char * str="ChinaPYG!";
  while(*str)printf("%c",*str++);
}
```

用 while 循环来输出该字符串，char * str 定义了一个 char 型的指针变量。既然 char 是一个类型，我们知道 char * 也是一个类型，也应该可以定义指针变量，指向的数据为 char * 的一个指针（指针和地址的区别就是通过指针可以定位内存地址对应的数据类型及长度）。

```
main()
{
  char * str="ChinaPYG!";
  char * *p=&str;
}
```

这个定义是 OK 的，这里*p 就相当于 str，所以我们将上例的代码进行等价替代：

```
main()
{
  char * str="ChinaPYG!";
  char * * p=&str;
  while(*(*p))printf("%c",*(*p)++);
```

```
}
```

我们将指针变量类型为指针（地址型）的变量叫指针变量，通过上例我们可以得出其实指向指针的指针也没什么复杂的。就是叫着绕口而已，和普通的指针变量一摸一样。

3.4 main 传参

我们开篇的第一个程序介绍过 main 函数，该函数也可以带参数的，参数在我们运行程序的时候给出，进入 cmd，进到纯 DOS 模式（比如先进一下 edit 再退出就可以来到纯 DOS 模式），输入： 程序名 参数一 参数二 参数…… 就可以给 main 函数传参。

main 函数的参数就是在命令行中运行时给出的，我们观察这个传参格式，肯定是一个多参函数，参数均为字符串但个数不知。如果我们自己设计 main 函数的参数情况的话，会如何设计呢？既然是多参函数，能够给出参数个数吗？那如何解决呢？C 语言是这样解决的，main 函数带两个参数，第一个参数为参数个数，第二个参数为一个指针数组的地址。为什么给指针数组的地址呢？大家可以思考一下 ^_^

```
main(int a,int b)
{
   int i;
   for(i=0;i<=a;i++)
   {
      puts(*((void **)b+i));
   }
}
```

定义两个参数，在使用第二个参数的时候告诉编译系统，该变量为一个数组的地址，该数组的数据为指针型（void *）。

输入：

```
test  hello  china
```

输出：

```
C:\TC\TEST.EXE
hello
china
```

因为参数为两个字节，所以我们随便定义一个两个字节的变量即可。而第二个变量在做强制转化时，除了 void * 、char * 、int * 等指针类型，**用 int 型也可，只要告诉编译器该变量为数组的地址，该数组每个元素的长度为两个字节。**如：

```
main(int a,int b)
{
   int i;
   for(i=0;i<=a;i++)
   {
      puts(*((int *)b+i));
   }
}
```

当然这里是有一个前提的，因为 TC2 中是不对 puts( )函数的参数做检测的，我们只需要将内存中数据的关系处理正确即可。我们既然摸清了内幕，当然可以还原回最原始状态了，搞明白是目标，方便自己才是关键。

```
main(int a,char * b[])
{
   int i;
   for(i=0;i<=a;i++)
   {
      puts(*(b+i));
   }
}
```

或

```
main(int a,char ** b)
{
   int i;
   for(i=0;i<=a;i++)
   {
      puts(*(b+i));
   }
}
```

了解 main 函数传参的格式后，我们来写一个通过 main 函数传参计算加减法的程序：

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

void main(int n ,char **p)
{
      float a,b;
```

```
    if(n!=4)
    {
        printf("Please enter 4 strings!");
        return;
    }
    a=atof(*(p+1));
    b=atof(*(p+3));
    if(!(strcmp(*(p+2),"+")))
    {
        printf("%5.2f + %5.2f = %5.2f \n",a,b,a+b);
        return;
    }
    if(!(strcmp(*(p+2),"-")))
    {
        printf("%5.2f - %5.2f = %5.2f \n",a,b,a-b);
        return;
    }
    printf("What is \'%s\' ? \n",*(p+2));
}
```

说明两点：
01.将字符串转化为浮点型数据的函数为 atof 包含在库文件 stdlib.h 中。
02.比较字符串的函数为 int strcmp(char *str1， char *str2); 若 str1==str2 返回 0 ； strcmp 函数包含在库文件 string.h 中。

我们写的这个“小计算器”使用 main 函数传参，带 4 个参数：第一个为程序路径；第二个和第四个为以字符串形式输入的浮点型数据；第三个为运算符。参数间以空格分开。我们程序的设计关键在于对运算符的判断。上文中我们使用了 strcmp 函数来做判断。当然我们也可以将 **(p+2) 赋值为一个字符变量来做判断运算符，但此时我们还需要判断该字符串长度是否为 1 了。

3.5 构造函数指针数组

在上一节中我们设计了一个可计算加减法的计算器，这一节我们将其功能扩充，使其支持加减乘除。我们将程序中加减乘除的运算分别写成函数，并将函数地址保存到一个函数指针数组中。我们知道定义一个指针数组的格式为： char *a[ ]; 那如何将数组中保存的指针属性扩展为函数地址呢？上文中我们也讲过定义一个指向函数的指针变量的格式为：char (*p)(); 那么我们将两种定义格式相融合是否可以构造出函数指针数组呢，格式应该是 char (*p[ ])(); 呢还是 char *p[ ]() 呢？我们来尝试一下：

```
float add(float *a,float *b){return *a+*b;}
float sub(float *a,float *b){return *a-*b;}
float (*p[])()={add,sub};

main()
{
    float a,b;
    scanf("%f %f",&a,&b);
    printf("%5.2f + %5.2f = %5.2f \n",a,b,p[0](&a,&b));
    printf("%5.2f - %5.2f = %5.2f \n",a,b,p[1](&a,&b));
}
```

从上例中我们已经得出定义函数指针数组的格式为：char (*p[ ])(); 即先定义一个指针数组，然后将各元素属性扩展为函数。这里需要注意的是，指针数组中存放的函数指针参数格式需要相同，否则存放时需进行相应格式转化。下面我们来设计这个程序。

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

float add(float *a,float *b){return *a+*b;}
float sub(float *a,float *b){return *a-*b;}
float mul(float *a,float *b){return *a**b;}
float divi(float *a,float *b){return *a/(*b);}

char str[]="+-*/";
float (*f[])()={add,sub,mul,divi};

void main(int n,char **p)
{
    float a,b;
    int find(char *);
    if(n!=4)
    {
        printf("Please enter 4 strings!");
        return;
    }
    a=atof(p[1]);
    b=atof(p[3]);
```

```
    if(find(p[2])!=-1)
        printf("%5.2f %c %5.2f = %5.2f \n",a,str[find(p[2])],b,f[find(p[2])](&a,&b));
    else
        printf("What is \'%s\'",p[2]);
}

int find(char *c)
{
    int i;
    for(i=0;i<4;i++)
    {
        if( str[ i ] == *c )
        return i;
    }
    return -1;
}
```

在程序中我们使用了函数指针数组，在定位指针数组元素时又引入了 find 函数来寻址，函数传参均采用了指针变量。在程序设计中，使用指针将有利于提高程序的执行效率。本例大量的使用函数及指针，目的就在于强化大家对指针的认识，我们对指针理解的越深入，在编写程序时就越轻松自如。

4 数据结构

4.1 数据的封装

C 语言中有一种特殊的数据类型——结构体，程序员可以根据需要，将一些不同类型的数据捆绑到一起，构建成一个新的数据类型。其语法格式为：

```
struct  结构体名
{
        成员类型  成员名
};
```

如我们将屏幕上某点的坐标构建成一个数据类型：

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

大家在定义结构体类型时，一定要注意在“{}”后加上 C 语句的结束符：“;”。这一点说明结构体和函数是不同的。我们利用该结构体来实现对屏幕坐标的设置：

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

SetPonit(struct Ponit * s,int x,int y)
{
   (*s).x=x;
   (*s).y=y;
   gotoxy((*s).x,(*s).y);
}

main()
{
```

```
    struct Ponit s;
    char * str="Hello,Struct!";
    SetPonit(&s,20,10);
    printf(str);
}
```

这里 struct Ponit 就是我们新创建的数据类型，访问其数值的语法格式为：结构体变量名.成员名，我们传参时使用了指向结构体的指针，这里我们将结构体所定义的变量视作普通变量即可，其空间大小一般情况下为该结构体中所有变量类型长度的和。为方便其书写，我们可以将 “ (*s). ” 些为 “s->” ，即：

```
SetPonit(struct Ponit * s,int x,int y)
{
    s->=x;
    s->=y;
    gotoxy(s->.x, s->.y);
}
```

C 语言中可以将数据类型重新命名，比如我们可以将 int 型重命名为 Element，就可以使用 Element 来定义 int 型的数据。

重定义数据类型的语法格式为：typedef 原类型 新类型名 ;

如：typedef int ElementType ; 此时：ElementType a; <==> int a; 等价。鉴于结构体冗长的名字，我们自然想对其进行重命名：

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

typedef struct Ponit Pmzb;
```

或：

```
typedef struct Ponit
{
    int x;
    int y;
}Pmzb;
```

还可以再精简：

```
typedef struct
{
   int x;
   int y;
}Pmzb;
```

写成这种格式的时候可能有些读者就感到费解了，那么我们把他重新整理一下：

```
typedef          int              ElementType;
typedef      struct{ int x; int y; }   Pmzb;
```

我们对比来看一下，最后一种精简其实就是省去了结构体名，直接将封装好的数据命名为 Pmzb。

C 语言中有两个非常核心、非常关键、非常好用的东西，一个是重定义数据类型，另一个是函数指针。下面引一段我 C 语言启蒙老师的解释进行说明：**这两个为什么这么重要？数据的封装，把一些具有独立逻辑的关系的数据项，合起来变成了一个完整的逻辑元素的数据项的一个展现。数据的封装就会产生面向对象，数据的封装为什么重要？因为程序就是要处理逻辑，而这种逻辑通常是联系，把这种密切的逻辑创建成一个逻辑体，这样的话容易人理解，概念很清晰，所以重要。为什么函数指针重要？函数的指针可以给一个函数传函数，函数是什么？是处理它里边的关系。可以给一种处理传一种处理，这使得程序的框架搭建起来非常容易。函数指针使得程序的这种过程的搭建，可以给一个过程传递过程，可以给一个关系传一种关系，关系之间的搭建成动态过程和关系之间的搭建变得非常融合，变得可以任意调节。**

4.2 顺序表

我们先来思考一个问题：输入一个字符串“Hello,World!”，如何实现对其追加（在字符串尾增加新字符）数据、删除其中的某个数据，在第几位数据前插入新数据、遍历所有数据（顺序显示）等操作。象字符串这种在存储空间中（如内存）顺序存储的数据，我们可以称其为顺序表。

为方便解决该问题，我们在内存中开辟一块空间来实现对顺序表中数据的操作。例如在此我们需要处理的对象就是 char 型的数据。对于这块内存结构，首先我们要控制其长度，防止追加时数据溢出。这就需要我们为该空间添加两个属性，一个是已存放的数据长度，一个是总数据长度。那现在再来看该空间，它就需要具备三个关系量：空间的地址、空间中存储的数据量、空间的总长度。所以我们创建一个结构体来处理该问题：

```
typedef char ElementType;

typedef struct
{
   ElementType * buffer;
   int length;
   int max;
}List;
```

然后我们来创建一个结构体，并创建一个空间来处理顺序表中的问题：

```
List * CreateList(int n)
{
   List * lp;
   lp=(List *)malloc(sizeof(List));
   if(!lp)return 0;
   lp->buffer=(ElementType *)malloc(sizeof(ElementType)*n);
   if(!lp->buffer)
   {
      free(lp);
      return 0;
   }
   lp->length=0;
   lp->max=n;
   return lp;
}
```

我们成功创建空间后，就可以通过 lp 对空间进行操控。下面请读者自行完成在线性表中对数据的追加、删除、插入、遍历以及对线性表的清空和销毁等功能函数。

作业：创建一个线性表，长度为 15，

1. 向其追加数据：Hello,World!，并进行遍历。
2. 删除第 1 个字符和第 12 个字符，遍历显示为：ello,World!
3. 追加数据：China，遍历显示为：ello,World! Chin
4. 删除第 7、9、13 个字符，遍历显示为：ello,Wrl! Chi
5. 向第 1 个、第 14 个字符分别插入：’H’、’n’，遍历显示为：Hello,Wrl!Chi
6. 清空并销毁线性表。

4.3 链表和堆栈

我们在处理链表结构时，可以抛开链路层不管（交给计算机负责），链表就可以认为等同于顺序表，只是链表的结构体中不仅需要包含每一个节点的内容，还需要包含下一个节点的指针，这里给出创建链表的函数，其他功能函数请自行完成。

```
typedef char ElementType;

typedef struct node;
{
   ElementType data;
   struct node * next;
}ChainNode;

typedef struct
{
   ChainNode * head;
}List;

List * CreateList()
{
   List * lp;
   lp=(List *)malloc(sizeof(List));
   if(!lp)return 0;
   lp->head=(ChainNOde *)malloc(sizeof(ChainNode));
   if(!lp->head)
   {
      free(lp);
      return 0;
   }
   return lp;
}
```

注意这里：typedef struct{ChainNode * head;} List; 将首节点抽象为一个链表名，使数据更抽象成一条链，而操控上忽略链路层，全然就是一个顺序表。对链表的操控中也需要清空、销毁、插入、删除、追加、取某结点数据等函数，下去之后请大家独立将其实现。

至于堆栈，其结构比较简单，同顺序表的不同点可能就是数据的长度我们在初始化时赋值为“-1”，添加数据前先对存储数据的长度加 1 后再进行赋值。创建堆栈的函数如下：

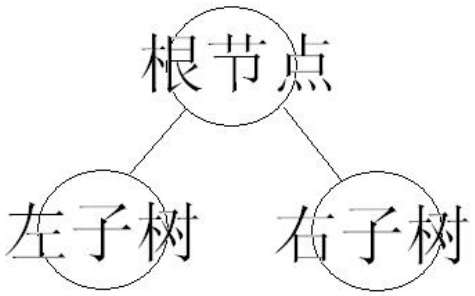
```
Stack * CreateStack(int n)
{
   Stack * sp;
   sp=(Stack *)malloc(sizeof(Stack));
   if(!sp)return 0;
   sp->buffer=(ElementType *)malloc(sizeof(ElementType)*n);
   if(!sp->buffer)
   {
      free(sp);
      return 0;
   }
   sp->top=-1;
   sp->max=n;
   return sp;
}

int Push(Stack * sp,ElementType data)
{
   if(IsFull(sp))return 0;
   sp->top++;
   memcpy(sp->buffer+sp->top,&data,sizeof(ElementType));
   return 1;
}
```

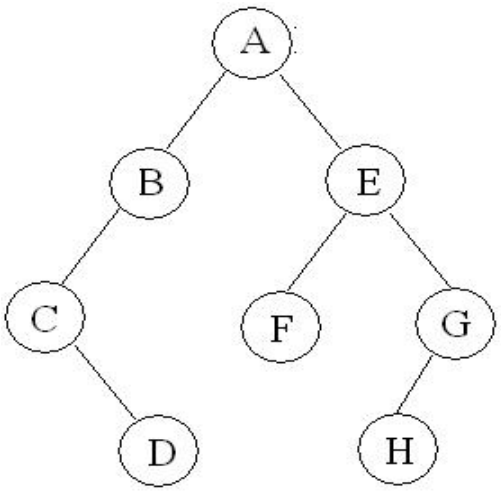
这里给大家实现一个压栈的函数，堆栈是非常重要的树结构之一，其处理数据的思想就是后进先出，凡是数据这种数据类型的关系，我们都可以使用堆栈来轻易的实现

栈结构中的清空、销毁、出栈，取栈顶元素、遍历等函数，也需要大家也要去独立完成。

4.4 二叉树

二叉树可以说其各操作几乎都融入了递归思想，是展现递归思想的代表之一。二叉树有一个根节点，每个节点都有一个左节点和右节点，如下图所示：





所以我们需要创建的结构体，就需要有一个跟节点存放数据，另外还要有一个指向左子树的指针，一个指向右子树的指针。在创建二叉树时，要先对跟节点赋值，然后对其左节点赋值，再对右节点赋值，以此类推。所以我们在创建树的时，用递归是最好的选择。我们创建如右图所示的二叉树。传入数据时，我们对于左、右节点为空的情况在相应字符之后加上“.”作为标志，以方便创建二叉树时作出相应的处理。所以我们在创建二叉树时候赋值的字符串内容要由 “ABCDEFGH”修改为："ABC.D...EF..GH..."。

```
/*
        tree.h 文件
*/

typedef char ElementType;

typedef struct node
{
   ElementType data;
   struct node * lchild;
   struct node * rchild;
}TreeNode;

TreeNode * CreateTree(ElementType ** p)
{
   TreeNode * rootp;
   rootp=(TreeNode *)malloc(sizeof(TreeNode));
```

```
    if(!rootp)return 0;
    if(**p =='.')
    {
        (*p)++;
        return 0;
    }
    memcpy(&rootp->data,(*p)++,sizeof(ElementType));
    rootp->lchild=CreateTree(p);
    rootp->rchild=CreateTree(p);
    return rootp;
}

void PreOrder(TreeNode * rootp)
{
    if(!rootp)return;
    printf("%c",rootp->data);
    PreOrder(rootp->lchild);
    PreOrder(rootp->rchild);
}
```

注意传入字符串的时候传参要用指针的指针，因为若用指针作参数，递归调用时，参数将作为函数的局部变量压栈，指针的自增运算将无法为我们返回正确数据。至于这里，大家可以好好思考一下。

```
/*
        main.c 文件
*/
#include "tree.h"

ElementType * str="ABC.D...EF..GH...";

int Show(TreeNode * rootp,void (*f)(TreeNode *))
{
    f(rootp);
    printf("\n");
}

main()
{
    TreeNode * rootp;
```

```
    rootp=CreateTree(&str);
    if(!rootp)return 0;
    Show(rootp,PreOrder);
}
```

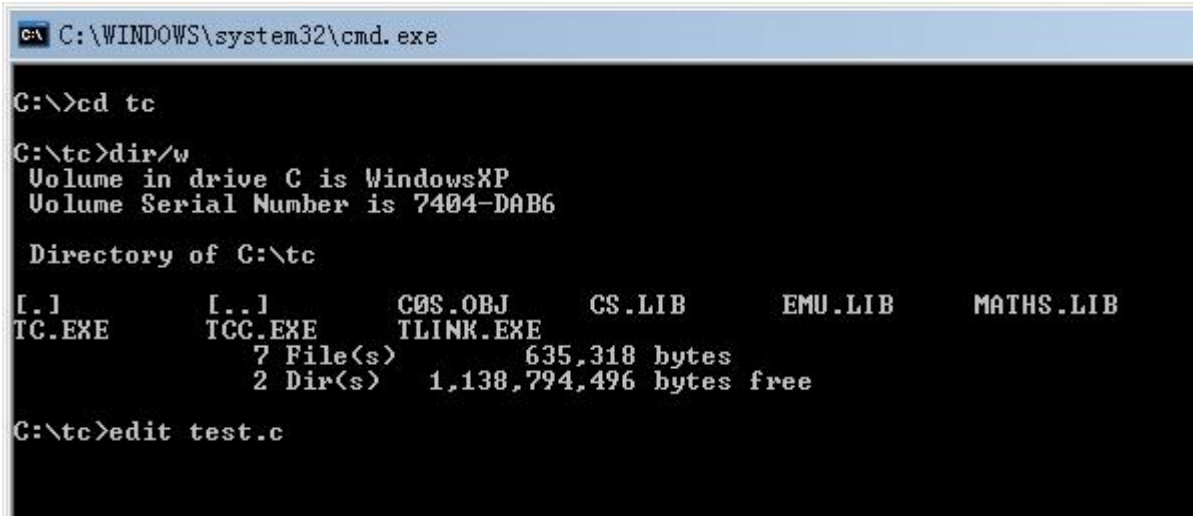
main.c 文件为调用 tree.h 的主文件，用 main.c 来调用二叉树的创建和前序遍历。值得说明的是，我们这里定义的数据 typedef char ElementType; 是 type 型，其他类型包括结构体均可。我们在写代码的时候要有意识的将代码写的通用，比如将赋值语句用 memcpy(); 函数来替代，就可以解决对结构体数据的处理。这样我就可以很轻易的将这些数据结构的源文件做成一个通用的文件，使用时只需要对数据类型进行修改并做一些小调整即可。

二叉树也是数结构中很重要的一个数据关系。如数据库的存储和查询中若使用二叉树效率将会大幅的提高。二叉树中还有一些函数有待大家完成，它们是：中序遍历、后序遍历、求树高、求叶子数、查找某个节点、删除树的某个节点、在树的某个节点中插入树、清空树、销毁树等。

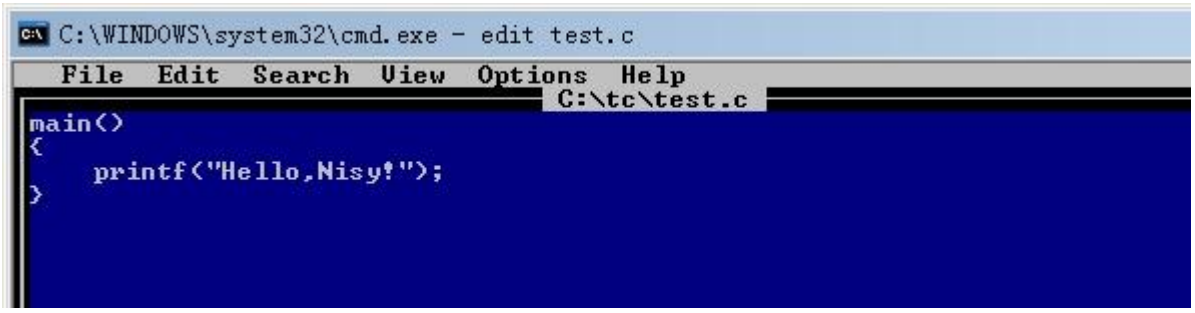
附录 一个简单的 TC 编译环境

先把我上传的 TC2 编译器解压到 C 盘根目录

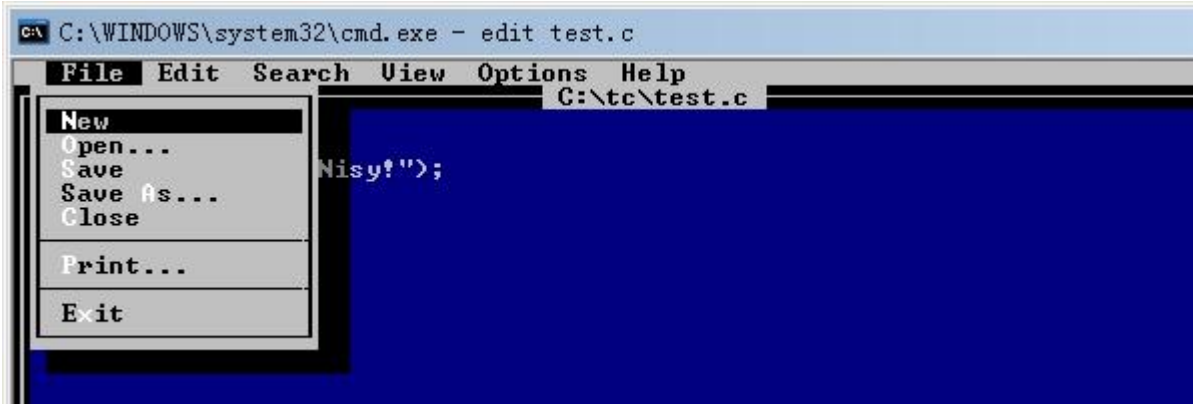
然后 XP 系统下点运行 输入：cmd
进入 DOS 界面后 输入：cd\ 该指令是返回该盘的跟目录 然后输入 cd tc 进入 TC 文件夹 如图 01 所示：



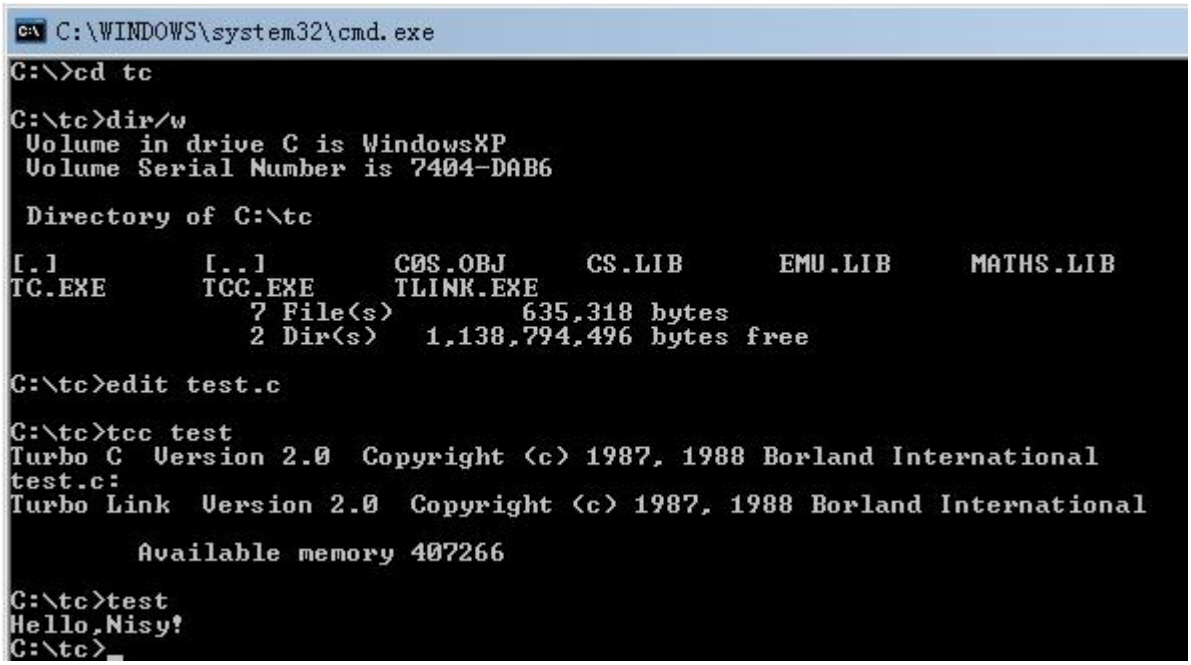
然后我们输入 edit 文件名.c 例如 edit test.c（注意文件要加上 .c 的后缀名） 就进入了 DOS 下 edit 工具的界面
我们在该界面下书写我们的 C 语言程序 如图 02 所示：



当程序书写完毕后 我们按 Alt+F 打开菜单 Alt+S 保存文件 Alt+X 退出 edit 如图 03 所示：



退出 edit 后我们返回到 DOS 主界面 输入 tcc 文件名 如 tcc test 即可编译连接 test.c 文件 编译成功后 在 DOS 下输入我们的文件名即可运行该程序 如图 04 所示：



注意：如果编译时，系统提示缺少.h 文件，请将 turboc2 中 INCLUDE 文件夹中相应的.h 文件复制过来即可。若还有问题请使用完整的 turboc2 进行编译。

附录二　用递归解决数组排序

上文中我们用程序模块化设计思想实现过排序程序，这里我们仍用冒泡法来分析如何使用递归思想来实现排序。我们先考虑按从小到大的顺序排序：
若数组 a[n]中只有一个元素，即 n == 1 时，则 return。
若数组 a[n]中元素个数 > =2 ，则循环处理，若前一个数>后一个数则进行交换。循环次数为 n-1。
前两步后已将最大数置于最后，我们只需重复前两步，每次循环的次数比上一轮少一次。找到该规律后，即可使用递归调用，参数地址不变，循环次数减一。

函数如下：

```
void PaiXu(ElementType * a,int n)
{
   int i;
   if(n==1)return;
   for(i=0;i<n-1;i++)
      if(*(a+i)>*(a+i+1))
        Change(a+i,a+i+1);     // 交换两数 参数为指针
   PaiXu(a,n-1);
}
```

我们继续观察循环部分代码：

```
for(i=0;i<n-1;i++)
      if(*(a+i)>*(a+i+1))
        Change(a+i,a+i+1);
```

该代码的逻辑就是依次比较两个数，将较大数后置。我们分析一下其逻辑：
当只有一个数时，return
当只有两个数时，比较*(a)和*(a+1)
当大于两个数时，比较*(a+1)和*(a+1+1)，即递归调用：将参数地址+1，长度-1。
用 C 来描述一下该过程，即：

```
if(n==1)return;
if(*(a)>*(a+1))
        Change(a,a+1);
PaiXu(a+1,n-1);
```

我们将循环部分替换一下，即可得到：

```
void PaiXu(ElementType * a,int n)
{
   if(n==1)return;
   if(*(a)>*(a+1))
         Change(a,a+1);
   PaiXu(a+1,n-1);
   PaiXu(a,n-1);
}
```

递归排序函数代码有了，我们将程序补全一下：

```
#include <stdio.h>
#define length 10
typedef int ElementType;

void Show(ElementType * a,ElementType n)
{
   int i;
   for(i=0;i<n;i++)
      printf("%d ",*(a+i));
   printf("\r\n");
}

void Change(ElementType * a,ElementType * b)
{
   ElementType t;
   t=*a;
   *a=*b;
   *b=t;
}

void Big(ElementType * a, ElementType * b)
{
   if(*a>*b)
      Change(a,b);
}
```

```
void Small(ElementType * a, ElementType *b)
{
   if(*a<*b)
      Change(a,b);
}

void PaiXu(ElementType * a,int n,char c)
{
   if(n==1)return;
   switch(c)
   {
      case 'b':
         Big(a,a+1);
         break;
      case 's':
         Small(a,a+1);
         break;
      default:
         return;
   }
   PaiXu(a+1,n-1,c);
   PaiXu(a,n-1,c);
}

int main()
{
   ElementType a[]={9,1,2,3,4,5,6,7,0,8};
   Show(a,length);
   PaiXu(a,length,'b');
   Show(a,length);
   PaiXu(a,length,'s');
   Show(a,length);
   return 0;
}
```

程序中用了一些小技巧，如 length、ElementType 的修饰，就是为了提高我们的程序的通用性，当需要对字符型、浮点型数组进行排序时，代码只需要稍做修改即可实现。

该思考题就是为了巩固大家对递归的理解，强化大家逻辑思维和 C 语言表述的能力，**递归和普通的函数一摸一样，用平常心对待即可。逻辑对，程序就对。用递归思**

想去解题，一定要注重程序的逻辑层，切记不要去展开，不要去单步跟踪调试，因为递归就是人的逻辑，去跟踪就等同于放弃了自己的逻辑，追随机器的流程。

附录三　作业两题

学完本书后，请大家自行设计一个汉诺塔的演示程序，效果见附件。这个程序只需要使用栈机制和 gotoxy 函数即可实现。

另外再设计一款小计算器。如输入：123.5+78*64.2-(45+3*7)/11 ， 结果得：5125.10。该程序只需要支持“+ - * / ( )”操作即可。这个计算器的实现上，大家需要动笔来设计一下如何实现。即应该如何解析字符串，如何去处理解析后数据的运算。这两个程序，权当是对本书知识点的一个巩固和总结，大家私下要把这两个程序写熟，力争每个程序都可以控制在半个小时左右完成，完成后若愿意和我交流的话，可将这两款程序发送至我的邮箱：aqiayi@sina.com。

这本书我努力将其写的很精简，是希望读者都能够在短时间内把它看完，如果看完后您觉得有那么一点收获，我也会觉得在打这些文字用去的这些时间是有价值的。任何一个语言是不可能通过只看看书听听课就可以学会的，必须要大量的上机练习，需要靠代码量的累积以驾驭这些语法以流畅的去表达自己的所想。所以希望各位对 C 的兴趣不要只停留在书的层面，多去实践才能真正的去掌握它，最后祝各位在计算机语言的学习中都有所提高。