

《C Primer Plus》读书笔记

By : riusksk (泉哥)

Blog : http://riusksk.blogbus.com

- 1 . scanf()在读取输入时会自动将空字符'\0'插入字符串末尾，而且当它遇到第一个空白字符空格(blank)制表符(tab)或者换行符(newline)时会停止读取，因此使用%s的 scnaf()只会把一个单词而不是把整个语句作为字符串读入，此时我们一般用 gets()来处理一般的字符串。
- 2 . 字符串常量"x"与字符常量'x'不同，'x'属于基本类型(char)而"x"则属于派生类型(char 数组)，另外，"x"实际上是由两个字符('x' 和空字符'\0')组成的。
- 3 . strlen()是以字符为单位给出字符串的长度，其中空字符'\0'并不计算在内，而 sizeof()是以字节为单位给出数据的大小，其中还包括空字符'\0'。
- 4 . 定义符号常量的意义：a. 提供更多的信息，增强代码的可读性；b. 便于更改代码，特别对于在多处使用同一常量而又必须改变它的值时更为适用。可将符号常量名定义为大写字母，当遇到大写的符号名时，就可知道它是一个常量而变量了，比如：

```
#define PI 3.14159
```

这里如果我们这样定义：

```
float pi = 3.14159;
```

由于 pi 是个变量，程序可能意外地改变它的值，因此我们使用#define 来定义它。除了以上方法之外，我们还可以使用 const 修饰符来创建符号常量，此时它就成为只读值，在计算中是不可改变的，比如：

```
const float pi = 3.14159;
```

- 5 . limits.h : 整数限制头文件，float.h : 浮点数限制头文件。例如：

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

int main(void)
{
    printf("Max int value on this system:%d\n",INT_MAX);
    printf("Min int value on this system:%d\n",INT_MIN);
    printf("Max float normal value on this system:%e\n",FLT_MAX);
    printf("Min float normal value on this system:%e\n",FLT_MIN);
    return 0;
}
```

输出结果：

```
Max int value on this system:2147483647
Min int value on this system:-2147483648
Max float normal value on this system:3.402823e+038
Min float normal value on this system:1.175494e-038
```

- 6 . 不匹配的浮点转换实例：

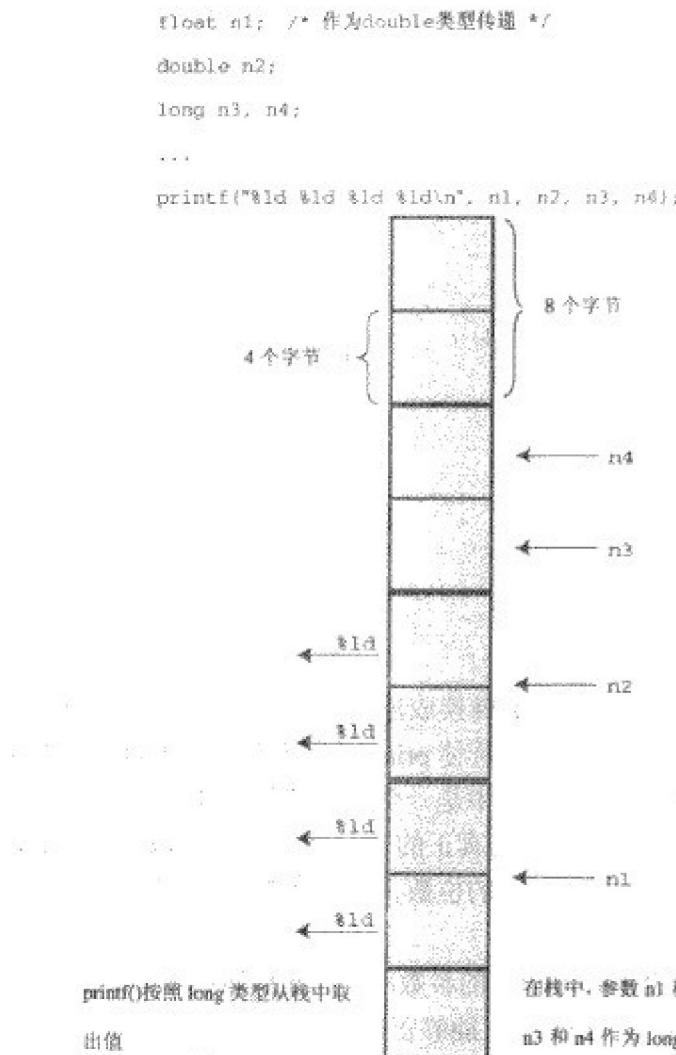


图 4.9 参数传递

n1 在堆栈中占用 8 字节(float 被转换成 double) ,n2 占用 8 字节 , 而 n3 和 n4 则分别占用 4 字节 ,printf() 在读取堆栈中的值时 , 它是根据转换说明符去读取的。 %ld 说明符指出 ,printf() 应该读取 4 个字节 , 所以 printf() 在堆栈中读取前 4 个字节作为它的第一个值 , 即 n1 的前半部分 , 它被解释成一个长整数 (long integer) 。下一个 %ld 说明符再读取 4 个字节 , 即 n1 的后半部分 , 它被解释成第二个长整数 (long integer) 。同样 ,%ld 的第三、四个实例使得 n2 的前半部分和后半部分被读出 , 并被解释成两个长整数 (long integer) 。

7. printf() 返回所打印的字符的数目 如果输出错误 , 则返回一个负数(旧版本的 printf 会有不同的返回值) , 比如 :

```
#include <stdio.h>
int main(void)
{
    int    test = 123;
    int    retval;

    retval = printf("the test value is %d\n",test);
    printf("The printf() function printed %d characters.\n",retval);
    return 0;
}
```

输出结果 :

the test value is 123

The printf() function printed 22 characters.

8. 在 scanf() 格式字符串的说明符中，除了 %c 以外，其它说明符均会自动跳过输入项之前的空格。比如：

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

{
    int a;
    printf("enter:\n");
    scanf("%d",&a);
    printf("%d.\n",a);
    return 0;
}
```

输出结果：

```
enter:
3
3.
```

这里并没有输出空格。又比如：

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

{
    char a;
    printf("enter:\n");
    scanf("%c",&a);
    printf("%c.\n",a);
    return 0;
}
```

输出结果：

```
enter:
a
.
```

这里就是输出空格了。

9. scanf() 函数返回成功读入的项目的个数，如果它没有读取任何项目（当它期望一个数字而你又键入一个非数字字符串时就会发生这种情况），scanf() 会返回值 0。当它检测到“文件结尾”时，它返回 EOF (stdio.h 中将 EOF 定义为 -1)。

10. printf() 和 scanf() 的 * 修饰符：

代码一：

```
//----使用可变宽度的输出字段----
#include <stdio.h>
int main(void)
{
    unsigned width, precision;
    int number = 256;
    double weight = 242.5;

    printf("What field width?\n");
```

```
scanf("%d", &width);
printf("The number is :%*d\n", width, number);
printf("Now enter a width and a precision:\n");
scanf("%d %d", &width, &precision);
printf("Weight = %.*f\n", width, precision, weight);
printf("Done!\n");

return 0;
}
```

变量 width 提供字段宽度，而 number 就是要打印的数字。其运行结果：

```
What field width?
3
The number is :256:
Now enter a width and a precision:
3 7
Weight = 242.500000
Done!
```

代码二：

```
/*跳过输入的头两个整数，此功能可用于读取一个文件中某个特定的列*/
#include <stdio.h>
int main(void)
{
    int n;

    printf("please enter three integers:\n");
    scanf("%*d %*d %d",&n);
    printf("the last integer was %d\n",n);
    return 0;
}
```

输出结果：

```
please enter three integers:
111 222 333
the last integer was 333
```

11. 取模运算符%只用于整数运算，对于浮点数使用该运算符将是无效的。

12. 前缀增量与后缀增量的区别：

先看下面的代码：

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 1;
    int aplus, plusb;

    aplus = a++; /* 后缀 */
    plusb = ++b; /* 前缀 */
    printf("a    aplus    b    plusb\n");
    printf("%1d %5d %5d %5d\n", a, aplus, b, plusb);
```

```
    return 0;  
}
```

运行结果：

```
a    aplus   b    plusb  
2      1      2      2
```

显然，a 和 b 都加一了，但 aplus 是 a 改变之前的值，而 plusb 却是 b 改变之后的值。

再比如， $q=2*++a$; 它会先将 $a+1$ ，然后再 $2*a$ ；而 $q=2*a++$ 却是先 $2*a$ ，再将积加 1。再举个例子：

$b = ++i$ //如果使用 $i++$ ，b 会有不同结果，而如果使用下列语句来代替它：

```
++i; //第 1 行
```

```
b=i; //如果在第 1 行使用了 i++,b 的结果仍会是相同的。
```

13. 增量运算符 $++$ 与 减量运算符 $--$ 具有很高的结合优先级，只有圆括号比它们的优先级高。所以 $x*y++$ 相当于 $(x)*(y++)$ 。
14. 在 $y=(4+x++)+(6+x++)$; 中表达式 $(4+x++)$ 不是一个完整的表达式，所以 C 不能保证在计算子表达式 $4+x++$ 后立即增加 x。这里，完整表达式是整个赋值语句，并且分号标记了顺序点，所以 C 能保证的是在程序进入后续语句前 x 将被增加两次。C 没有指明 x 是在每个子表达式被 计算后增加还是在整个表达式被计算后增加，这就是我们要避免使用这类语句的原因。
15. 请看以下代码：

```
#include <stdio.h>  
int main(void)  
{  
    int i=1;  
    float n;  
  
    while(i++<5)  
    {  
        n = (float)1/i;           //  n = 1.0/i; 如果写成 1/i，则当 i>1 时，n 都会等于 0  
        printf("%f\n",n);  
    }  
    return 0;  
}
```

16. 请看代码：

```
#include <stdio.h>  
#define FORMAT "%s! C is cool!\n"  
int main(void)  
{  
    int num = 10;  
  
    printf(FORMAT,FORMAT);  
    printf("%d\n",++num);  
    printf("%d\n",num++);  
    printf("%d\n",num--);  
    printf("%d\n",num);  
    return 0;  
}
```

运行结果：

```
%s! C is cool!  
! C is cool!
```

```
11  
11  
12  
11
```

17. while 循环语句在遇到第一个分号之后就退出循环，例如以下代码：

```
#include <stdio.h>  
int main(void)  
{  
    int n = 0;  
  
    while (n++ < 3); /* line 7 */  
    printf("n is %d\n", n); /* line 8 */  
    printf("That's all this program does.\n");  
    return 0;  
}
```

输出结果：

```
n is 4  
That's all this program does.
```

由于 while(n++ < 3); 之后存在分号，因此它只是循环的执行 n++，直至它小于 3 才退出循环，相当于一个空语句，退出循环时 n 刚好等于 4。有时，程序员有意地使用带有空语句的 while 语句，因为所有的工作都在判断语句中进行。例如，你想要跳过输入直到第一个不为空格或数字的字符，可以使用这样的循环：

```
while ( scanf( "%d",&num ) == 1 )  
; /*跳过整数输入*/
```

只要输入一个整数，则 scanf() 就返回 1，循环就会继续。

18. math.h 头文件中声明的 fabs() 函数用于返回一个浮点值的绝对值，即没有代数符号的值。

19. 请看代码：

```
#include <stdio.h>  
  
int main(void)  
{  
    int num = 0;  
  
    for (printf("keep entering numbers!\n");num!=6;)  
        scanf("%d",&num);  
    printf("that's the one I want!\n");  
    return 0;  
}
```

输出结果：

```
keep entering numbers!  
1  
2  
3  
4  
5  
6
```

that's the one I want!

20. houseprice = 249,500; 相当于 :

houseprice = 249;

500;

而 houseprice = (249,500);相当于 houseprice =500;

21. 求 $S=1 + 1/2 + 1/4 + 1/8 + 1/16 + \dots$

```
#include <stdio.h>

int main(void)
{
    int t_ct;
    double time,x;
    int limit;

    printf("enter the number of terms you want:");
    scanf("%d",&limit);
    for (time=0,t_ct=1,x=1;t_ct<=limit;t_ct++,x*=2.0)
    {
        time += 1.0/x;
        printf("time = %f when terms = %d.\n",time,t_ct);
    }
    return 0;
}
```

运行结果 :

```
enter the number of terms you want:15
time = 1.000000 when terms = 1.
time = 1.500000 when terms = 2.
time = 1.750000 when terms = 3.
time = 1.875000 when terms = 4.
time = 1.937500 when terms = 5.
time = 1.968750 when terms = 6.
time = 1.984375 when terms = 7.
time = 1.992188 when terms = 8.
time = 1.996094 when terms = 9.
time = 1.998047 when terms = 10.
time = 1.999023 when terms = 11.
time = 1.999512 when terms = 12.
time = 1.999756 when terms = 13.
time = 1.999878 when terms = 14.
time = 1.999939 when terms = 15.
```

22. 代码 :

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

    for ( k = 1,printf("%d:Hi!\n",k) ; printf("k = %d\n",k),
```

```
k*k<26 ; k+=2,printf ( "Now k is %d\n",k )  
    printf ("k is %d in the loop\n",k);  
return 0;  
}
```

运行结果：

```
1 : Hi!  
k = 1  
k is 1 in the loop  
Now k is 3  
k = 3  
k is 3 in the loop  
Now k is 5  
k is 5 in the loop  
Now k is 7  
k = 7
```

23. 让程序要求用户输入一个大写字母，使用嵌套循环产生像下面这样的金字塔图案，比如输入 E:

```
A  
ABA  
ABCBA  
ABCDcba  
ABCDEDCBA
```

提示：使用一个外部循环来处理行，在每一行中使用三个内部循环，一个处理空格，一个以升序打印字母，一个以降序打印字母。源码如下：

```
#include<stdio.h>  
int main(void)  
{  
    int a;  
    char i,j,k;  
  
    printf("请输入字母： ");  
    scanf(" %c",&i); /* 输入的字母用 i */  
    for(k='A' ; k<=i ; k++) /* 输入的字母 i 减掉 A 的数目就是要做的行数 */  
    {  
        for(a=(i-k) ; a>0 ; a--)  
            printf(" "); /* 印出空白字元 */  
  
        for(j='A' ; j<=k ; j++)  
            printf("%c",j); /* 递增印出字母 */  
  
        for(j=k-1 ; j>='A' ; j--)  
            printf("%c",j); /* 递减印出字母 */  
  
        printf("\n");  
    }  
    return 0;  
}
```

24. getchar()没有参数，它返回来自输入设备的下一个字符，比如：ch = getchar(); 相当于 scanf(%c,&ch);

putchar()打印出它的参数，比如 putchar(ch); 相当于 printf("%c",ch);

这两个函数通常在 stdio.h 文件中定义，而且通常只是个预处理宏，而不是真正的函数。

25. ctype.h 系列字符函数：ctype.h 头文件包含了一些函数的原型，这些函数接受一个字符作为参数，如果该字符属于某特定的种类则返回真，否则返回假，比如 isalpha() 函数判断是否为字母。一些映射函数，比如转换为小写字母 tolower() 函数，它并不改变原始的参数，它们只返回改变后的值，也就是说，tolower(ch); //对 ch 没有影响，若要改变 ch，可以这样做：ch = tolower(ch); 。

表 7.1

ctype.h 的字符判断函数

函 数 名	为如下参数时，返回值为真
isalnum ()	字母数字（字母或数字）
isalpha ()	字母
isblank()	一个标准的空白字符（空格、水平制表符或者换行）或者任何其他本地化指定为白色符的字符
isctrl ()	控制符，例如 Ctrl+B
isdigit ()	阿拉伯数字
isgraph ()	除空格符之外的所有可打印字符
islower ()	小写字母
isprint ()	可打印字符
ispunct ()	标点符号（除空格和字母数字外的可打印字符）
isspace ()	空白字符：空格、换行、走纸、回车、垂直制表符、水平制表符、或可能是其他本地化定义的字符
isupper ()	大写字母
isxdigit ()	十六进制数字字符

表 7.2

ctype.h 的字符映射函数

函 数 名	动 作
tolower ()	如果参数是大写字符，返回相应的小写字符；否则，返回原始参数
toupper ()	如果参数是小写字符，返回相应的大写字符；否则，返回原始参数

26.

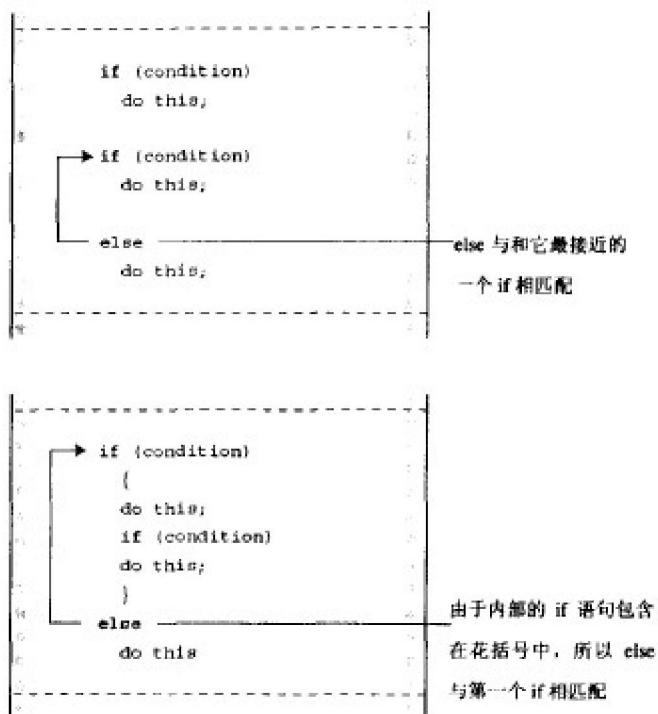


图 7.3 if-else 的配对规则

27. 求 2 到 num 之间的所有数，看它们是否可以整除 num，比如求 144 的约数，由于一次成功的 num%div 测试可以得到两个约数，拿 144 除以 2，可以得到 72，则这两个数均为其约数，这样，我们可以将测试的数据缩小到 num 的平方根即可，而不必到 num。如果测试的数是一个完全平方数，就只需打印出一个数。如果测试的数是一个素数，那么程序流程也进不了 if 语句中，对此我们可设置一个布尔值变量，即标志 (flag)：

```
#include <stdio.h>
#include <windows.h>      // BOOL , TRUE , FALSE 包含在 windows.h 头文件中
int main(void)
{
    unsigned long num;
    unsigned long div;
    BOOL isPrime;

    printf("Please enter an integer for analysis,enter q to quit.\n");
    while (scanf("%lu", &num) == 1)
    {
        for (div = 2, isPrime= TRUE; (div * div) <= num; div++)
        {
            if (num % div == 0)
            {
                if ((div * div) != num)
                    printf("%lu is divisible by %lu and %lu.\n",
                           num, div, num / div);
                else
                    printf("%lu is divisible by %lu.\n",
                           num, div);
                isPrime= FALSE;
            }
        }
    }
}
```

```
if (isPrime && num != 1) //因为 1 不进入以上循环，而 isPrime 被初始化为真，以致其直接执行以下语句，因此要排除 num=1 的情况。  
    printf("%lu is prime.\n", num);  
    printf("Please enter another integer for analysis,enter q to quit.\n");  
}  
printf("Bye!\n");  
return 0;  
}
```

执行结果：

```
Please enter an integer for analysis,enter q to quit.  
144  
144 is divisible by 2 and 72.  
144 is divisible by 3 and 48.  
144 is divisible by 4 and 36.  
144 is divisible by 6 and 24.  
144 is divisible by 8 and 18.  
144 is divisible by 9 and 16.  
144 is divisible by 12.  
Please enter another integer for analysis,enter q to quit.  
1  
Please enter another integer for analysis,enter q to quit.  
q  
Bye!
```

28. 逻辑运算符 : (练习&&时间) ==完美
29. 由于世界各地，并不是所有的键盘都有与美式键盘相同的符号。因此，C99 标准为逻辑运算符增加了可选择的拼写法，它们在 iso646.h 头文件中定义。如果包含了这个头文件，那么就可以用 and 代替&&，用 or 代替||，用 not 代替!。
30. !运算符的优先级仅次于圆括号，&&运算符的优先级高于||，这二者的优先级都低于关系运算符而高于赋值运算，即() > ! > && > ||，对于逻辑表达式是从左至右求值的。
31. 在定义测试范围时，请勿效仿数学上常用的写法：

```
if ( 90 <= range <= 100) // 语义错误  
    printf ("Good show!\n");
```

问题在于该代码存在语义错误，而不是语法错误，所以编译器并不会捕获它（尽管可能会发出警告），因为对<=运算符的求值顺序是由左到右的，所以它会被解释成：

(90 <= range) <= 100

子表达式 90 <= range 的值为 1 或 0，无论是其中任何一个值，它都将小于 100 因此不管 range 的值是什么，整个表达式总为真，所以需要使用&&来检查范围》

```
if ( range >= 90 && range <= 100)  
    printf ("Good show!\n");
```

32. 大量现有代码利用范围测试来检测一个字符是不是（比方说）小写字母。例如，假设 ch 是个 char 变量：

```
if ( ch >= 'a' && ch <= 'z')  
    printf ( "That's a lowercase character.\n");
```

对于 ASCII 那样的字符编码可以工作，因为在这种编码中连续字母的编码是相邻的数值。然而，对于包括 EBCDIC 在内的一些编码就不正确了。进行这种测试的移植性最好的方法是使用 ctype.h 系列中的 islower()函数：

```
if ( islower(ch))
```

```
printf ("That's a lowercase character.\n");
```

33. 计算给定的平方英尺的面积涂油漆，全部涂完需要多少罐油漆。当不知道 1 罐，比如 1.7 罐时，就取 2 罐，用条件运算符即可处理以上这种情况，而且在适当的时候也可用来打印 can 或 cans。

```
#include "stdio.h"  
  
#define COVERAGE 200 /*每罐漆可喷的平方英尺数*/  
  
int main (void)  
{  
    int sq_feet;  
    int cans;  
  
    printf("enter number of square feet to be painted:\n");  
    while (scanf("%d",&sq_feet)==1)  
    {  
        cans = sq_feet/COVERAGE;  
        cans += ((sq_feet% COVERAGE==0)) ? 0 : 1;  
        printf("you need %d %s of paint.\n",cans,  
               cans == 1? "can": "cans");  
        printf("Enter next value (q to quit):\n");  
    }  
    return 0;  
}
```

34. continue 语句只能用在循环体中，它可使程序跳过其余循环的用于处理有效输入的部分，本语句只结束本层本次的循环，并不跳出循环。比如：

```
int main (void)  
{  
    for (n=1;n<=100;n++)  
    {  
        if (n%5!=0)  
            continue;  
        printf("%d",\n);  
    }  
}
```

相当于：

```
int main (void)  
{  
    for (n=1;n<=100;n++)  
    {  
        if (n%5==0)  
            printf("%d",\n);  
    }  
}
```

或者

```
int main (void)  
{  
    for (n=1;n<=100;n++)  
    {
```

```
    if (n%5!=0)
        ;
        //分号
    else
        printf("%d",\n);
    }
}
```

35. 分析以下两份代码的不同：

代码一：

```
#include "stdio.h"
int main (void)
{
    int count = 0;
    char ch;
    for  (count=0;count<10;count++)
    {
        ch = getchar();
        if (ch == '\n')
            continue;
        putchar(ch);
    }
    printf("\n");
    return 0;
}
```

代码二：

```
#include "stdio.h"
int main (void)
{
    int count = 0;
    char ch;
    while (count < 10)
    {
        ch = getchar();
        if (ch == '\n')
            continue;
        putchar(ch);
        count++;
    }
    printf("\n");
    return 0;
}
```

由于 continue 语句会导致循环体的剩余部分被跳过，因此在**代码一中**，当 continue 语句被执行时，首先递增 count，然后把 count 与 10 相比较，这时换行符就包括在计数中了，而在**代码二中**，它读入 10 个字符（**换行符除外**，ch 为换行符时会跳过 count++;语句）并回显它们。当你在代码一中输入 9 个字符，然后回车后就能打印出来。而当你在代码二中输入 9 个字符，然后回车并不能显示出字符，必须再输入 1 个字符以凑足 10 个才会回显。

```
#include <cs.h>
main()
{
    char ch;
    while ( (ch = getchar()) != EOF)
    {
        blahblah(ch);
        if (ch == '\n')
            break;
        yakayak(ch);
    }
    blunder(n, m); ←
}
```

```
#include <cs.h>
main()
{
    char ch;
    while ( (ch = getchar()) != EOF)
    {
        blahblah(ch);
        if (ch == '\n')
            continue;
        yakayak(ch);
    }
    blunder(n, m);
}
```

图 7.4 比较 break 和 continue

37.

```
switch(number)
{
    case 1: statement 1;
    break;
    case 2: statement 2;
    break;
    case 3: statement 3;
    break;
    default: statement 4;
}
statement 5;
```

```
switch(number)
{
    case 1: statement 1;
    case 2: statement 2;
    case 3: statement 3;
    default: statement 4;
}
statement 5;
```

图 7.5 switch 语句中有和没有 break 时的程序流程

38. 在 switch 语句中 ,其圆括号中的判断表达式应该具有整数值(包括 char 类型),case 标签必须是整形(包括 char)常量或整数常量表达式 (公包含整数常量的表达式) 不能用变量作为 case 标签。
39. 只读取一行的首字符 :

```
while ( getchar () != '\n' )
```

continue; //跳过输入行的剩余部分

这个循环从输入读取字符，直到出现由回车键产生的换行符。注意，函数返回值并未赋予任何变量，因此它只是被读取并丢弃而已。因为最后一个被丢弃的字符是换行符，所以下个读入的字符是下一行的首字符。

40. 原则上，C 程序根本不需要使用 goto 语句。但是有一种情况是被许多 C 专业人员所容忍的：在出现故障时从一组嵌套的循环中跳出（单条 break 仅仅跳出最里层的循环）。具有讽刺意味的是 C 不需要 goto，却有一个比其它语言更好的 goto，因为它允许您在标签中使用描述性的单词而不是数字。
41. 编写一个程序读取输入，直到#，并报告序列 ei 出现的次数。

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

int SearchChar(char* SourcesString)
{
    int i,Num=0;

    for (i=0;*(PBYTE)(SourcesString+i)!= '#';i++)
    {
        if (*(PBYTE)(SourcesString+i)=='e' && *(PBYTE)(SourcesString+i+1)== 'i')
        {
            Num++;
        }
    }
    return Num;
}

int main()
{
    char Teststr[]="heiCJKDJKeJHKieihjkklHC#";

    printf("Num=%d\n",SearchChar(Teststr));

    return 0;
}
```

运行结果：

Num=2

42. 缓冲分为两类：完全缓冲 I/O 和行缓冲 I/O。对完全缓冲输入来说，缓冲区满时被清空（内容被发送至其目的地）。这种类型的缓冲通常出现在文件输入中。缓冲区的大小取决于系统，但 512 字节和 4096 字节是常见的值。对行缓冲 I/O 来说，遇到一个换行字符时将被清空缓冲区。键盘输入是标准的行缓冲，因此按下回车键将清空缓冲区。
43. 在 windows 7，VC6 的环境下，将一行的开始位置键入的 Ctrl+Z 解释成文件尾信号。
44. 程序越大，就越应该使用模块化（函数）的方法进行编程。
45. 输入由字符组成，但 scanf() 可以将输入转换成整数或浮点值。使用像 %d 或 %f 这样的说明符时能限制可接受的输入的字符类型，但 getchar() 和使用 %c 的 scanf() 可接受任何字符。
46. getchar(putchar()) 不是一个合法的表达式，因为 getchar() 可以不用参数，而 putchar() 需要一个参数。
47. 在缓冲系统中把数值输入与字符输入相混合时，由于数值输入会忽略空格和换行符，而字符输入不会，因此当先执行字符输入再执行数值输入时，会将换行符发送给后面的数值输入，导致不希望的结果。所以在字符输入之前进行了数字输入时，应该添加代码以在获取字符输入之前剔除换行字符。例如：

```
while ( scanf( "%d", &input ) != 1 )
{
    while ( ( ch = getchar() ) != '\n' )
        putchar( ch ); // 剔除换行字符
    printf( " is not an integer.\n Please enter an " );
    printf( "integer value,such as 25,-178,or 3:" );
}
```

48. 猜测程序：在 1 到 100 之间，猜测数值，比如程序最初猜 50，让其询问用用户该猜测值是大、小还是正确。如果该猜测值小，则令下一次猜测值为 50 和 100 的中值，也就是 75。如果 75 大，则下一次猜测值为 75 和 50 的中值，等等。使用这种二分搜索策略，起码如果用户没欺骗，该程序很快会获得正确答案。

代码：

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int high = 100;
    int low = 1;
    int guess = (high + low) / 2;
    char response;

    printf("Pick an integer from 1 to 100. I will try to guess ");
    printf("it.\nRespond with a y if my guess is right, with");
    printf("\na h if it is high, and with an l if it is low.\n");
    printf("Uh...is your number %d?\n", guess);
    while (scanf("%s", &response) && response != 'y')
    {
        if (response == '\n')
            continue;
        if (response != 'h' && response != 'l')
        {
            printf("I don't understand that response. Please enter h for\n");
            printf("high, l for low, or y for correct.\n");
            continue;
        }

        if (response == 'h')
            high = guess - 1;
        else if (response == 'l')
            low = guess + 1;
        guess = (high + low) / 2;
        printf("Well, then, is it %d?\n", guess);
    }
    printf("I knew I could do it!\n");
    return 0;
}
```

运行结果：

Pick an integer from 1 to 100. I will try to guess it.

Respond with a y if my guess is right, with

a h if it is high, and with an l if it is low.

Uh...is your number 50?

50

I don't understand that response. Please enter h for
high, l for low, or y for correct.

h

Well, then, is it 25?

l

Well, then, is it 37?

n

I don't understand that response. Please enter h for
high, l for low, or y for correct.

y

I knew I could do it!

49. 当函数返回值的类型和声明的类型不相同时，比如：

```
int what_if ( int n)
{
    double z = 100.0 / (double) n;
    return z;
}
```

这时，实际返回值是当把指定要返回的值赋给一个具有所声明的返回类型的变量时得到的数值。因此本例中，执行结果相当于把 z 的数值赋给一个 int 类型的变量，然后返回该数值。

50. 递归举例：

```
#include <stdio.h>
void up (int);

int main(void)
{
    up(1);
    return 0;
}

void up (int n)
{
    printf("Level %d:n location %p \n",n,&n); // 语句#1
    if (n<4)
        up (n+1);
    printf("LEVEL %d: n location %p\n",n,&n); // 语句#2
}
```

输出如下：

```
Level 1:n location 0012FEF8
Level 2:n location 0012FEA0
Level 3:n location 0012FE48
Level 4:n location 0012FDF0
LEVEL 4: n location 0012FDF0
```

```
LEVEL 3: n location 0012FE48
LEVEL 2: n location 0012FEA0
LEVEL 1: n location 0012FEF8
```

分析：首先 main() 使用参数 1 调用了函数 up()。于是 up() 中形参 n=1，故打印语句#1 输出 Level 1。然后，由于 n 的数值小于 4，所以 up()（第 1 级）使用参数 n+1 即数值 2 调用了 up()（第 2 级）。这使得 n 第 2 级调用中被赋值 2，打印语句#1 输出的是 Level 2。与之类似，下面的两次调用分别打印出 Level 3 和 Level 4。当开始执行第 4 级调用时，n 的值是 4，因此 if 语句的条件不满足。这时不再继续调用 up() 函数。第 4 级调用接着执行打印语句#2，即输出 LEVEL 4，因此 n 的值是 4。现在函数需要执行 return 语句，此时第 4 级调用结束，把控制返回给该函数的调用函数，也就是第 3 级调用函数。第 3 级调用函数中前一个执行过的语句是在 if 语句中进行第 4 级调用。因此它开始继续执行其后续的代码，即执行打印语句#2，这将会输出 LEVEL 3。当第 3 级调用结束后，第 2 级调用函数开始继续执行，即输出了 LEVEL 2，以此类推。

51. 递归的基本原理：

第一。每一级的函数调用都有自己的变量。因此上例中程序实际上创建了 4 个独立变量，虽然每个变量的名字都是 n，但它们分别具有不同的值。

变量:	n	ans	n	ans
第 1 级调用后	1			
第 2 级调用后	1	2		
第 3 级调用后	1	2	3	
第 4 级调用后	1	2	3	4
从第 4 级调用返回后	1	2	3	
从第 3 级调用返回后	1	2		
从第 2 级调用返回后	1			
从第 1 级调用返回后				

图 9.4 递归中的变量

第二。第一次函数调用都会有一次返回。当程序流执行到某一级递归的结尾处时，它会转移到前第 1 级递归继续执行。程序不能直接返回到 main() 中的初始调用部分，而是通过递归的每一级逐步返回。

第三。递归函数中，位于递归调用前的语句和各级被调函数具有相同的执行顺序。

第四。递归函数中，位于递归调用后的语句的执行顺序和各个被调函数的顺序相反。

第五。虽然每一级递归都有自己的变量，但是函数代码并不会得到复制。实际上，递归有时可被用来代替循环，反之亦然。

第六。递归函数中必须包含可以终止递归调用的语句。通常情况下，递归函数会使用一个 if 条件语句或其他类似的语句以便当函数参数达到某个特定值时结束递归调用。

52. 使用循环和递归计算阶乘：

```
long fact (int n)      // 使用循环计算阶乘
{
    long ans;
    for (ans =1; n>1 ;n--)
        ans *= n ;
    return ans;
}

long rfact ( int n )    // 使用递归计算阶乘
{
    long ans;
    if ( n>0)
        ans = n* rfack (n-1);
    else
```

```
    ans = 1;
    return ans;
}
```

对于循环和递归，一般来讲，选择循环更好一些。首先，因为每次递归调用都拥有自己的变量集合，所以就需要占有较多的内存；每次递归调用需要把新的变量集合存储在堆栈中。其次，由于进行每次函数调用需要耗费一定的时间，所以递归的执行速度较慢。

53. 以二进制形式输出整数：

```
void to_binary(unsigned long n)
{
    int r;

    r = n % 2;
    if (n>=2)
        to_binary(n/2);
    putchar('0'+ r); // putchar( r ? '1' : '0');
    return;
}
```

54. 斐波纳契数列定义：第一个和第二个数字都是 1，而后续的每个数字是其前两个数字之和。例如，数列中前几个数字是 1、1、2、3、5、8 和 13。代码如下：

```
long Fibonacci ( int n)      // 递归方法
{
    if ( n > 2)
        return Fibonacci (n-1) + Fibonacci (n-2);
    else
        return 1;
}
```

下面使用循环语句来求斐波纳契数列：

```
int f0,f1,f2;
f0=f1=1;
for(i=2;i<=n;++i){
    f2=f0+f1;
    f0=f1;
    f1=f2;
}
printf("f[%d]=%d\n",n,f2);
```

55. 把函数原型和常量定义放在一个头文件中是一个很好的编程习惯。在 UNIX 和 DOS 环境下，指令 #include “file.h”中的双引号表示被包含的文件位于当前工作目录下。

56. *和指针名之间的空格是可选的，通常程序员在声明中使用空格，而在指向变量时将其省略。

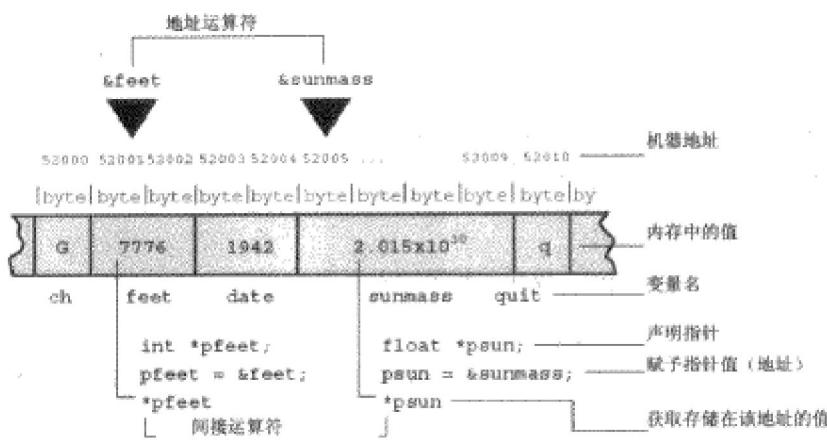


图 9.5 指针的声明和使用

57. 如下调用形式要求函数定义部分必须包含一个和 x 具有相同数据类型的形式参数，如：

```
int function1 ( int num )
```

第二种形式要求函数定义部分的形式参数必须是指向相应数据类型的指针：

```
int function2 ( int * ptr )
```

使用函数进行数据计算等操作时，可以使用第一种调用形式。但是，如果需要改变调用函数中的多个变量的值时，就需要使用第二种调用形式。其实 scanf() 中已经使用了第二种形式。

- 58.

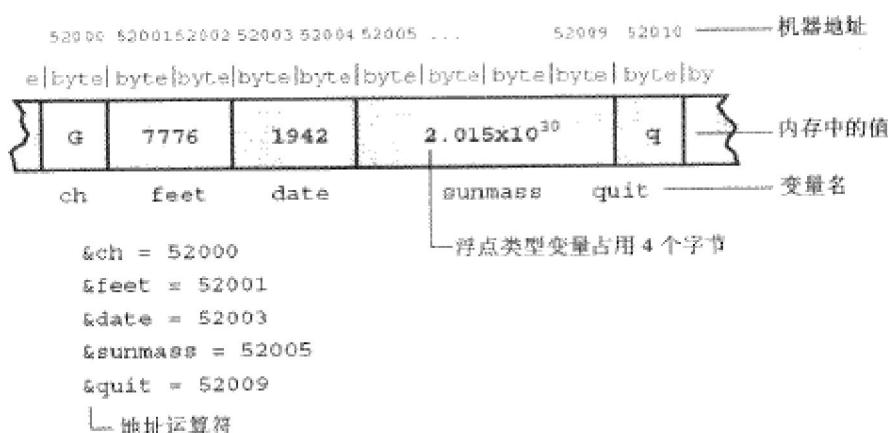


图 9.6 字节编址的系统（如 IBM PC）中的名称、地址和数值

59. 参数用于把调用函数中的数值传递给被调函数。假如变量 a 和 b 的数值分别为 5 和 2，则下面的函数调用语句会把数值 5 和 2 分别传递给变量 x 和 y：

```
int diff ( int x,int y )
{
    int z;
    z = x - y;
    return z;
}
c = diff ( a , b );
```

关键字 return 把函数中的某一数值返回到调用函数中去。变量 c 获得了变量 z 的数值，也就是 3。一般来讲，函数不会改变其调用函数中的变量。当需要在某函数中直接操作其调用函数中的变量时，可以使用指针作为参数。同时，指针参数也可以用来把多个数值返回到调用函数中。

60. 有时需要使用只读数组，也就是程序从数组中读取数值，但是程序不向数组中写数据。在这种情况下声明并初始化数组时，建议使用关键字 const。这样，程序会把数组中每个元素当成常量来处理。和普通变量一样，需要在声明 const 数组时对其进行初始化，因为在声明之后，不能再对它赋值。
61. 按照 C99 规定，在初始化列表中使用带有方括号的元素下标可以指定某个特定的元素：

int arr[6] = { [5] = 212 }; // 把 arr[5] 初始化为 212

指定初始化项目有两个重要特性：第一，如果在一个指定初始化项目后跟有不止一个值，例如在序列 [4]=31,30,31 中这样，则这些数值将用来对后续的数组元素初始化。第二，如果多次对一个元素进行初始化，则最后的一次有效。

62. /* 无效的数组赋值 */

```
#define SIZE 5
int main ( void )
{
    int oxen [SIZE] = {5,3,2,8};
    int yaks [SIZE];

    yaks = oxen;      // 不允许
    yaks[SIZE] = oxen[SIZE]; // 不正确
    yaks[SIZE] = {5,3,2,8}; // 不起作用，C 不支持把数组作为一个整体进行赋值，也不支持用花括号括起来的列表形式进行赋值（初始化的时候除外）

    .....
}
```

64.

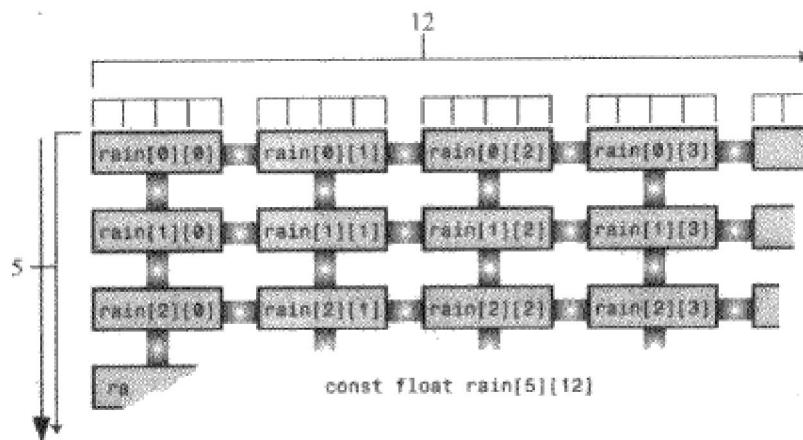


图 10.1 二维数组

65.

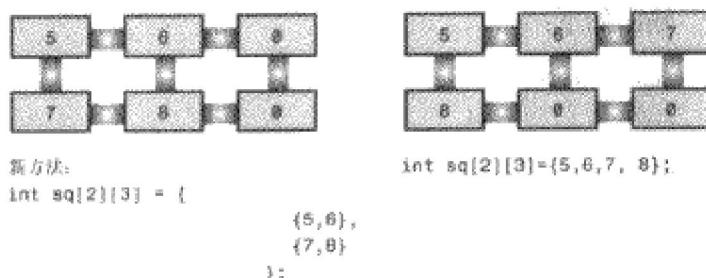
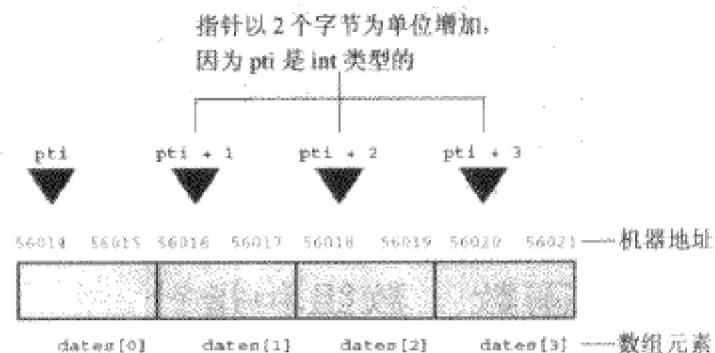


图 10.2 初始化数组的两种方法

66. 数组名是该数组首元素的地址，比如：flizny == &flizny[0]; 两者都是常量，因为在程序运行过程中它们保持不变。

67. 在 C 中，对一个指针加 1 的结果是对该指针增加 1 个存储单元。对于数组而言，地址会增加到下一个元素的地址，而不是下一个字节。因此即使指针是指向标量的，也需要声明指针类型，否则指针操作不能正确返回数值。如下图所示：



```
int dates[y], *pti;  
pti=dates; (或 pti=&dates[0];)
```

▲ 把数组 dates 的首元素的地址赋给指针变量 pti

图 10.3 数组和指针加法

68. 指针的数值就是它所指向的对象的地址。地址的内部表示方式是由硬件来决定的。很多种计算机（包括 PC 机和 Macintosh 机）都是以字节编址的，这意味着对每个内在字节顺序进行编号。对于包含多个字节的数据类型，比如 double 类型的变量，对象的地址通常是指的是其首字节的地址。在指针前运用运算符 `*` 就可以得到该指针所指向的对象的数值。
对指针加 1，等价于对指针的值加上它指向的对象的字节大小。

比如：

```
dates + 2 == &data[2]; /* 相同的地址 */  
* (dates + 2) == dates[2]; /* 相同的值 */
```

69. 正如可以在指针符号中使用数组名一样，也可以在数组符号中使用指针。

```
int sum ( int *ar,int n) // 相当于 int sum ( int ar[],int n)  
{  
    int i;  
    int total = 0;  
  
    for ( i = 0; i <n; i++) // 使用 n 表示元素个数  
        total += ar[i]; // ar[i] 与 *( ar + i ) 相同  
    return total;  
}
```

70. 使用指针参数对一数组的所有元素求和：

```
#include <stdio.h>  
#define SIZE 4  
  
int sum (int * start,int * end );  
int main(void)  
{  
    int mar[SIZE]={23,3,43,3};  
    long answer;  
    answer = sum(mar,mar+SIZE);  
    printf("%d\n",answer);  
    return 0;  
}
```

```
int sum (int * start,int * end )
{
    int total = 0;
    for (;start<end;start++)
        total += *start;
    return total;
}
```

71. 指针操作：

```
#include <stdio.h>
int main(void)
{
    int urn[5] = {100,200,300,400,500};
    int * ptr1, * ptr2, *ptr3;

    ptr1 = urn;          // 把一个地址赋予指针
    ptr2 = &urn[2];      // 同上
                    // 取得指针指向的值
                    // 并且得到指针的地址
    printf("pointer value, dereferenced pointer, pointer address:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);

    // 指针加法
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
           ptr1 + 4, *(ptr1 + 3));
    ptr1++;            // 递增指针
    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);
    ptr2--;            // 递减指针
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
           ptr2, *ptr2, &ptr2);
    -ptr1;             // 恢复为初始值
    ++ptr2;             // 恢复为初始值
    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
    // 一个指针减去另一个指针
    printf("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %d\n",
           ptr2, ptr1, ptr2 - ptr1);
    // 一个指针减去一个整数
    printf("\nsubtracting an int from a pointer:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n",
           ptr3, ptr3 - 2);
```

```
    return 0;  
}
```

运行结果：

pointer value, dereferenced pointer, pointer address:

ptr1 = 0012FF34, *ptr1 = 100, &ptr1 = 0012FF30

adding an int to a pointer:

ptr1 + 4 = 0012FF44, *(ptr1 + 3) = 400

values after ptr1++:

ptr1 = 0012FF38, *ptr1 = 200, &ptr1 = 0012FF30

values after --ptr2:

ptr2 = 0012FF38, *ptr2 = 200, &ptr2 = 0012FF2C

Pointers reset to original values:

ptr1 = 0012FF34, ptr2 = 0012FF3C

subtracting one pointer from another:

ptr2 - ptr1 = 0012FF3C - 0012FF34 = 2 // 指针差值的单位是相应类型的大小，即差值 2 表示指针所指向对象之间的距离为 2 个 int 数值大小，而不是 2 个字节。

subtracting an int from a pointer:

ptr3 = 0012FF44, ptr3 - 2 = 0012FF3C // 将一个整数加（减）给指针时，这个整数都会和指针所指类型的字节数相乘，然后所得的结果会加（减）到初始地址上。

72.

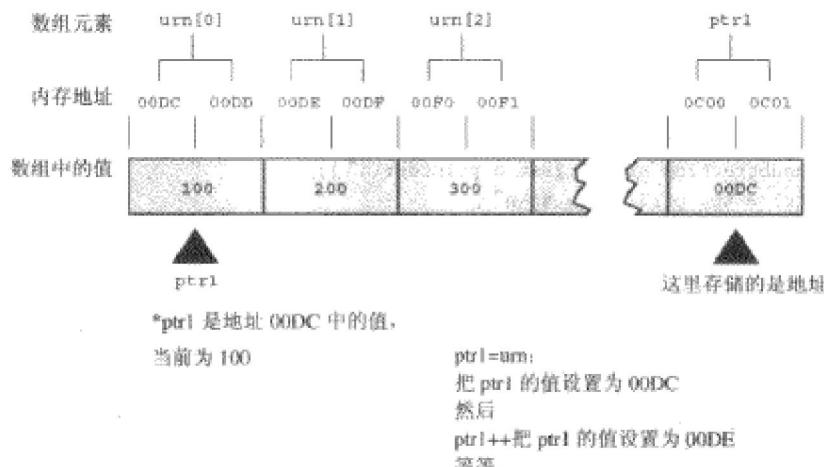


图 10.4 增加一个 int 指针的值

73. 不能对未初始化的指针取值，如：

```
int *pt; // 未初始化的指针  
*pt = 5; // 一个可怕的错误
```

由于 `pt` 没有被初始化，因此它的值是随机的，不知道 5 会被存储到什么位置。这个位置也许对系统危害不大，但也许会覆盖程序数据或者代码，甚至导致程序的崩溃。切记：当创建一个指针时，系统只分配了用来存储指针本身的内存空间，并不分配用来存储数据的内存空间。因此在使用指针之前，必须给它赋予一个已分配的内存地址。比如，可以把一个已存在的变量地址赋给指针（当使用带有一个

指针参数的函数时，就属于这种情况）。或者使用函数 malloc() 来首先分配内存。

74. 下列函数的功能是计算数组中所有元素的和，所以它不应该改变数组的内容。然则由于 ar 实际上是一个指针，所以编程上的错误可以导致原始数据遭到破坏。例如，表达式 ar[i]++ 就会导致每个元素的值增加 1：

```
int sum (int ar[], int n)      // 错误的代码
{
    int i;
    int total = 0;

    for (i=0;i<n;i++)
        total += ar[i]++;
    return total;
}
```

对此，我们可以对形参使用 const，比如：int sum (const int ar[], int n)，这茁壮成长，函数就会当把 ar 所指向的精妙绝伦作为包含常量数据的数组对待。总之，如果函数想修改数组，那么在声明数组参数时就不要使用 const；如果函数不需要修改数组，那么在声明数组参数时最好使用 const。

75. 将常量或非常量数据的地址赋给指向常量的指针是合法的：

```
double rates [5] = { 88.99 , 100.23 , 32.42 , 32.31 , 35.87};
const double locked[4] = {0.08,0.054,0.035,0.07};

const double *pc = rates; // 合法
pc = locked;      // 合法
pc = &rates[3];   // 合法
```

然而，只有非常量数据的地址才可以赋给普通的指针：

```
double rates [5] = { 88.99 , 100.23 , 32.42 , 32.31 , 35.87};
const double locked[4] = {0.08,0.054,0.035,0.07};

double * pnc = rates; // 合法
pnc = locked;        // 非法
pnc = &rates [3];    // 合法
```

76. 使用 const 来声明并初始化指针，这样的指针仍然可用于修改数据，但它只能指向最初赋给它的地址。

比如：

```
double rates [5] = { 88.99 , 100.23 , 32.42 , 32.31 , 35.87};
double * const pc = rates; // pc 指向数组的开始处
pc = &rates[2]; // 不允许
*pc = 92.99;   // 可以，更改 rates[0] 的值
```

可以使用两个 const 来创建指针，这个指针即不可以更改所指向的地址，也不可以修改所指向的数据：

```
double rates [5] = { 88.99 , 100.23 , 32.42 , 32.31 , 35.87};
const double * const pc = rates; // pc 指向数组的开始处
pc = &rates[2]; // 不允许
*pc = 92.99;   // 不允许
```

77. 指针和多维数组：

```
#include <stdio.h>
int main(void)
{
    int zippo [4][2] = { {2,4}, {6,8},{1,3},{5,7} };

    printf(" zippo = %p, zippo+1 = %p\n",zippo,zippo+1);
    printf(" zippo[0] = %p, zippo[0] + 1 = %p\n",zippo[0],zippo[0] + 1);
```

```
printf(" * zippo = %p , *zippo + 1 = %p\n",*zippo , *zippo + 1);
printf(" zippo[0][0] = %d\n",zippo[0][0]);
printf(" * zippo[0] = %d\n",*zippo[0]);
printf(" ** zippo = %d\n",**zippo);
printf(" zippo[2][1] = %d\n",zippo[2][1]);
printf(" *(zippo+2)+1=%d\n",*(zippo+2)+1);

return 0;
}
```

运行结果：

```
zippo = 0012FF28, zippo+1 = 0012FF30
zippo[0] = 0012FF28, zippo[0] + 1 = 0012FF2C
* zippo = 0012FF28 , *zippo + 1 = 0012FF2C
zippo[0][0] = 2
* zippo[0] = 2
** zippo = 2
zippo[2][1] = 3
*(zippo+2)+1=3 // 相当于 zippo[2][1]
```

分析：因为 zippo[0]是其首元素 zippo[0][0]的地址，所以*(zippo[0])代表存储在 zippo[0][0]中的数值，即一个 int 数值。同样，*zippo 代表其首元素 zippo[0]的值，但是 zippo[0]本身就是个 int 数的地址，即&zippo[0][0]，因此*zippo 是&zippo[0][0]。对这两个表达式同时应用取值运算符将得到**zippo 等价于*&zippo[0][0]，后者简化后即为一个 int 数 zippo[0][0]。简言之，zippo 是地址的地址，需要两次取值才可以得到通常的数值。地址的地址或指针的指针是双重间接的典型例子。

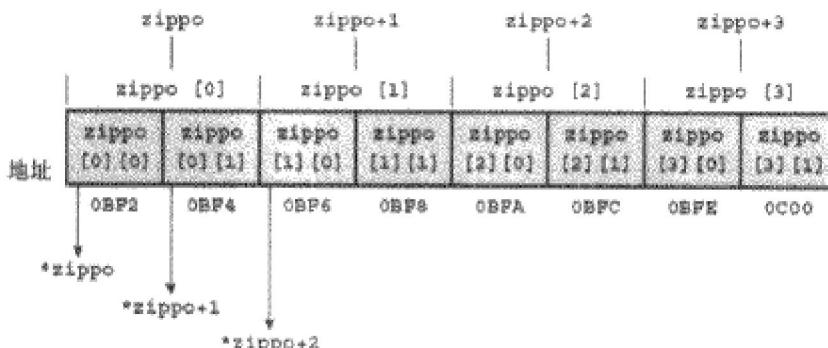


图 10.5 数组的数组

78. 在声明指向二维数组的指针变量时，该指针必须指向一个包含两个 int 值的数组，而不是指向一个单个 int 值。下面是正确的代码：

```
int (* pz )[2]; // pz 指向一个包含 2 个 int 值的数组
```

79. 声明处理二维数组的函数的形式：

```
void sum (int ar[][4], int 3); // 空的方括号表示 ar 是一个指针，而 ar[ ][ ]的声明是错误的
void sum (int [] [4], int ); // 可以省略名称
void sum (int (*ar)[4], int 3); // 另一种语法形式
```

80. 一般地，声明 N 维数组的指针时，除了最左边的方括号可以留空之外，其他都需要填写数值。

```
int sum4d ( int ar[][12][34][56],int rows);
```

相当于：

```
int sum4d ( int (*ar)[12][34][56],int rows); // ar 是一个指针
```

因为首个方括号表示这是一个指针，而其他方括号描述的是所指向对象的数据类型。

81. 声明带有一个二维变长数组参数的函数:

```
int sum2d( int rows, int cols, int ar[rows][cols]); // ar 是一个变长数组
```

在参数列表中，rows,cols 的声明需要早于 ar。在 C99 标准中规定，可以省略函数原型中的名称，但是如果省略名称，则需要用星号来代替省略的维数：

```
int sum2d( int , int , int ar[*][*]); // ar 是一个变长数组 (VLA)，其中省略了维数参量的名称
```

82. 对于数组来说，复合文字(VC6 为不支持)看起来像是在数组的初始化列表前面加上用圆括号括起来的类型名。例如：

```
( int [2] ) {10, 20}; // 一个复合文字
```

由于这些复合文字没有名称，因此不可能在一个语句中创建它们，然后在另一个语句中使用。而是必须在创建它们的同时通过某种方法来使用它们，一种方法是使用指针保存其位置。比如：

```
int * pt1;  
pt1 = ( int [2] ){ 10,20 }; // 此时*pt1=10, pt1[1]=20
```

83. 数组与指针的主要区别：数组名是个常量，但数组元素是变量，而指针则是个变量。因此，不允许对数组名进行递增或递减运算，而指针则可以。

84. 矩形数组与指针数组：

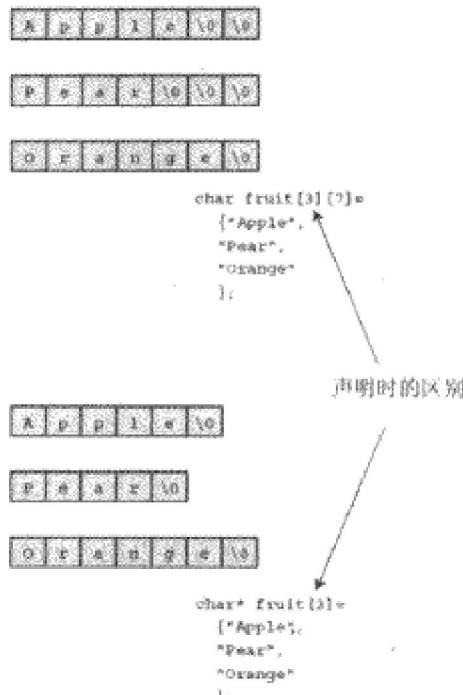


图 11.2 矩形数组和不规则数组

第一种方法表示一个 char 数组的数组，而第二种则是一个指向 char 的指针的数组，两者之间主要的区别在于第一种方法建立了一个所有行的长度都相同的数组，而第二种方法则是建立一个不规则的指针数组。

85. 代码：

```
#include <stdio.h>  
  
int main(void)  
{  
    char side1[] = "side a"; // 书中是使用单引号，但我在 VC6 下编译错误，只能双引号  
    char dont[] = {'w','o','w','!'};  
    char side2[] = "side b";  
    puts (dont); // dont 不是一个字符串
```

```
    return 0;  
}
```

运行结果：

```
wow!side a
```

分析：与 printf() 不同，puts() 显示字符串时自动在其后添加一个换行符。但由于 dont 缺少一个表示结束的空字符，因此它不是一个字符串，这样 puts() 就不知道应该到哪里停止。它只是一直输出内存中 dont 后面的字符，直到发现一个空字符。

86. fgets() 与 gets() 函数的区别：

fgets 需要第二个参数来说明最大读入字符数。如果这个参数值为 n, fgets() 就会读取最多 n-1 个字符或者读完一个换行符为止，由这二者中最先满足的那个来结束输入。 gets() 读取换行符之前（不包括换行符）的所有字符，并在这些字符后添加一个空字符（\0）。

如果 fgets() 读取到换行符，就会把它存到字符串里，而不是像 gets() 那样丢弃它。

它还需要第三个参数来说明读哪一个文件。从键盘上读取数据时，可以使用 stdin 作为该参数，这个标识符在 stdio.h 定义。

87. fputs() 与 puts() 的区别：

fputs() 需要第二个参数来说明要写的文件。可以使用 stdout 作为其参数来进行输出显示，stdout 在 stdio.h 中定义。

与 puts() 不同，fputs() 并不为输出自动添加换行符。

88. 在 while 循环中使用下面的判断条件：

```
while (*string) // 相当于 while ( string[i] != '\0' )
```

当 string 指向空字符时，*string 的值为 0，这将结束循环。

89. 声明一个数组将为数据分配存储空间，而声明一个指针只为一个地址分配存储空间。

90.

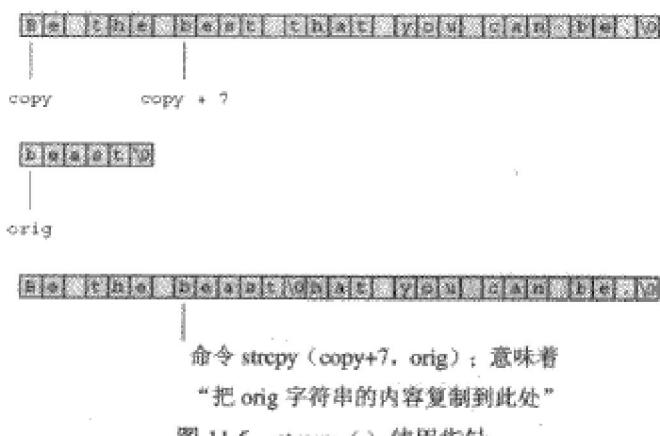


图 11.5 strcpy() 使用指针

ps = strcpy(copy+7, orig); , 则 ps 指向 copy 的第 8 个元素 (索引为 7) , 因此 puts(ps) 将从这个地方开始输出字符串。

91.

ANSI C 库有 20 多个处理字符串的函数，下面的列表总结了其中最常用的一些：

- char *strcpy (char * s1, const char * s2);

该函数把 s2 指向的字符串（包括空字符）复制到 s1 指向的位置，返回值是 s1。

- char *strncpy (char * s1, const char * s2, size_t n);

该函数把 s2 指向的字符串复制到 s1 指向的位置，复制的字符数不超过 n 个。返回值是 s1。空字符后的字符不被复制。如果源字符串的字符数少于 n 个，在目标字符串中就以空字符填充。如果源字符串的字符数大于或等于 n 个，空字符就不被复制。返回值是 s1。

● `char *strcat (char * s1, const char * s2);`

s2 指向的字符串被复制到 s1 指向字符串的结尾。复制过来的 s2 所指字符串的第一个字符覆盖了 s1 所指字符串结尾的空字符。返回值是 s1。

● `char *strncat (char * s1, const char * s2, size_t n);`

s2 字符串中只有前 n 个字符被追加到 s1 字符串，复制过来的 s2 字符串的第一个字符覆盖了 s1 字符串结尾的空字符。s2 字符串中的空字符及其后的任何字符都不会被复制，并且追加一个空字符到所得结果后面。返回值是 s1。

● `int strcmp (const char * s1, const char * s2);`

如果 s1 字符串在机器编码顺序中落后于 s2 字符串，函数的返回值是一个正数；如果两个字符串相同，返回值是 0；如果第一个字符串在机器编码顺序中先于第二个字符串，返回值是一个负数。

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

该函数的作用和 strcmp () 一样，只是比较 n 个字符后或者遇见第一个空字符时会停止比较，由二者中最先被满足的那个条件终止比较过程。

● `char *strchr (const char * s, int c);`

该函数返回一个指向字符串 s 中存放字符 c 的第一个位置的指针（标志结束的空字符是字符串的一部分，因此也可以搜索到它）。如果没找到该字符，函数就返回空指针。

● `char *strpbrk (const char * s1, const char * s2);`

该函数返回一个指针，指向字符串 s1 中存放 s2 字符串中的任何字符的第一个位置。如果没找到任何字符，函数就返回空指针。

● `char *strrchr (const char * s, int c);`

该函数返回一个指针，指向字符串 s 中字符 c 最后一次出现的地方（标志结束的空字符是字符串的一部分，因此也可以搜索到它）。如果没找到该字符，函数就返回空指针。

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

该函数返回一个指针，指向 s1 字符串中第一次出现 s2 字符串的地方。如果在 s1 中没找到 s2 字符串，函数就返回空指针。

● `size_t strlen (const char * s);`

该函数返回 s 字符串中的字符个数，其中不包括标志结束的空字符。

注意，这些原型使用关键字 `const` 来指出哪个字符串是函数不能改动的。例如，考虑下面这个原型：

`char *strcpy (char * s1, const char * s2);`

这意味着 s2 指向一个不可改变的字符串，至少 strcpy () 函数不会改变它，但是 s1 指向的字符串却可以改变。这是因为，s1 是需要改变的目标字符串，而 s2 是不应当有改变的源字符串。

92. ctype.h 中的 ispunct() 函数用于测试字符是否为标点符号或特殊符号。

93.

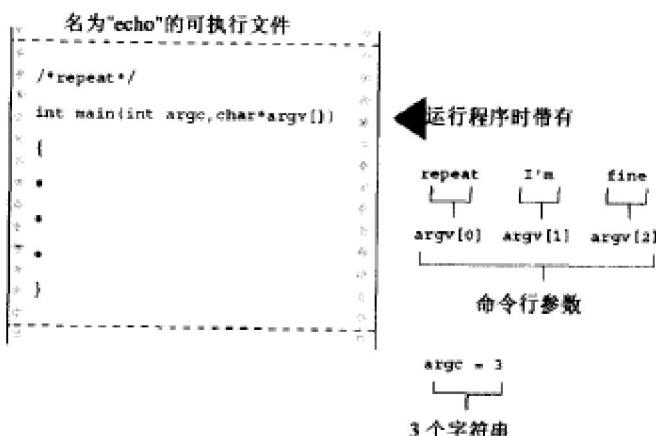


图 11.7 命令行参数

在 DOS 下允许使用引号把多个单词集中在一个参数里。例如：

```
D:\>example.exe "i am riusksk" quange now
the command line has 3 arguments;
1: i am riusksk
2: quange
3: now
```

94. 命令行参数是以字符串形式被读取的，若想在参数中使用数字值，就必须先把字符串转换成数字。如果数字是个整数，那就可以使用 atoi ()函数，该函数包含在 stdlib.h 头文件中。atoi ()函数以字符串为参数，返回相应的整数值，当遇到非数字字符时，就会停止转换，并返回 0。

代码：

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
{
    int i,times;

    if (argc<2 || (times = atoi (argv[1]))<1)
    {
        printf("usage:%s positive-number\n",argv[0]);
    }
    else
        for(i=0;i<times;i++)
            puts("hello,good looking");

    return 0;
}
```

运行结果：

```
D:\>example.exe 3
hello,good looking
hello,good looking
hello,good looking
```

分析：命令行参数 3 以字符串“3\0”的形式存放。atoi ()函数把这个字符串转换成整数 3，然后将其赋予 times。

95. 假定有下列声明：

```
char sign = '$';
```

则 sign 存储需要多少字节？'\$'呢？"\$"呢？

分析：字符变量占用一个字节，所以 sign 占用一个字节。但是字符常量是被存储在一个 int 中的，也就是说'\$'通常会使用 2 个或 4 个字节；但是实际上只使用 int 的一个字节来存储'\$'的编码。字符串"\$"使用两个字节，一个用来保存'\$'，另一个用来保存'\0'。

96. C 变量具有下列链接：外部链接，内部链接，空链接。具有代码块作用域或者函数原型作用域的变量有空链接，意味着它们是由其定义所在的代码块或函数原型所私有的。具有文件作用域的变量可能有内部或者外部链接。一个具有外部链接的变量可以在一个多文件程序的任何地方使用。一个具有内部链接的变量可以在一个文件的任何地方使用。判断文件作用域变量具有内部链接还是外部链接，主要看它是否使用了存储类说明符 static:

```
int giants = 5; // 文件作用域，外部链接
static int dodgers = 3; // 文件作用域，内部链接
```

表 12.1

5 种存储类

存 储 类	时 期	作 用 域	链 接	声 明 方 式
自动	自动	代码块	空	代码块内
寄存器	自动	代码块	空	代码块内， 使用关键字 register
具有外部链接的静态	静态	文件	外部	所有函数之外
具有内部链接的静态	静态	文件	内部	所有函数之外， 使用关键字 static
空链接的静态	静态	代码块	空	代码块内， 使用关键字 static

98. 不同于自动变量，只可以用常量表达式来初始化文件作用域变量（全局变量）：

```
int y = 2 * x; // 不可以，x 是一个变量
```

99. 不要用关键字 extern 来进行外部定义，只用它来引用一个已经存在的外部定义。一个外部变量只可进行一次初始化，而且一定是在变量被定义时进行。下面的语句是错的：

```
Extern char permis = 'y'; // 错误
```

因为关键字 extern 的存在标志着这是一个引用声明，而非定义声明。

100. 事实上，rand() 是一个“伪随机数发生器”，这意味着可以预测数字的实际顺序（计算机不具有自发性），但这些数学在可能的取值范围内均匀地分布。

101. 产生随机数：

```
static unsigned long int next = 1; // 种子
int rand0 ( void )
{
    // 产生伪随机数的魔术般的公式
    next = next * 1103515245 + 12345;
    return (unsigned int )(next/65536) % 32768; // 返回 0 到 32767 范围内的任意值
}
```

102. ANSI C 标准使用了一个新类型：指向 void 的指针，被用作“通用指针”。

103. 为数组动态分配存储空间：

```
#include "stdio.h"
#include "stdlib.h"
int main(void)
{
    double *ptd;
    int max;
    int number;
    int i=0;

    puts("what is the max number of type double entries?");
    scanf("%d",&max);
    ptd = (double *)malloc(max * sizeof(double));
    if (ptd == NULL)
    {
        puts("memory allocation failed.");
        exit(EXIT_FAILURE);
    }
    puts("enter the values(q to quit):");
    while(i<max && scanf("%lf",&ptd[i])==1)
        i++;
    printf("here are your %d entries:\n",number = i);
    for (i=0;i<number;i++)
```

```
{  
    printf("% 7.2f",ptd[i]);  
    if(i % 7 == 6)  
        putchar('\n');  
    }  
    if(i % 7 != 0)  
        putchar('\n');  
    puts("Done.");  
    free(ptd);  
  
    return 0;  
}
```

运行结果：

```
what is the max number of type double entries?  
8  
enter the values(q to quit):  
32 43 53 34 53 75 43 21 56 78  
here are your 8 enteries:  
32.00 43.00 53.00 34.00 53.00 75.00 43.00  
21.00  
Done.
```

使用动态数组主要是获得程序灵活性。假定知道一个程序在大多数情况下需要的数组元素不超过 100 个；而在某些情况下，却需要 10000 个元素。在声明数组时，不得不考虑到最坏情形并声明一个具有 10000 个元素的数组。在多数情况下，程序将浪费内存。如果有一次需要 10001 个元素，程序就会出错。这里就可以使用动态数组来使程序适应不同的情形。

104. 错误代码：

```
int main ( )  
{  
double glad [2000];  
int i;  
.....  
for ( i = 0; i<1000;i++)  
    gobble ( glad,2000);  
.....  
}  
void gobble (double ar[],int n)  
{  
double * temp = (double *) malloc (n*sizeof(double));  
.....  
/* free(temp); // 忘记使用 free()释放内存 */  
}
```

分析：第一次调用 gobble () 时，它创建了指针 temp，并用 malloc () 为之分配 16000 字节的内存。由于没有使用 free () 释放内存，因此当函数终止时，指针 temp 作为一个自动变量消失了，但它所指向的 16000 个字节的内存仍旧存在。我们无法访问这些内存，因为地址不见了。当第二次调用 gobble () 时，它又创建指针 temp，再次分配 malloc () 为之分配 16000 字节的内存。由于第一个 16000 字节的块已不可用，因此 malloc () 不得不再找一个 16000 字节的块。当函数终止时，这个内存的块也无法访问，不可再利用。这次循环执行了 1000 次后，就已经有 1600 万字节的内存从内存池中移走。

事实上，在到达这一步前，程序很可能已经内存溢出了。这类问题被称为内存泄漏（memory leak），可以通过在函数末尾处调用 free（）防止该问题出现。

105. calloc 函数原型：extern void *calloc(int num_elems, int elem_size);

用法：#include <alloc.h>

功能：为具有 num_elems 个长度为 elem_size 元素的数组分配内存，并将块中的全部位都置为 0。

说明：如果分配成功则返回指向被分配内存的指针，否则返回空指针 NULL。

当内存不再使用时，应使用 free() 函数将内存块释放。

举例：

```
// calloc.c
#include <syslib.h>
#include <alloc.h>

main()
{
    char *p;

    clrscr();           // clear screen
    p=(char *)calloc(100,sizeof(char));
    if(p)
        printf("Memory Allocated at: %x",p);
    else
        printf("Not Enough Memory!\n");

    free(p);

    getchar();
    return 0;
}
```

106. C99 授予类型限定词一个新属性：它们现在是幂等（idempotent），这意味着可以在一个声明中不止一次地使用同一限定词，多余的将被忽略掉：

```
const const const int n = 6; //相当于 const int n = 6;
```

107. float * const pt; // pt 是一个常量指针

它必须总是指向同一个地址，但所指向的值可以改变。而下面的声明：

```
const float * const ptr;
```

意味着 ptr 必须总是指向同一个位置，并且它所指位置存储的值也不能改变。

```
float const * pfc; //等同于：const float * pt;
```

正如注释所表示的那样，把 const 放在类型名的后边和*的前边，意味着指针不能够用来改变它所指向的值。**一个位于*左边任意位置的 const 使得数据成为常量，而一个位于*右边的 const 使得指针自身成为常量。**

108. 限定词 volatile 告诉编译器该变量除了可被程序改变以外，还可被其他代理改变。典型地，它被用于硬件地址和其他并行程序共享的数据。例如，一个地址中可能保存着当前的时钟时间。不管程序做什么，该地址的值都会随着时间而改变。另一种情况是一个地址被用来接收来自其他计算机的信息。

比如：

```
volatile int loc1; // loc1 是一个易变的位置
```

```
volatile int * ploc; // ploc 指向一个易变的位置
```

109. 理解 volatile 关键词：

函数：

```
void func (void)
{
    unsigned char xdata xdata_junk;
    unsigned char xdata *p = &xdata_junk;
    unsigned char t1, t2;

    t1 = *p;
    t2 = *p;
}
```

编译的汇编为：

```
0000 7E00      R      MOV     R6,#HIGH xdata_junk
0002 7F00      R      MOV     R7,#LOW xdata_junk
;---- Variable 'p' assigned to Register 'R6/R7' ----

0004 8F82          MOV     DPL,R7
0006 8E83          MOV     DPH,R6

;!!!!!!!!!!!!!!!!!!!!!! 注意
0008 E0          MOVX    A,@DPTR
0009 F500      R      MOV     t1,A

000B F500      R      MOV     t2,A
;!!!!!!!!!!!!!!!!!!!!!!
000D 22          RET
```

将函数变为：

```
void func (void)
{
    volatile unsigned char xdata xdata_junk;
    volatile unsigned char xdata *p = &xdata_junk;
    unsigned char t1, t2;

    t1 = *p;
    t2 = *p;
}
```

编译的汇编为：

```
0000 7E00      R      MOV     R6,#HIGH xdata_junk
0002 7F00      R      MOV     R7,#LOW xdata_junk
;---- Variable 'p' assigned to Register 'R6/R7' ----

0004 8F82          MOV     DPL,R7
0006 8E83          MOV     DPH,R6

;!!!!!!!!!!!!!!
0008 E0          MOVX    A,@DPTR
0009 F500      R      MOV     t1,A      a 处

000B E0          MOVX    A,@DPTR
```

```
000C F500      R      MOV      t2,A  
;!!!!!!!!!!!!!!  
  
000E 22          RET
```

比较结果可以看出来，未用 volatile 关键字时，只从*p 所指的地址读一次

如在 a 处*p 的内容有变化，则 r2 得到的则不是真正*p 的内容。

110. 关键字 restrict 通过允许编译器优化某几咱代码增强了计算支持。它只可用于指针，并表明指针是访问一个数据对象的唯一且初始的方式。比如：

```
void * memcpy ( void * restrict s1,const void * restrict s2,size_t n);  
void * memmove ( void *s1,const void * s2,size_t n);
```

每一个函数都从位置 s2 把 n 个字节复制到位置 s1。函数 memcpy() 要求两个位置之间不重叠，但 memmove() 没有这个要求。把 s1 和 s2 声明为 restrict 意味着每个指针都是相应数据的唯一访问方式，因此它们不能访问同一数据块。这满足了不能有重叠的要求。函数 memmove() 允许重叠，它不得不在复制数据时更小心，以防在使用数据前就覆盖了数据。

- 111.

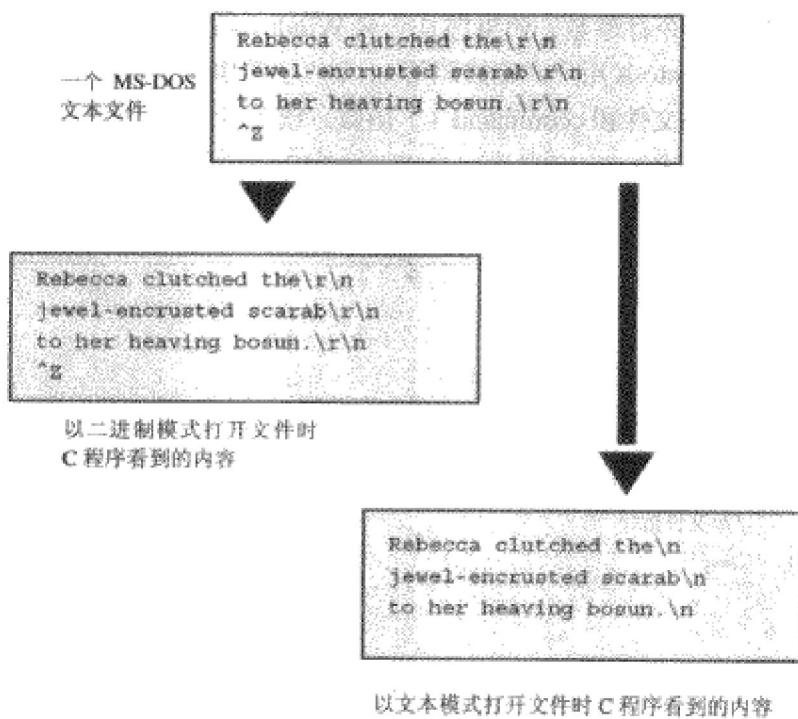


图 13.1 二进制视图和文本视图

- 112.

表 13.1

fopen() 函数的模式字符串

模式字符串	意 义
"r"	打开一个文本文件，可以读取文件
"w"	打开一个文本文件，可以写入文件，先将文件的长度截为零。如果该文件不存在则先创建之
"a"	打开一个文本文件，可以写入文件，向已有文件的尾部追加内容，如果该文件不存在则先创建之
"r+"	打开一个文本文件，可以进行更新，也即可以读取和写入文件
"w+"	打开一个文本文件，可以进行更新（读取和写入），如果该文件存在则首先将其长度截为零；如果不存在则先创建之
"a+"	打开一个文本文件，可以进行更新（读取和写入），向已有文件的尾部追加内容，如果该文件不存在则先创建之；可以读取整个文件，但写入时只能追加内容
"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"	与前面的模式相似，只是使用二进制模式而非文本模式打开文件

113. 如果使用任何一种“w”模式打开一个已有的文件，文件内容将被删除，以便程序以一个空文件开始操作。

114.

stdio.h 文件把 3 个文件指针与 3 个 C 程序自动打开的标准文件进行了关联，如表 13.2 所示。

表 13.2

标准文件及与其相关联的文件指针

标 准 文 件	文 件 指 针	一 般 使 用 的 设 备
标准输入	stdin	键盘
标准输出	stdout	显示器
标准错误	stderr	显示器

115. 简单的文件压缩程序：

```
// reducto.c -- reduces your files by two-thirds!
#include <stdio.h>
#include <stdlib.h>    // for exit()
#include <string.h>    // for strcpy(), strcat()
#define LEN 40

int main(int argc, char *argv[])
{
    FILE *in, *out;    // declare two FILE pointers
    int ch;
    char name[LEN];    // storage for output filename
    int count = 0;

    // check for command-line arguments
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(1);
    }
    // set up input
    if ((in = fopen(argv[1], "r")) == NULL)
    {
        fprintf(stderr, "I couldn't open the file \"%s\"\n",
                argv[1]);
    }
```

```
    exit(2);
}

// set up output
strncpy(name,argv[1], LEN - 5); // copy filename
name[LEN - 5] = '\0';
strcat(name, ".red");           // append .red
if ((out = fopen(name, "w")) == NULL)
{
    // open file for writing
    fprintf(stderr,"Can't create output file.\n");
    exit(3);
}
// copy data
while ((ch = getc(in)) != EOF)
{
    if (count++ % 3 == 0)
        putc(ch, out); // print every 3rd char
}
// clean up
if (fclose(in) != 0 || fclose(out) != 0)
    fprintf(stderr,"Error in closing files\n");

return 0;
}
```

116. rewind () 命令使程序中跳至文件开始处，它是以一个文件指针作为参数。

117. fgets () 函数读取到它所遇到的第一个换行字符的后面，或者读取比字符串的最大长度少一个的字符，或者读取到文件结尾。然后 fgets 函数向末尾添加一个空字符以构成一个字符串。所以字符串的最大长度代表字符的最大数目再加上一个空字符。如果 fgets 函数在达到字符最大数目之前读完了一整行，它将在字符串的空字符之前添加一个换行符以标识一行结束。这就是 fgets 函数与 gets 函数的不同之处，后者读取换行符后将其丢弃。与 gets () 类似，fgets () 遇到 EOF 的时候会返回 NULL 值，可以据此检查文件结尾。

118. 与 puts () 函数不同，fputs () 函数打印的时候并不添加一个换行符。由于 fgets () 函数保留了换行符，而 fputs () 函数不会添加换行符，所以它们配合得相当好。

119. fseek () 函数允许您像对待数组那样对待一个文件，在 fopen () 打开的文件中直接移动到任意字节处。ftell () 函数以一个 long 类型值返回一个文件的当前位置，它是一个 long 类型，通过返回距文件开始处的字节数目来确定文件位置，文件的第一个字节到文件开始处的距离是字节 0，依次类推。在 ANSI C 下，ftell () 这种定义适用于以二进制模式打开的文件，但是对于以文本模式打开的文件来讲，不一定是这样。代码：

```
#include <stdio.h>
#include "stdlib.h"
#define CNTL_Z '\032'
#define SLEN 20

int main(void)
{
    char ch;
    FILE *fp;
    char name[SLEN];
    long last,count;
```

```
puts("please enter a file name:");
gets(name);
if((fp=fopen(name,"rb"))==NULL)      // 以二进制模式打开文件，方便使用 ftell( ) 函数
{
    puts("open file failed!");
    exit(1);
}
fseek(fp,0L,SEEK_END);
last = ftell(fp);
for (count=1;count<=last;count++)
{
    fseek(fp,-count,SEEK_END);
    ch=getc(fp);
    if (ch!=CNTL_Z || ch=='\r')
        putc(ch,stdout);
}
printf("\n");
fclose(fp);
return 0;
}
```

运行结果：

```
please enter a file name:
word.txt
ksksuir
```

120. MS-DOS 用\r\n 组合表示文本文件中的换行符。以文本模式打开文件的 C 程序将\r\n 当成\n。但是如果使用二进制模式打开相同的文件，程序将看到这两个字符。对于 MS-DOS，ftell() 返回一个将\r\n 看成一个字节的计数值。

121. int fgetpos (FILE * restrict stream , fpos_t * restrict pos);

取得当前文件的句柄。被调用时，该函数在 pos 所指的位置放置一个 fpos_t 值（其中 fpos_t 代表 file position type，文件定位类型，用于代表位置），这个值描述了文件中的一个位置。如果成功，函数返回 0，否则返回一个非零值。

int fsetpos (FILE * stream , const fpos_t * pos);

定位流上的文件指针。被调用时，该函数使用 pos 指向的位置上的那个 fpos_t 值设定文件指针指向该值所指示的位置。如果成功，函数返回 0；否则返回一个非零值。这个 fpos_t 值应是通过调用 fgetpos() 函数获取的。

示例代码：

```
#include <stdlib.h>
#include <stdio.h>
void showpos(FILE *stream);
int main(void)
{
    FILE *stream;
    fpos_t filepos;
    /* open a file for update */
    stream = fopen("DUMMY.FIL", "w+");
    /* save the file pointer position */
    fgetpos(stream, &filepos);
```

```
/* write some data to the file */
fprintf(stream, "This is a test");
/* show the current file position */
showpos(stream);
/* set a new file position, display it */
if (fsetpos(stream, &filepos) == 0)
    showpos(stream);
else
{
    fprintf(stderr, "Error setting file \
pointer.\n");
    exit(1);
}
/* close the file */
fclose(stream);
return 0;
}

void showpos(FILE *stream)
{
    fpos_t pos;
    /* display the current file pointer
position of a stream */
    fgetpos(stream, &pos);
    printf("File position: %ld\n", pos);
}
```

运行结果：

File position: 14

File position: 0

122. int ungetc (int c , FILE * fp)函数将 c 指定的字符放回输入流中。

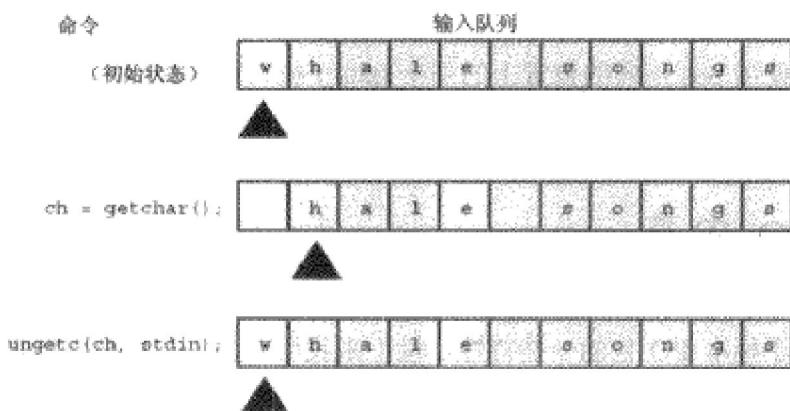


图 13.2 ungetc () 函数

123. fflush () 函数用于清除文件缓冲区，文件以写方式打开时将缓冲区内容写入文件。函数原型：

int fflush (FILE *fp) // 如果 fp 为空指针时，将刷新掉所有的输出缓冲。

124. setvbuf () 函数建立一个供标准 I/O 函数使用的替换缓冲区。打开文件以后，在没有对流进行任何操作以前，可以调用这个函数。setvbuf 可减少磁盘 IO 的次数，从而获得更好的效率。函数原型：

int setvbuf (FILE * restrict fp, char * restrict buf, int mode, size_t size);

其中 mode :

_IOFBF(满缓冲) : 当缓冲区为空时 , 从流读入数据。或者当缓冲区满时 , 向流写入数据。

_IOLBF(行缓冲) : 每次从流中读入一行数据或向流中写入一行数据。

_IONBF(无缓冲) : 直接从流中读入数据或直接向流中写入数据 , 而没有缓冲区。

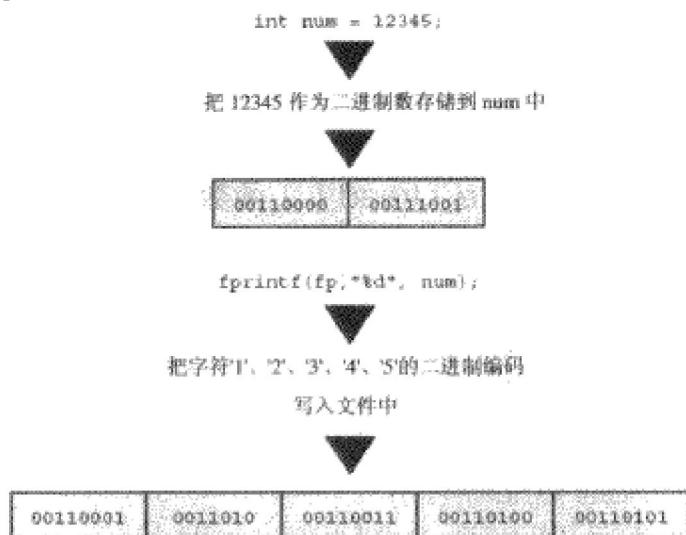
默认情况下 , 终端输出为行缓冲 , 标准错误输出为不缓冲 , 文件 I/O 操作为全缓冲 .

代码 :

```
#include <stdio.h>
int main ()
{
    FILE *pFile;
    pFile=fopen ("myfile.txt","w");
    setvbuf ( pFile , NULL , _IOFBF , 1024 );
    // File operations here
    fclose (pFile);
    return 0;
}
```

在这个例子中 , 每次写 1024 字节 , 所以只有缓冲区满了 (有 1024 字节) , 才往文件写入数据 , 可以减少 IO 次数 .

125. fprintf () 和 fwrite () :



`fwrite(&num, sizeof (int), 1, fp);`

把值 12345 的二进制编码写入文件中

00110000 00111001

(本图假设整数的大小为 16 位)

图 13.3 二进制输出和文本输出

126. int feof (FILE * fp) 和 int ferror (FILE *fp)

当标准输入函数返回 EOF 时 , 通常表示已经到达了文件结尾。可是 , 这也有可能表示发生了读取错误。

使用 feof () 和 ferror () 函数可以区分这两种可能性。如果最近一次输入调用检测到文件结尾。feof 函数返回一个非零值 , 否则返回零值。如果发生读写错误 , ferror 函数返回一个非零值 , 否则返回零值。

127. 把多个文件的内容追加到一个文件中 :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 1024
#define SLEN 81
void append(FILE *source, FILE *dest);

int main(void)
{
    FILE *fa, *fs;      // fa for append file, fs for source file
    int files = 0;      // number of files appended
    char file_app[SLEN]; // name of append file
    char file_src[SLEN]; // name of source file
    puts("Enter name of destination file:");
    gets(file_app);
    if ((fa = fopen(file_app, "a")) == NULL)
    {
        fprintf(stderr, "Can't open %s\n", file_app);
        exit(2);
    }
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Can't create output buffer\n", stderr);
        exit(3);
    }
    puts("Enter name of first source file (empty line to quit):");
    while (gets(file_src) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Can't append file to itself\n", stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Can't open %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Can't create input buffer\n", stderr);
                continue;
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr, "Error in reading file %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr, "Error in writing file %s.\n",
                        file_app);
            fclose(fs);
            files++;
        }
    }
}
```

```
    printf("File %s appended.\n", file_src);
    puts("Next file (empty line to quit):");
}
}

printf("Done. %d files appended.\n", files);
fclose(fa);

return 0;
}

void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // allocate once

    while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
        fwrite(temp, sizeof (char), bytes, dest);
}
```

128. 随机存取，二进制 I/O :

```
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
    double numbers[ARSIZE];
    double value;
    const char * file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile;

    // create a set of double values
    for(i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1);

    // attempt to open file
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Could not open %s for output.\n", file);
        exit(1);
    }

    // write array in binary format to file
    fwrite(numbers, sizeof (double), ARSIZE, iofile);
    fclose(iofile);

    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
```

```
"Could not open %s for random access.\n", file);
exit(1);
}
// read selected items from file
printf("Enter an index in the range 0-%d.\n", ARSIZE - 1);
scanf("%d", &i);
while (i >= 0 && i < ARSIZE)
{
    pos = (long) i * sizeof(double); // calculate offset
    fseek(iofile, pos, SEEK_SET);    // go there
    fread(&value, sizeof (double), 1, iofile);
    printf("The value there is %f.\n", value);
    printf("Next index (out of range to quit):\n");
    scanf("%d", &i);
}
// finish up
fclose(iofile);
puts("Bye!");

return 0;
}
```

测试结果：

```
Enter an index in the range 0-999.
33
The value there is 3300.029412.
Next index (out of range to quit):
0
The value there is 1.000000.
Next index (out of range to quit):
-1
Bye!
```

129. 结构指针：struct guy * him;

这个声明不是建立一个新的结构，而是意味着指针 him 现在可以指向任何现有的 guy 类型的结构，例如，如果 barney 是一个 guy 类型的结构，那么可以这样做：

```
him = &barney;
```

和数组不同，一个结构的名字不是该结构的地址，必须使用&运算符。

当 him = &fellow[0]，时，fellow[0].income == (*him).income // 必须要有圆括号，因为‘.’运算符比‘*’的优先级更高。总之，如果 him 是指向名为 barney 的 guy 类型结构的指针，则下列表达式是等价的：

```
barney.income == ( * him ).income == him->income
```

130. 通常，程序员为了追求效率而使用结构指针作为函数参数（只需传递单个地址，执行速度快），当需要保存数据、防止意外改变数据时对指针使用 const 限定词；而传递结构值作为函数参数是处理小型结构最常用的方法。

131. 在结构中使用字符数组还是字符指针：

```
struct names {
    char first [20];
    char last [20];
```

```
};

struct pnames {
    char * first;
    char * last;
};

struct names accountant;
struct pnames attorney;

puts ("Enter the last name of your accountant :");
scanf ("%s",accountant.last);
puts("Enter the last name of your attorney:");
scanf ("%s",attorney.last); // 这里存在潜在的危险
```

对会计师，他的名字存储在 accountant 变量的最后一个成员中，这个结构有一个用来存放字符串的数据组。对律师，scanf() 把字符串放在 attorney.last 给出的地址中。因为这是个未经初始化的变量，所以该地址可能是任何值，程序就可以把名字放在任何地址（可使用 malloc 分配内存来存储字符串以解决此问题）。如果幸运的话，程序至少有些时候可以正常运行。否则这个操作可以使程序彻底停止。实际上，如果程序运行，那是很不幸的，因为程序中会含有您未觉察到的危险的编程错误。

因此，如果需要一个结构来存储字符串，请使用字符数组成员。存储字符指针有它的用处，但也有被严重误用的可能。

132. 利用伸缩型数组成员可以声明最后一个成员是一个具有特殊属性的数据的结构，该数组成员的特殊属性之一是它不存在，至少不立即存在。第二个特殊属性是你可以编写适当的代码使用这个伸缩型数组成员，就像它确实存在并且拥有你需要的任何数目的元素一样。首先看看声明一个伸缩型数组成员的规则：

- 伸缩型数组成员必须是最后一个数组成员。
- 结构中必须至少有一个其他成员。
- 伸缩型数组就像普通数组一样被声明，除了它的方括号内是空的。

例子：

```
struct flex
{
    int count;
    double average;
    double scores[]; // 伸缩型数组成员
};
```

这里我们不能使用 scores 做任何事情，因为没有为它分配任何内存空间。实际上，C99 的意图并不是让你声明 struct flex 类型的变量；而是希望你声明一个指向 struct flex 类型的指针，然后使用 malloc() 来分配足够的空间，以存放 struct flex 结构的常规内容和伸缩型数组成员需要的任何额外空间。例如，假设要用 scores 表示含有 5 个 double 型数值的数组，那么就要这样做：

```
struct flex * pf; // 声明指针
// 请求一个结构和一个数组的空间
pf = malloc (sizeof(struct flex)+5*sizeof(double));
// 使用指针 pf 来访问这些成员
pf->count = 5; // 设置 count 成员的值
pf->scores[2] = 18.5; // 访问数组成员的一个元素
```

133. 联合是一个能在同一个存储空间里（但不同时）存储不同类型数据的数据类型。一个典型的应用是一种表，设计它是用来以某种既没有规律、事先也未知的顺序保存混合类型数据。编译器分配足够的空间以保存所描述的可能性的最大需要，比如联合中含有一个 int，一个 double 以及一个 char 型数值，那么列出的最大可能性是 double 型数据，即 8 个字节。由于联合只存储一个值，所以初始化的规则与结构的初始化不同。具体地，有 3 种选择：可以把一个联合初始化为同样类型的另一个联合；可以初

始化联合的第一个元素；或者，按照 C99 标准，可以使用一个指定初始化项目。

```
union hold {  
    int digit;  
    double bigfl;  
    char letter;  
};  
  
union hold valA;  
valA.letter = 'R';  
  
union hold valB = valA; // 把一个联合初始化为另一个联合  
  
union hold valC = {88}; // 初始化联合的 digit 成员  
  
union hold valD = {.bigfl = 118.2}; // 指定初始化项目
```

134. 可以使用枚举类型声明代表整数常量的符号名称。实际上，enum 常量是 int 类型的，因此在使用 int 类型的任何地方都可以使用它。枚举类型的是提高程序的可读性。例如，可以使用如下的声明：

```
enum spectrum ( red , orange , yellow , green, blue, violet );  
  
enum spectrum color;
```

第一个声明设置 spectrum 为标记名，从而允许你把 enum spectrum 作为一个类型名使用。第二个声明使得 color 成为该类型的一个变量。花括号中的标识符枚举了 spectrum 变量可能有的值。因此，color 可能为 red、orange、yellow。可以使用如下语句：

```
int c;  
color = blue;  
if ( color == yellow )  
    ..... ;  
for ( color = red; color <= violet; color++ )  
    ..... ;
```

那么 blue 和 red 到底为何物呢？其实它们是 int 类型的常量。看以下代码：

```
printf ( "red = %d, orange = %d\n", red, orange);
```

输出：

```
red = 0, orange = 1
```

默认情况下，枚举列表中的常量被指定为整数值 0、1、2 等等，当然你也可选择常量具有的整数值：

```
enum levels { low =100, medium = 500, high =2000 };
```

而如果只对一个常量赋值，而没有对后面的常量赋值，那么这些后面的常量会被赋予后续的值，例如：

```
enum feline {cat, lynx = 10, puma, tiger};
```

那么，cat 的值默认为 0, lynx、puma 和 tiger 的值分别为 10、11 和 12。

135. `typedef` 与 `#define` 的不同：

与 `#define` 不同，`typedef` 给出的符号名称仅限于对类型，而不是对值。

`typedef` 的解释由编译器，而不是预处理器执行。

虽然它的范围有限，但在其受限范围内，`typedef` 比 `#define` 更灵活。

如果使用 `typedef`：

```
typedef char * STRING;  
  
STRING name, sign; // 相当于 char * name, * sign;
```

如果使用 `#define`：

```
#define STRING char *  
  
STRING name, sign; // 相当于 char * name, sign; , 这时只有 name 是指针
```

也可使用 `typedef` 定义结构：

```
typedef struct complex {  
    float real;
```

```
float imag;  
} COMPLEX;
```

这时你就可以使用类型 COMPLEX 来代替 struct complex 来表示复数。

当使用 `typedef` 时，它并不创建新的类型，而只是创建了便于使用的标签。

136. 奇特的声明：

```
int board[8][8]          // int 数组的数组  
int **ptr;              // 指向 int 的指针的指针  
int * risks[10];         // 具有 10 个元素的数组，每个元素是一个指向 int 的指针  
int ( * rusks ) [10];    // 一个指针，指向具有 10 个元素的 int 数组  
int * oof [3][4];        // 一个 3 x 4 的数组，每个元素是一个指向 int 的指针  
int ( * uuf ) [3][4];    // 一个指针，指向 3 x 4 的 int 数组  
int ( * uof [3] ) [4];    // 一个具有 3 个元素的数组，每个元素是一个  
                          // 指向具有 4 个元素的 int 数组的指针  
char * fump ();          // 返回指向 char 的指针的函数  
char ( * frmp ) ();      // 指向返回类型为 char 的函数的指针  
char ( * flump [3] ) (); // 由 3 个指针组成的数组，每个指针指向返回类型为 char 的函数  
  
typedef int aar5[5];  
typedef aar5 * p_arr5;  
typedef p_arr5 arrp10[10];  
arr5 togs;   // togs 是具有 5 个元素的 int 数组  
p_arr5 ps;   // ps 是一个指针，指向具有 5 个元素的 int 数组  
arrp10 ap;   // ap 是具有 10 个元素的指针数组，每个指针指向具有 5 个元素的 int 数组
```

137. 如果想要声明一个指向一个特定类型函数的指针，可以声明一个特定类型的函数，然后用一个 `(* pf)` 的形式的表达式来替代函数名，以创建一个函数指针声明。比如：

```
void ToUpper ( char * ) // 把字符串转换为大写
```

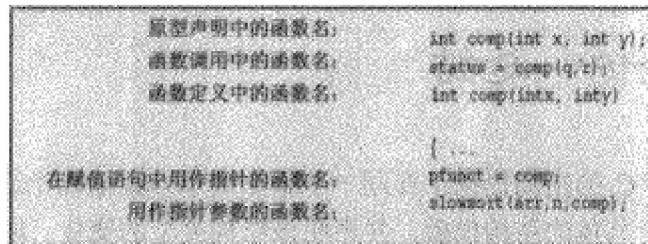
要声明指向这种类型的函数指针 `pf`，可以：

```
void ( * pf )(char * ) // pf 是一个指向函数的指针
```

若省略前一个圆括号，则：

```
void *pf( char * ) // pf 是返回一个指针的函数
```

138.



139. 使用函数指针：

```
#include "stdio.h"  
#include "string.h"  
#include "ctype.h"  
  
char showmenu(void);  
void eatline ( void );  
void show ( void (* fp)(char *),char * str );  
void ToUpper ( char * );
```

```
void ToLower (char *);  
void Transpose (char *);  
void Dummy (char *);  
  
int main (void)  
{  
    char line [81];  
    char copy[81];  
    char choice;  
    void (*pfun)(char *);  
  
    puts("enter a string (empty line to quit):");  
    while(gets(line) != NULL && line[0]!='\0')  
    {  
        while((choice = showmenu())!='n')  
        {  
            switch(choice)  
            {  
                case 'u': pfun = ToUpper;break;  
                case 'l': pfun = ToLower;break;  
                case 't': pfun = Transpose;break;  
                case 'o': pfun = Dummy;break;  
            }  
            strcpy(copy,line);  
            show(pfun,copy);  
        }  
        puts("enter a string (empty line to quit):");  
    }  
    puts("bye!");  
    return 0;  
}  
  
char showmenu(void)  
{  
    char ans;  
    puts("enter menu choice:");  
    puts("u) uppercase l) lowercase");  
    puts("t) transposed case o) original case");  
    puts("n) next string");  
    ans = getchar();  
    ans = tolower(ans);  
    eatline();  
    while(strchr("ulton",ans)==NULL)  
    {  
        puts("please enter a u,l,t,o,or n:");  
        ans = getchar();  
        ans = tolower(ans);  
        eatline();  
    }  
}
```

```
}

return ans;
}

void eatline(void)
{
    while(getchar() != '\n')
        continue;
}

void ToUpper(char *str )
{
    while(*str)
    {
        *str = toupper(*str);
        str++;
    }
}

void ToLower(char *str)
{
    while(*str)
    {
        *str = tolower(*str);
        str++;
    }
}

void Transpose(char *str)
{
    while(*str)
    {
        if(islower(*str))
            *str = toupper(*str);
        else if (isupper(*str))
            *str = tolower(*str);
        str++;
    }
}

void Dummy(char *str)
{

}

void show (void (*fp) (char *),char * str)
{
    (*fp)(str);
}
```

```
    puts (str);  
}
```

运行结果：

```
enter a string (empty line to quit):  
does c make you feel loopy?  
enter menu choice:  
u) uppercase l) lowercase  
t) transposed case o) original case  
n) next string  
t  
DOES C MAKE YOU FEEL LOOPY?  
enter menu choice:  
u) uppercase l) lowercase  
t) transposed case o) original case  
n) next string  
l  
does c make you feel loopy?  
enter menu choice:  
u) uppercase l) lowercase  
t) transposed case o) original case  
n) next string  
n  
enter a string (empty line to quit):  
  
bye!
```

140. 二进制补码：按位取反加一；

二进制反码：按位取反。

二进制计数法只能精确地表示多个 $1/2$ 的幂的和，因此 $3/4$ 和 $7/8$ 可以精确地表示为二进制小数，但是 $1/3$ 和 $2/5$ 却不能。

每个八进制位对应于 3 个二进制位，如：八进制数 0377 相当于二进制数 11111111，其中 $011=3,111=7$ 。

每个十六进制位对应于一个 4 位的二进制数，因此两个十六进制位恰好对应于一个 8 位字节。

141. 取反 (\sim)：变 1 为 0，变 0 为 1，比如： $\sim(10011010) = 01100101$

与运算 AND (&)：遇 0 则 0，比如： $(10010011) \& (00111101) = 00010001$

或运算 OR (|)：遇 1 则 1，比如： $(10010011) | (00111101) = 10111111$

异或运算 (^)：相同则为 0，遇 1 则反，比如： $(10010011) ^ (00111101) = 10101110$

142. 掩码：运用 AND 运算保留特定位的值，其它位清零。flags = flags & MASK

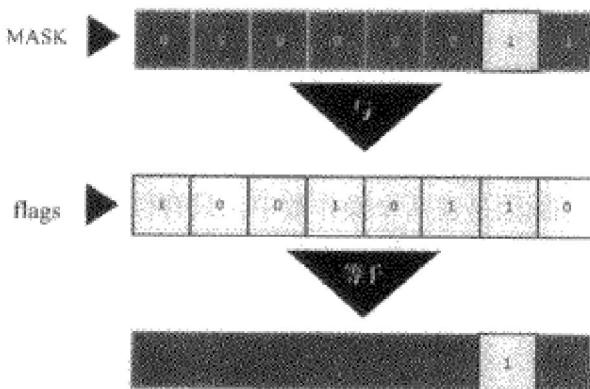


图 15.2 一个掩码

运用 OR 运算打开位：打开一个值中特定的位，同时保持其他位不变。 flags = flags | MASK

关闭位：不影响其它位，同时能够将特定的位关闭。 flags = flags & ~MASK 或者 flags &= ~MASK

转置位：转置一个位表示如果该位打开，则关闭该位；如果该位关闭，则打开该位。如果使用异或将一个值与掩码组合，那么该值中对应掩码位为 1 的位被转置，对应掩码位为 0 的位不改变。比如：

flag = flag ^ MASK;

查看一位的值：

```
if (( flag & MASK) == MASK)
```

```
    puts ("wow!");
```

143. **左移<<**：将其左侧操作数的值的每 位向左移动，移动的位数由其右侧操作数指定。空出的位用 0 填充，并且丢弃移出左侧操作数末端的位，相当于乘以 2 的次幂。比如：(10001010) << 2，结果为 (00101000)。

右移>>：将其左侧操作数的值的每位向右移动，移动的位数由其右侧操作数指定，丢弃移出左侧操作数右端的位，如果数值非负，则相当于除以 2 的次幂。对于 unsigned 类型，使用 0 填充左端空出的位。对于有符号类型，结果依赖于机器，空出的位可能用 0 填充，或者使用符号（最左端的）位的副本填充：

(10001010) >> 2，结果可能为(00100010)，也可能为(11100010)

对于无符号值：

(10001010) >> 2，结果为(00100010)

144. 使用位运算显示二进制数：

```
#include "stdio.h"  
  
char * itobs(int n,char * ps);  
void show_bstr(const char *);  
int invert_end(int num,int bits);  
  
int main(void)  
{  
    int num;  
    char ps[8*sizeof(int)+1];  
  
    printf("please a int:\n");  
    while(scanf("%d",&num)==1)  
    {  
        itobs(num,ps);  
        show_bstr(ps);  
        putchar('\n');  
        num = invert_end(num,4);  
        printf("inverting the last 4 bits gives\n");  
        show_bstr(itobs(num,ps));  
        putchar('\n');  
    }  
    puts("bye!");  
  
    return 0;  
}  
  
char * itobs(int n,char * ps)  
{  
    int i;
```

```
static int size = 8 * sizeof(int);

for(i = size -1;i >= 0;i--,n >>= 1)
    ps[i]=(n&1) + '0';      //由于是字符数组，因此加'0'进行转换
ps[size]='\0';

return ps;
}

void show_bstr(const char *str)
{
int i = 0;
while(str[i])
{
    putchar(str[i]);
    if(++i%4 == 0 && str[i]) //4位一组显示二进制字符串
        putchar(' ');
}
}

int invert_end(int num,int bits) // 反转一个值中的最后 n 位，参数为 n，和要反转的值
{
int mask = 0;
int bitval = 1;
// 依次对 mask 后面 bits 位置 1，然后再与需反转的值进行异或运算
while(bits-->0)
{
    mask |= bitval;
    bitval <<= 1;
}
return num ^ mask;
}
```

运行结果：

```
please a int:
4
0000 0000 0000 0000 0000 0000 0000 0100
inverting the last 4 bits gives
0000 0000 0000 0000 0000 0000 0000 1011
34
0000 0000 0000 0000 0000 0000 0010 0010
inverting the last 4 bits gives
0000 0000 0000 0000 0000 0000 0010 1101
q
bye!
```

145. 位字段是一个 signed int 或 unsigned int 中一组相邻的位 (C99 还允许 _Bool 类型位字段)，例如声明以下 4 个 1 位字段：

```
struct {
```

```
unsigned int autfd:1;  
unsigned int bldfc:1;  
unsigned int undln:1;  
unsigned int itals:1;  
} prnt;
```

该定义使 prnt 包含 4 个 1 位字段，你可以使用普通的结构成员运算符将值赋给单独的字段：

```
prnt.itals = 0;  
prnt.undln = 1;
```

你还可以创建多位字段：

```
struct {  
    unsigned int code1:2;  
    unsigned int code2:2;  
    unsigned int code3:8;  
} procd;
```

这段代码创建两个 2 位字段和一个 8 位字段，你可以使用以下进行赋值：

```
prcode.code1 = 0;  
prcode.code2 = 3;  
prcode.code3 = 102;
```

只须确保值没有超出字段的容量。

如果你声明的总位数超过一个 unsigned int 大小，那将会使用下一个 unsigned int 存储位置。不允许一个字段跨越两个 unsigned int 之间的边界。编译器自动地移位一个这样的字段定义，使字段按 unsigned int 边界对齐。发生这种情况时，会在第一个 unisgned int 中留下一个未命名的洞。你还可以使用未命名的字段宽度来“填充”未命名的洞，使用一个宽度为 0 的未命名的字段迫使下一个字段与下一个整数对齐：

```
struct {  
    unsigned int field1:1;  
    unsigned int :2;  
    unsigned int field2:1;  
    unsigned int :0;  
    unsigned int field3:1;  
} stuff;
```

stuff.field1 和 suff.field2 之间有一个 2 位的间隙，stuff.field3 存储在下一个 int 中。

146. 位运算和位字段：

```
#include <stdio.h>  
/*位字段常量 */  
/*是否透明和是否可见 */  
#define YES      1  
#define NO       0  
/*边框线的样式 */  
#define SOLID    0  
#define DOTTED   1  
#define DASHED   2  
/*三原色 */  
#define BLUE     4  
#define GREEN    2  
#define RED      1  
/*混合色 */
```

```
#define BLACK    0
#define YELLOW   (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN     (GREEN | BLUE)
#define WHITE    (RED | GREEN | BLUE)

/* 位运算中使用的常量 */
#define OPAQUE        0x1
#define FILL_BLUE     0x8
#define FILL_GREEN    0x4
#define FILL_RED      0x2
#define FILL_MASK     0xE
#define BORDER        0x100
#define BORDER_BLUE   0x800
#define BORDER_GREEN  0x400
#define BORDER_RED   0x200
#define BORDER_MASK   0xE00
#define B_SOLID       0
#define B_DOTTED      0x1000
#define B_DASHED     0x2000
#define STYLE_MASK   0x3000

const char * colors[8] = {"black", "red", "green", "yellow",
                         "blue", "magenta", "cyan", "white"};
struct box_props {

    unsigned int opaque          : 1;
    unsigned int fill_color      : 3;
    unsigned int                 : 4;
    unsigned int show_border    : 1;
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                 : 2;
};

union Views /*把数据看作结构或 unsigned short 常量 */
{
    struct box_props st_view;
    unsigned int ui_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(unsigned int n, char * ps); // 把 short 值以二进制字符串的形式显示

int main(void)
{
    /*创建 Views 对象，初始化结构 box view */
}
```

```
union Views box = {{YES, YELLOW, YES, GREEN, DASHED}};

char bin_str[8 * sizeof(unsigned int) + 1];

printf("Original box settings:\n");
show_settings(&box.st_view);

printf("\nBox settings using unsigned int view:\n");
show_settings1(box.ui_view);

printf("bits are %s\n",
itobs(box.ui_view,bin_str));

box.ui_view &= ~FILL_MASK; /*把代表填充色的位清 0 */
box.ui_view |= (FILL_BLUE | FILL_GREEN); /*重置填充色 */

box.ui_view ^= OPAQUE; /*转置指示是否透明的位 */
box.ui_view |= BORDER_RED; /*错误的方法 */
box.ui_view &= ~STYLE_MASK; /* 清除样式位 */
box.ui_view |= B_DOTTED; /* 把样式设置为点*/

printf("\nModified box settings:\n");
show_settings(&box.st_view);

printf("\nBox settings using unsigned int view:\n");
show_settings1(box.ui_view);

printf("bits are %s\n",
itobs(box.ui_view,bin_str));

return 0;
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
pb->opaque == YES? "opaque": "transparent");

    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
pb->show_border == YES? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf ("The border style is ");
    switch(pb->border_style)
    {
        case SOLID : printf("solid.\n"); break;
        case DOTTED : printf("dotted.\n"); break;
        case DASHED : printf("dashed.\n"); break;
        default      : printf("unknown type.\n");
    }
}

void show_settings1(unsigned short us)
{
    printf("box is %s.\n",
us & OPAQUE == OPAQUE? "opaque": "transparent");
```

```
printf("The fill color is %s.\n",
       colors[(us >> 1) & 07]);
printf("Border %s.\n",
       us & BORDER == BORDER? "shown" : "not shown");
printf ("The border style is ");
switch(us & STYLE_MASK)
{
    case B_SOLID   : printf("solid.\n"); break;
    case B_DOTTED : printf("dotted.\n"); break;
    case B_DASHED : printf("dashed.\n"); break;
    default        : printf("unknown type.\n");
}
printf("The border color is %s.\n",
       colors[(us >> 9) & 07]);

}

/* convert int to binary string */
char * itobs(unsigned int n, char * ps)
{
    int i;
    static int size = 8 * sizeof(unsigned int);

    for (i = size - 1; i >= 0; i--, n >= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}
```

运行结果：

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is opaque.
The fill color is yellow.
Border shown.
The border style is dashed.
The border color is green.
bits are 000000000000000010010100000111

Modified box settings:
Box is transparent.
```

The fill color is cyan.
Border shown.
The border color is yellow.
The border style is dotted.

Box settings using unsigned int view:
box is transparent.
The fill color is cyan.
Border not shown.
The border style is dotted.
The border color is yellow.
bits are 0000000000000000101100001100147.

147. 代码：

```
#define LIMIT 20
const int M = 50;
static int data1[LIMIT];      //合法
static int data2[LIM];        //无效
const int LIM2 = 2 * LIMIT; //合法
const int LIM3 = 2 * LIM;   //无效
```

148.

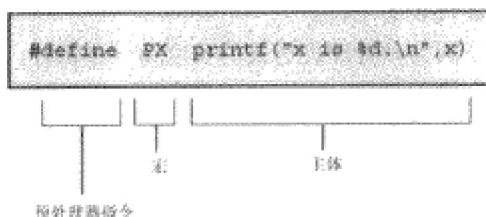


图 16.1 类对象宏定义的组成

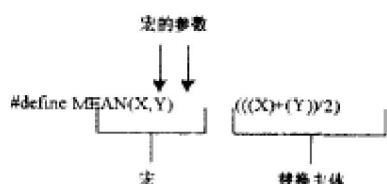


图 16.2 类函数宏定义的组成

149. 预处理器不进行运算，而只进行字符串替换，比如：

```
#include "stdio.h"
#define SQUARE(x) x*x
#define PR(X) printf("The result is %d.\n",X)

int main()
{
    int x=4;

    printf("SQUARE(x):\n");
    PR(SQUARE(x));
    printf("SQUARE(x+2):\n");
    PR(SQUARE(x+2));
```

```
printf("100/SQUARE(2):\n");
PR(100/SQUARE(2));
printf("SQUARE(++x):\n");
PR(SQUARE(++x));
printf("After incrementing, x is %x.\n",x);

return 0;
}
```

运行结果：

```
SQUARE(x):
The result is 16.

SQUARE(x+2):
The result is 14.

100/SQUARE(2):
The result is 100.

SQUARE(++x):
The result is 36.

After incrementing, x is 6.
```

分析：

SQUARE(x+2)会被替换为 $x+2*x+2 = 4+2*4+2 = 14$ ，并非(x+2)的平方值，此时我们可使用以下表达值来解决此问题：

```
#define SQUARE (x) (x) * (x)
```

现在 SQUARE(x+2)会被替换为 $(x+2) * (x+2)$ 。

100/SQUARE(2)会被替换为 $100/2*2=100$ ，我们可以使用以下表达值来解决此问题：

```
#define SQUARE (x) (x*x)
```

联合处理以上两种情况，需要这样定义：

```
#define SQUARE (x) ((x) * (x))
```

SQUARE(++x)被替换成 $++x * ++x = 5*6$ ，结果 x被加了两次，因此在宏的参数中应避免使用 $++x$ ，一般来说，在宏中不要使用增量或减量运算符。

150. 预处理器的粘合剂：##运算符

和#运算符一样，##运算符可以用于类函数宏的替换部分。例如：

```
#define XNAME(n) x ## n
```

下面的宏调用：

XNAME (4) 会展开形成下列形式：

x4

151. 可变宏：...和__VA_ARGS__

例如：

```
#define PR(...) printf(__VA_ARGS__)
```

调用该宏：

```
PR("Howdy");
PR("weight = %d, shipping = $%.2f\n",wt,sp);
```

第一次调用中，__VA_ARGS__展开为 1 个参数：

“ Howdy ”

第二次调用中，它展开为 3 个参数：

“ weight = %d, shipping = \$%.2f\n”,wt,sp);

因此展开后的代码为：

```
printf ("Howdy");
```

```
printf("weiht = %d, shipping = $%.2f\n",wt,sp);
```

其中的省略号只能代替最后的宏参数，下面的定义是错误的：

```
#define WORNG(X,...,Y) #X #__VA_ARGS__ #y
```

152. 宏与函数间的选择实际上是时间与空间的权衡。宏产生内联代码；也就是说，在程序中产生语句。如果使用宏 20 次，则会把 20 行代码插入程序中。如果使用函数 20 次，那么程序中只有一份函数语句的拷贝，因此节省了空间。另一方面，程序的控制必须转移到函数中并随后返回调用程序，因此这比内联代码花费的时间多。**宏不检查其中的变量类型**（这是因为宏处理字符型字符串，而不是实际值），程序员一般将宏用于简单函数。
153. 宏的名字中不能有空格，但是在替代字符串中可以使用空格。ANSI C 允许在参数列表中使用空格。
- 用圆括号括住每个参数，并括住宏的整体定义。
- 用大写字母表示宏函数名。
- 如果打算使用宏代替函数来加快程序的运行速度，那么首先应确定宏是否会引起重大差异。在程序中只使用一次的宏对程序运行时间可能不会产生明显的改善。在嵌套循环中使用宏更有助于加速运行。许多系统提供程序配置器以帮助程序员压缩最耗运行时间的程序部分。
- 154.

表 16.2 使用#include 指令的一些例子

#include <stdio.h>	搜索系统目录
#include "hot.H"	搜索当前工作目录
#include "/usr/biff/p.h"	搜索/usr/biff 目录

155. 为了在不同的工作环境下选择不同的代码类型，可以通过改变一些#define 宏的值，以实现不同系统间的代码移植。**#undef 指令取消前面的#define 定义**（即使前面未用#define 进行定义也是可以用#undef 取消的），#if、#ifdef、#ifndef、#else、#elif 和#endif 指令可用于选择什么情况下编译哪些代码。**#line 指令用于重置行和文件信息**，#error 指令用于给出错误信息，#pragma 指令用于向编译器发出指示。
156. 当某文件包含几个头文件，而且每个头文件都可能定义了相同的宏时，**用#ifndef 可以防止对该宏重复定义**，此时，第一个头文件中的定义变成有效定义，而其他头文件中的定义则被忽略。比如：

```
/*arrays.h*/  
#ifndef SIZE  
    define SIZE 100  
#endif
```

下面是另一个应用，假设把下面这行代码

```
#include "arrays.h"
```

放在一个文件头部，那么 SIZE 定义为 100，但如果把：

```
#define SIZE 10  
#include "arrays.h"
```

放在文件头部，那么 SIZE 定义为 10.

157. 多次包含同一文件的原因：许多包含文件自身包含了其他文件，因此可能显式地包含其他文件已经包含的文件，而头文件中的有些语句在一个文件中只能出现一次（如结构类型的声明）。

158. 通常编译器提供商使用以下方法来确保使用的标识符在其他任何地方都没有定义过：

用文件名做标识符，并在文件中使用大写字母、用下划线代替文件名中的句点字符、用下划线（可能使用两条下划线）作前缀和后缀。例如：

```
#ifndef _STDIO_H // 或者 STDIO_H_  
#define _STDIO_H  
#endif
```

159. 带有多次包含保护的修订版本：

```
/*names_st.h*/
#ifndef NAMES_H_
#define NAMES_H_

#define SLEN 32

struct names_st
{
    char first[SLEN];
    char last[SLEN];
};

typedef struct names_st names;

void get_names (names *);
void show_names(const names *);

#endif
```

以下文件调用上面的头文件：

```
#include <stdio.h>
#include "names.h"
#include "names.h" //不小心两次包含同一头文件

int main ()
{
    names winner = {"Less","Ismoor"};
    printf("The winner is %s %s.\n",winner.first,winner.last);
    return 0;
}
```

上面的代码编译会成功，但是如果去掉头文件中的#ifndef 保护后，程序则不能通过编译，在 VC6 下提示：error C2011: 'names_st' : 'struct' type redefinition

160. #if 后跟常量整数表达式，如果表达式为非零值，则表达式为真。

```
#if SYS == 1
#include "ibm.h"
#endif
```

可以使用#elif 指令扩展 if-else 序列，例如：

```
#include <stdio.h>
#define SYS 2

int main ()
{
#if SYS == 1
    printf("This is 1\n");
#elif SYS == 2
    printf("This is 2\n");
#elif SYS == 3
    printf("This is 3\n");
}
```

```
#endif  
return 0;  
}
```

运行结果：

```
This is 2
```

许多新的实现提供另一种方法来判断一个名字是否已经定义。无需使用：

```
#ifdef VAX
```

而是采用下面的形式：

```
#if defined (VAX)
```

它的优点在于它可以和#elif一起使用。

161.

表 16.3

预定义宏

宏	意 义
__DATE__	进行预处理的日期 ("Mmm dd yyyy"形式的字符串文字)
__FILE__	代表当前源代码文件名的字符串文字
__LINE__	代表当前源代码文件中的行号的整数常量
__STDC__	设置为 1 时，表示该实现遵循 C 标准
__STDC_HOSTED__	为本机环境设置为 1，否则设为 0
__STDC_VERSION__	为 C99 时设置为 199901L
__TIME__	源文件编译时间，格式为"hh: mm: ss"

代码示例：

```
#include <stdio.h>  
  
int main ()  
{  
    printf("the file is %s.\n", __FILE__);  
    printf("the date is %s.\n", __DATE__);  
    printf("the time is %s.\n", __TIME__);  
    printf("this is line %d.\n", __LINE__);  
  
    return 0;  
}
```

运行结果：

```
the file is D:\riusksk\TOOL\MSDev98\MyProjects\c primer plus\example\example.c.  
the date is May 29 2010.  
the time is 00:12:32.  
this is line 8.
```

162. C99 提供了_Pragma 预处理器运算符，可将字符串转换成常规的编译指示。例如：

```
_Pragma ( "nonstandardtreatmenttype on" )
```

等价于下面的指令：

```
#pragma nonstandardtreatmenttype on
```

由于该运算符没有使用#符号，所以可将它作为宏展开的一部分：

```
#define PRAGMA(x) _Pragma(#x)
```

```
#define LIMRG(x) PRAGMA (STDC CX_LIMITED_RANGE X)
```

```
LIMRG (ON)
```

163. 把函数变为内联函数是为了让编译器尽可能快速地调用该函数，它会让内联函数的函数体代替函数调用。因为内联函数没有预留给它的单独的代码块，所以无法获得内联函数的地址（实际上，可以获得地址，但这样会使编译器产生非内联函数）。另外，内联函数不会在调试器中显示。**内联函数应该比较短小**，对于很长的函数，调用函数的时间少于执行函数主体的时间，此时使用内联函数不会节省多少时间。

164.

表 16.4 ANSI C 标准数学函数原型描述

原 型	描 述
double acos (double x)	返回余弦值为 x 的角度值（0 到π弧度）
double asin (double x)	返回正弦值为 x 的角度值（-π/2 到π/2 弧度）
double atan (double x)	返回正切值为 x 的角度值（-π/2 到π/2 弧度）
double atan2 (double y, double x)	返回正切值为 y/x 的角度值（-π 到π弧度）
double cos (double x)	返回 x 的余弦值，x 单位为弧度
double sin (double x)	返回 x 的正弦值，x 单位为弧度
double tan (double x)	返回 x 的正切值，x 单位为弧度
double exp (double x)	返回 x 的指数函数的值（e 的 x 次方）
double log (double x)	返回 x 的自然对数值
double log10 (double x)	返回 x 的以 10 为底的对数值
double pow (double x, double y)	返回 x 的 y 次幂的值
double sqrt (double x)	返回 x 的平方根
double ceil (double x)	返回不小于 x 的最小整数值
double fabs (double x)	返回 x 的绝对值
double floor (double x)	返回不大于 x 的最大整数值

165. atexit()指定执行 exit()时调用的特定函数，它使用函数指针作为参数。Atexit () 把作为其参数的函数在调用 exit () 执行的函数列表中进行注册。ANSI 保证在这个列表中至少可放置 32 个函数。通过使用一个单独的 atexit () 调用把每个函数添加到列表中。最后，调用 exit () 函数时，按**先进后出（先执行最后添加的函数）**的顺序执行这些函数。

代码：

```
#include <stdio.h>
#include "stdlib.h"
void sign_off (void);
void too_bad (void);

int main(void)
{
    int n;

    atexit(sign_off);
    puts("enter an int:");
    if(scanf("%d",&n)!=1)
    {
        puts("that's no int!");
        atexit(too_bad);
        exit(EXIT_FAILURE);
    }
}
```

```
printf("%d is %s.\n",n,(n%2==0)?"even":"odd");
return 0;
}

void sign_off()
{
puts( "sign_off");
}

void too_bad()
{
puts("too_bad");
}
```

运行结果：

```
enter an int:
3
3 is odd.
sign_off
```

另一个运行实例：

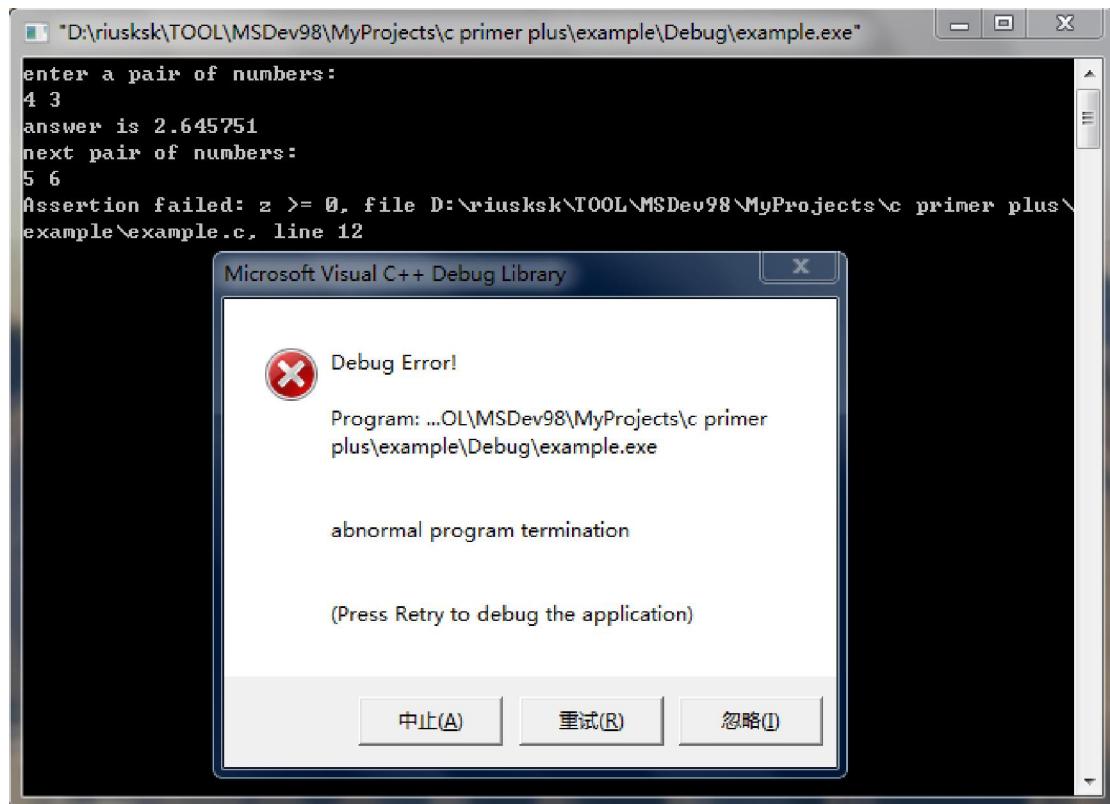
```
enter an int:
riusksk
that's no int!
too_bad
sign_off
```

166. 头文件 assert.h 支持的诊断库是设计用于辅助调试程序的小型库。它由宏 assert() 构成，该宏接受整数表达式作为参数。如果表达式值为假（非零），宏 assert() 向标准错误流(stderr)写一条错误消息并调用 abort() 函数以终止程序（在头文件 stdlib.h 中定义 abort() 函数的原型）。assert()宏的作用为：标识出程序中某个条件应为真的关键位置，并在条件为假时用 assert() 语句终止该程序。通常，assert() 的参数为关系或逻辑表达式。如果 assert() 终止程序，那么它首先会显示失败的判断、包含该判断的文件名和行号。示例代码：

```
#include <stdio.h>
#include "math.h"
#include "assert.h"
int main()
{
    double x,y,z;
    puts("enter a pair of numbers:");
    while(scanf("%lf%lf",&x,&y) == 2
        && (x!=0 || y!=0))
    {
        z = x*x - y*y;
        assert(z >= 0); //这里并不是声称 z>=0,而是声称 z>=0 这个条件没有得到满足 ,
                        //相当于 if(z<0) abort();
        printf("answer is %f\n",sqrt(z));
        puts("next pair of numbers:");
    }
}
```

```
return 0;  
}
```

运行结果：



如果在 assert.h 前定义：

```
#define NDEBUG
```

那么编译器将禁用文件中所有的 assert() 语句，比如：

```
enter a pair of numbers:  
5 6  
answer is -1.#IND00  
next pair of numbers:
```

167. string.h 库中的 memcpy() 和 memmove()：

函数原型：

```
void *memcpy(void * restrict s1,const void * restrict s2, size_t n);  
void *memmove(void *s1,const void *s2, size_t n);
```

两个函数均是从 s2 指向的位置复制 n 字节数据到 s1 指向的位置，且均返回 s1 的值。两者间的差别由关键字 restrict 造成，即 memcpy() 可以假定两个内存区域之间没有重叠。memmove() 函数则不作这个假定，因此，复制过程类似于首先将所有字节复制到一个临时缓冲区，然后再复制到最终目的地。如果两个区域存在重叠时使用 memcpy() 时，其行为是不可预知的，即可能正常工作，也可能失败。

示例代码：

```
#include <stdio.h>  
#include "string.h"  
#include "stdlib.h"  
#define SIZE 4  
void show_array (const int ar[],int n);  
  
int main()  
{
```

```
int value[SIZE]={1,2,3,4};  
int target[SIZE];  
double curious[SIZE/2] = { 1.0, 2.0};  
  
puts("memcpy() used:");  
puts("values (original data):");  
show_array(value,SIZE);  
memcpy(target,value,SIZE*sizeof(int));  
puts("target (copy of values):");  
show_array(target,SIZE);  
  
puts("using memmove() with overlapping ranges:");  
memmove(value+2,value,(SIZE/2)*sizeof(double)); // 可能会导致崩溃，实际中不推荐使用  
show_array(value,SIZE);  
  
puts("using memcpy ( ) to copy double to int:");  
memcpy(target,curious,(SIZE/2)*sizeof(double));  
puts("target -- 2 doubles into 4 int positions:");  
show_array(target,SIZE);  
  
return 0;  
}  
  
void show_array(const int ar[],int n)  
{  
    int i;  
  
    for(i=0;i<n;i++)  
        printf("%d ",ar[i]);  
    putchar('\n');  
}
```

运行结果：

```
memcpy() used:  
values (original data):  
1 2 3 4  
target (copy of values):  
1 2 3 4  
using memmove() with overlapping ranges:  
2 3 4 4  
using memcpy ( ) to copy double to int:  
target -- 2 doubles into 4 int positions:  
0 1072693248 0 1073741824
```

168. 使用可变个数的参数：

```
#include <stdio.h>  
#include <stdarg.h>  
double sum (int, ...);  
int main (void)
```

```
{  
    double s,t;  
    s = sum (3, 1.1, 2.5, 13.3);  
    t = sum(6, 1.1, 2.1, 13.1, 4.2, 5.1, 6.1);  
    printf("return value for sum (3, 1.1, 2.5, 13.3):%g\n",s);  
    printf("return value for sum (6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1):%g\n",t);  
    return 0;  
}  
  
double sum (int lim, ...)  
{  
    va_list ap; // 声明用于存放参数的变量  
    double tot = 0;  
    int i;  
    va_start (ap, lim); //把 ap 初始化为参数列表  
    for ( i=0; i<lim; i++)  
        tot += va_arg (ap, double); //访问参数列表中的每一个项目  
    va_end (ap); //清理工作  
    return tot;  
}
```

运行结果：

```
return value for sum (3, 1.1, 2.5, 13.3):16.9  
return value for sum (6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1):31.7
```

169.

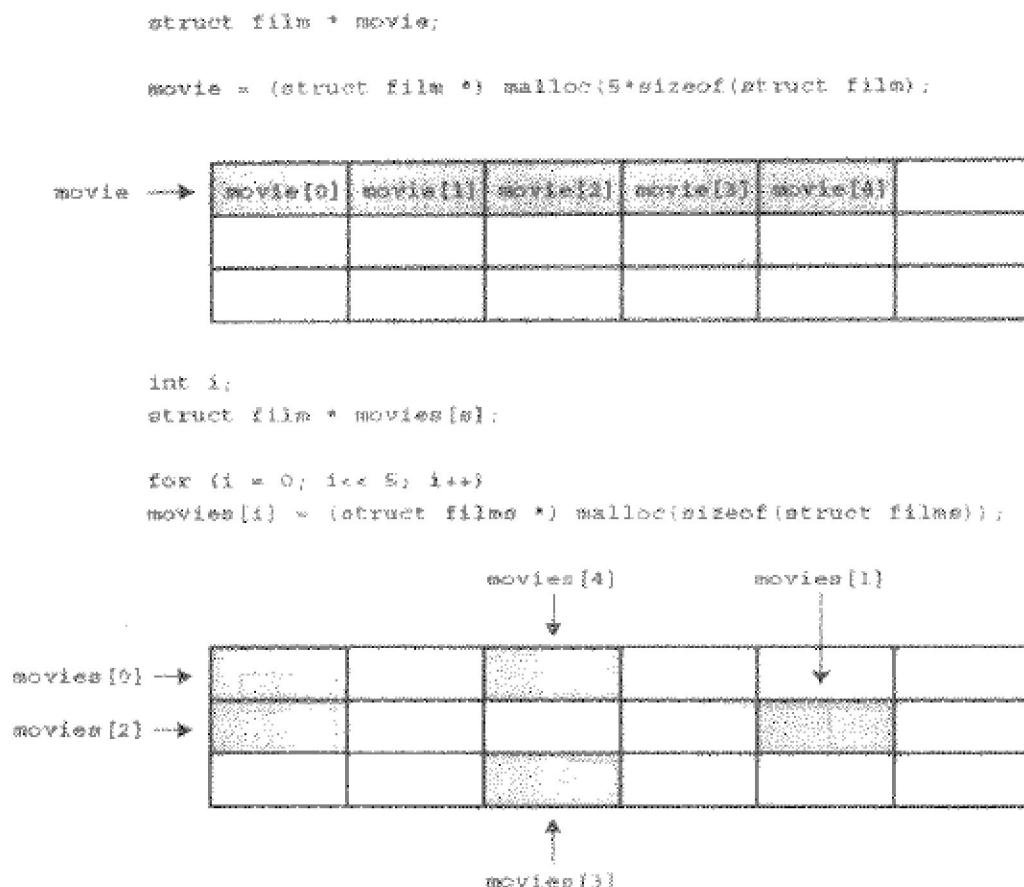


图 17.1 按块分配结构空间和个别地分配结构空间

170.

```
#define TSIZE 45
struct film {
    char title[TSIZE];
    int rating;
    struct film * next;
};
struct film * head;
```



图 17.2 链表中的第一项

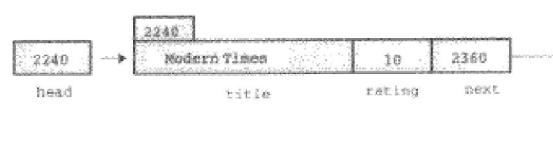


图 17.3 含有两个项目的链表

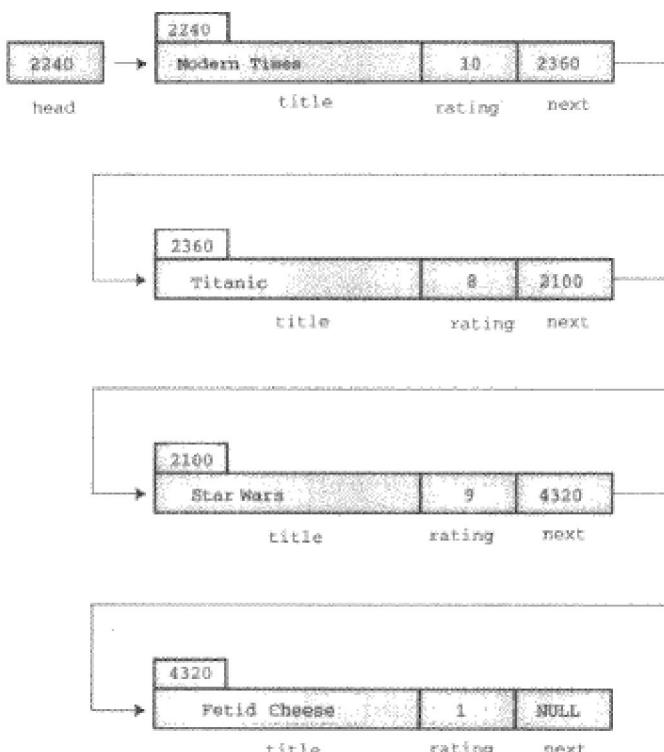


图 17.4 含有多个项目的链表

171. 使用结构链表：

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#define TSIZE 45

struct film {
    char title [TSIZE];
    int rating;
    struct film * next;
};

int main (void)
{
    struct film * head = NULL;
    struct film * prev, * current;
    char input[TSIZE];

    puts("enter first movie title:");
}
```

```
while(gets(input)!=NULL && input[0] != '\0')
{
    current = (struct film *) malloc(sizeof(struct film));
    if (head == NULL)
        head = current;
    else
        prev->next = current;
    current->next = NULL;
    strcpy(current->title,input);
    puts("enter your rating:");
    scanf("%d",&current->rating);
    while(getchar() != '\n')
        continue;
    puts("enter next movie title (empty line to stop):");
    prev = current;
}
if(head == NULL)
    printf("no data entered.");
else
    printf("here is the movie list:\n");
current = head;
while (current != NULL)
{
    printf("Movie:%s Rating:%d\n",current->title,current->rating);
    current = current->next;
}

current = head;
while (current!= NULL)
{
    free(current);
    current = current->next;
}
printf("bye!\n");
return 0;
}
```

172. 为类型的属性和可对类型执行的操作提供一个抽象的描述，这样一种正式的抽象描述被称为抽象数据类型（ADT）。

表 17.1

列表类型总结

类 型 名 称:	简 单 列 表
类型属性:	可保存一个项目序列
类型操作:	把列表初始化为空列表
	确定列表是否为空
	确定列表是否已满
	确定列表中项目的个数
	向列表末尾添加项目
	遍历列表，处理列表中每个项目
	清空列表

173. 把实现和最终接口相隔离的做法对于大型编程工程来说尤其有用，这称为数据隐藏，因为详细的数据表示对终端用户是不可见的。
174. 用抽象数据类型方法进行 C 语言编程包含下面三个步骤：
1. 以抽象、通用的方式描述一个类型，包括其操作。
 2. 设计一个函数接口来表示这种新类型。
 3. 编写具体代码以实现这个接口。
175. 队列是具有两大特殊属性的列表。第一，新的项目只能被添加到列表结尾处；第二，项目只能从列表开始处被移除，它是一种“先进先出”的数据形式。

表 17.2

队列类型总结

类 型 名 称:	队 列
类型属性:	可保存一个规则的项目序列
类型操作:	把队列初始化为空队列
	确定队列是否为空
	确定队列是否已满
	确定队列中的项目数
	向队列尾端添加项目
	从队列首端删除和恢复项目
	清空队列

176.

表 17.3

比较数组和链表

数 据 形 式	优 点	缺 点
数 组	C 对其直接支持 提供随机访问	编译时决定其大小 插入和删除元素很费时
链 表	运行时决定其大小 快速插入和删除元素	不能随机访问 用户必须提供编程支持

向数组插入一个元素，必须移动其他元素以便安插新元素，新元素离数组头越近，要移动的元素越多。而向链表插入一个节点，只需分配两个指针值。类似地，从数组中删除一个元素要重新安置大量元素，而从链表中删除一个节点只需重新设置一个指针并释放被删除节点使用的内存。对数组来说，可以用数组索引直接访问任意元素，这被称为随机访问。对链表来说，必须从列表头开始，逐个节点地移动到所需的节点处，这叫做顺序访问。数组也可以顺序访问，只要在数组中按顺序使用数组索引递增即可。

177. 折半搜索法：n 次比较能处理具有 $2^n - 1$ 个成员的数组。

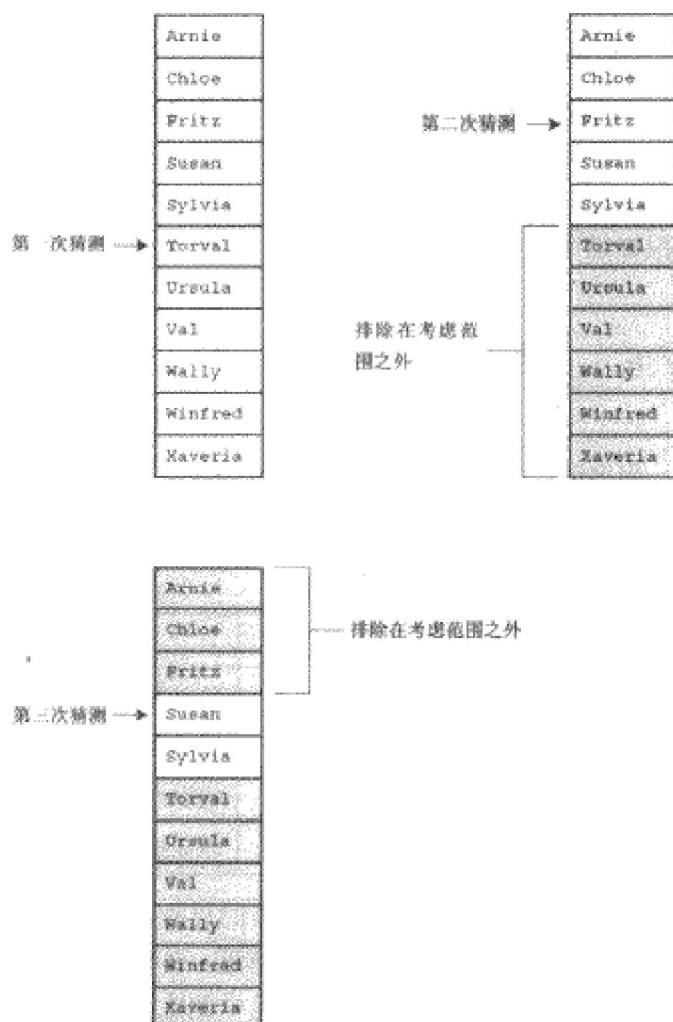


图 17.11 以折半搜索法查找 Susan

178. 二叉搜索树：一种既支持频繁地插入和删除元素又支持频繁搜索的数据类型，链表和数组都不是针对这个目标的理想选择。

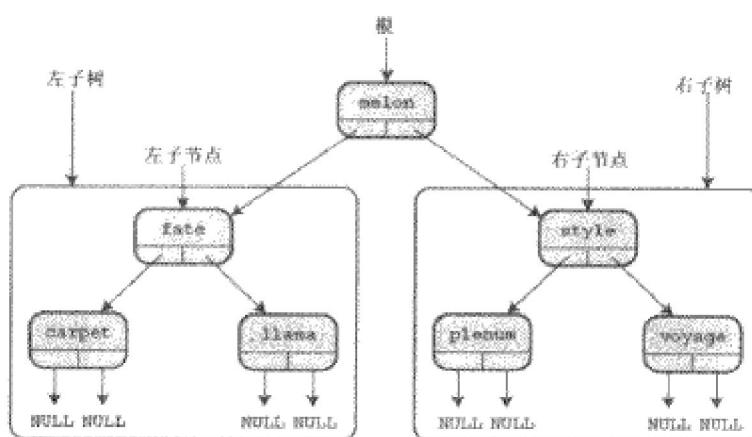


图 17.12 一个存储单词的二叉搜索树

179. 删除节点：

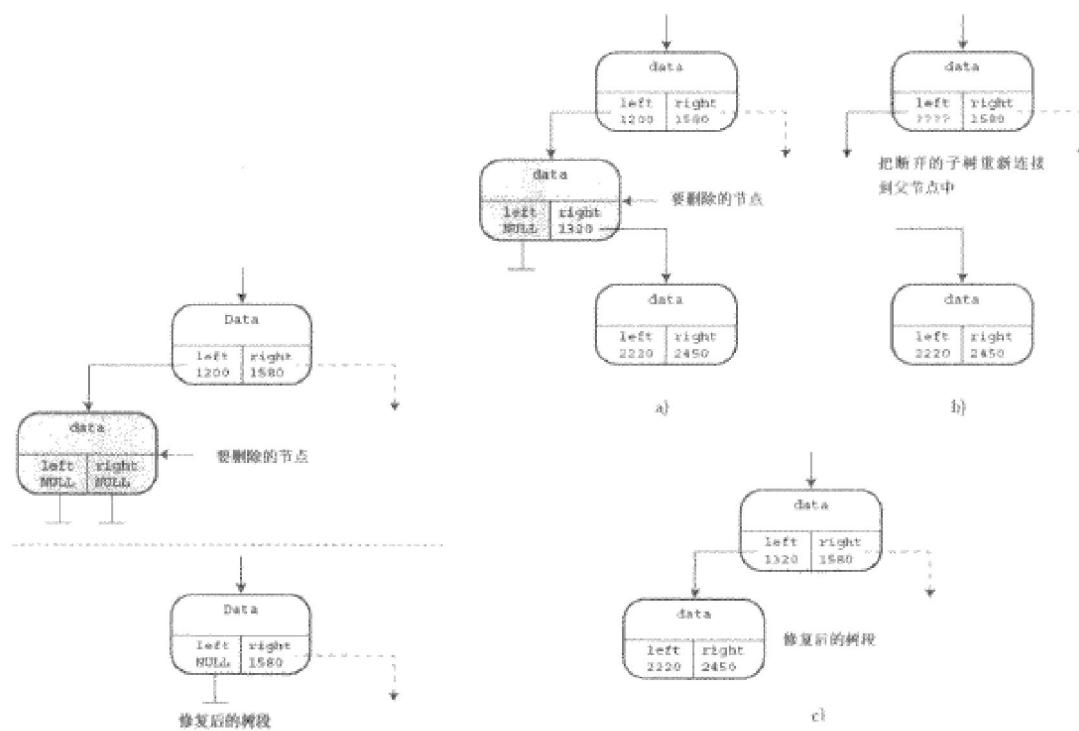
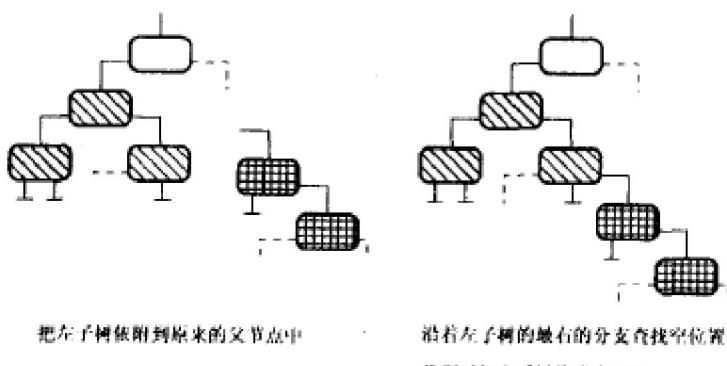
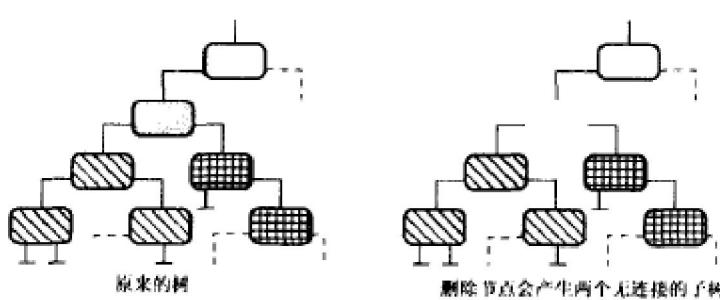


图 17.14 删有一个子节点的节点



180. 树类型的支持函数：

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/*局部数据类型 */
typedef struct pair {
```

```
Node * parent;
Node * child;
} Pair;

/*局部函数原型 */
static Node * MakeNode(const Item * pi);
static boolToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode (Node * new_node, Node * root);
static void InOrder(const Node * root, void (* pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Node **ptr);
static void DeleteAllNodes(Node * ptr);

/* 函数定义 */
void InitializeTree(Tree * ptree)
{
    ptree->root = NULL;
    ptree->size = 0;
}

bool TreeIsEmpty(const Tree * ptree)
{
    if (ptree->root == NULL)
        return true;
    else
        return false;
}

bool TreeIsFull(const Tree * ptree)
{
    if (ptree->size == MAXITEMS)
        return true;
    else
        return false;
}

int TreeItemCount(const Tree * ptree)
{
    return ptree->size;
}

bool AddItem(const Item * pi, Tree * ptree)
{
    Node * new_node;

    if  (TreeIsFull(ptree))
    {
```

```
fprintf(stderr,"Tree is full\n");
return false;           /* 提前返回 */
}

if (SeekItem(pi, ptree).child != NULL)
{
    fprintf(stderr, "Attempted to add duplicate item\n");
    return false;           /* 提前返回 */
}

new_node = MakeNode(pi);      /* 指向新节点 */
if (new_node == NULL)
{
    fprintf(stderr, "Couldn't create node\n");
    return false;           /* 提前返回 */
}

/* 成功创建了一个新节点 */
ptree->size++;

if (ptree->root == NULL)      /* 情况 1：树为空树 */
    ptree->root = new_node;   /* 新节点即为树的根节点 */
else                          /* 情况 2：树非空 */
    AddNode(new_node, ptree->root); /* 把新节点添加到树中 */

return true;                  /* 成功返回 */
}

bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}

bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;

    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;

    if (look.parent == NULL)      /* 删除根项目 */
        DeleteNode(&ptree->root);
    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;

    return true;
}
```

```
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{
    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}

/* 局部函数 */
static void InOrder(const Node * root, void (* pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

static void DeleteAllNodes(Node * root)
{
    Node * pright;

    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
        free(root);
        DeleteAllNodes(pright);
    }
}

static void AddNode (Node * new_node, Node * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL)      /* 空子树      */
            root->left = new_node;  /* 因此把节点添加到此处      */
        else
    }
}
```

```
        AddNode(new_node, root->left);/*否则处理该子树*/
    }

    else if (ToLeft(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else /* 不应含有相同的项目 */
    {
        fprintf(stderr,"location error in AddNode()\n");
        exit(1);
    }
}

static bool ToLeft(const Item * i1, const Item * i2)
{
    int comp1;

    if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (comp1 == 0 && strcmp(i1->petkind, i2->petkind) < 0 )
        return true;
    else
        return false;
}

static bool ToRight(const Item * i1, const Item * i2)
{
    int comp1;

    if ((comp1 = strcmp(i1->petname, i2->petname)) > 0)
        return true;
    else if (comp1 == 0 && strcmp(i1->petkind, i2->petkind) > 0 )
        return true;
    else
        return false;
}

static Node * MakeNode(const Item * pi)
{
    Node * new_node;

    new_node = (Node *) malloc(sizeof(Node));
    if (new_node != NULL)
    {
        new_node->item = *pi;
```

```
    new_node->left = NULL;
    new_node->right = NULL;
}

return new_node;
}

static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look; /* 提前返回 */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else /*如果前面两种情况都不满足，必定为相等的情况 */
            break; /* look.child 是目标项目节点的地址 */
    }

    return look; /*成功返回 */
}

static void DeleteNode(Node **ptr)
/* ptr 是指向目标节点的父节点指针成员的地址 */
{
    Node * temp;

    puts((*ptr)->item.petname);
    if ((*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ((*ptr)->right == NULL)
```

```
{  
    temp = *ptr;  
    *ptr = (*ptr)->left;  
    free(temp);  
}  
else /* 被删除节点有两个子节点 */  
{  
    /*找到右子树的依附位置 */  
    for (temp = (*ptr)->left; temp->right != NULL;  
         temp = temp->right)  
        continue;  
    temp->right = (*ptr)->right;  
    temp = *ptr;  
    *ptr = (*ptr)->left;  
    free(temp);  
}  
}
```

181. 二叉树搜索只在满员（或者称为平衡）时效率最高，而搜索不平衡二叉树时并不比顺序搜索链表来得更快。

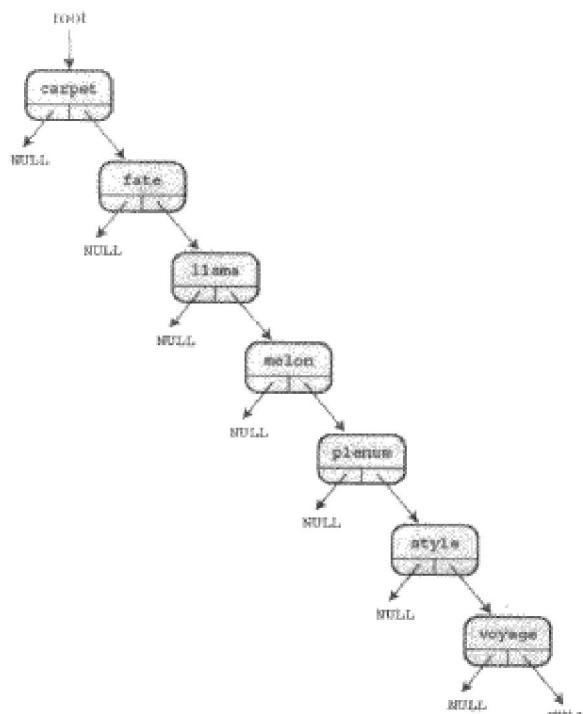


图 17.16 不平衡的二叉搜索树

182. AVL 树是最先发明的自平衡二叉查找树。在 AVL 树中任何节点的两个儿子子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是 $O(\log n)$ 。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

