C语言宝典

2011年1月

1912 制作

第一部分：基本概念及其它问答题

1、关键字 static 的作用是什么？

这个简单的问题很少有人能回答完全。在 C 语言中，关键字 static 有三个明显的作用：

1)．在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。

2)．在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。

3)．在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

大多数应试者能正确回答第一部分，一部分能正确回答第二部分，同是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数

据和代码范围的好处和重要性。

2、“引用”与指针的区别是什么？

答 、1) 引用必须被初始化，指针不必。

2) 引用初始化以后不能被改变，指针可以改变所指的对象。

3) 不存在指向空值的引用，但是存在指向空值的指针。

指针通过某个指针变量指向一个对象后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用

3、.h 头文件中的 ifndef/define/endif 的作用？

答：防止该头文件被重复引用。

4、#include<file.h> 与 #include "file.h"的区别？

答：前者是从 Standard Library 的路径寻找和引用 file.h，而后者是从当前工作路径搜寻并引用 file.h。

5、描述实时系统的基本特性

答 ：在特定时间内完成特定的任务，实时性与可靠性。

6、全局变量和局部变量在内存中是否有区别？如果有，是什么区别？

答 ：全局变量储存在静态数据区，局部变量在堆栈中。

7、什么是平衡二叉树？

答 ：左右子树都是平衡二叉树 且左右子树的深度差值的绝对值不大于 1。

8、堆栈溢出一般是由什么原因导致的？

答 ：1.没有回收垃圾资源

2.层次太深的递归调用

9、冒泡排序算法的时间复杂度是什么？

答 ：O(n^2)

10、什么函数不能声明为虚函数？

答：constructor

11、队列和栈有什么区别？

答：队列先进先出，栈后进先出

12、不能做 switch()的参数类型

答 ： switch 的参数不能为实型。

13、局部变量能否和全局变量重名？

答：能，局部会屏蔽全局。要用全局变量，需要使用"::"

局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内

14、如何引用一个已经定义过的全局变量？

答 、可以用引用头文件的方式，也可以用 extern 关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变量，假定你将那个变量写错了，那么在编译期间会报错，如果你用 extern 方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。

15、全局变量可不可以定义在可被多个.C 文件包含的头文件中？为什么？

答 、可以，在不同的 C 文件中以 static 形式来声明同名全局变量。

可以在不同的 C 文件中声明同名的全局变量，前提是其中只能有一个 C 文件中对此变量赋初值，此时连接不会出错。

16、语句 for(； 1 ；)有什么问题？它是什么意思？

答 、和 while(1)相同，无限循环。

17、do……while 和 while……do 有什么区别？

答 、前一个循环一遍再判断，后一个判断以后再循环。

18、statac 全局变量、局部变量、函数与普通全局变量、局部变量、函数

static 全局变量与普通的全局变量有什么区别？static 局部变量和普通局部变量有什么区别？static 函数与普通函数有什么区别？

答 、全局变量(外部变量)的说明之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。 这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误。

从以上分析可以看出， 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域， 限制了它的使用范围。

static 函数与普通函数作用域不同。仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件

static 全局变量与普通的全局变量有什么区别：static 全局变量只初使化一次，防止在其他文件单元中被引用;

static 局部变量和普通局部变量有什么区别：static 局部变量只被初始化一次，下一次依据上一次结果值;

static 函数与普通函数有什么区别：static 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝

19、程序的内存分配

答：一个由 c/C++编译的程序占用的内存分为以下几个部分

1、栈区（stack）—由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap）—一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
3、全局区（静态区）（static）—全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。
4、文字常量区—常量字符串就是放在这里的。程序结束后由系统释放。
5、程序代码区—存放函数体的二进制代码

例子程序
这是一个前辈写的，非常详细

```
//main.cpp
  int a=0;    //全局初始化区
  char *p1;   //全局未初始化区
  main()
  {
   intb;栈
   char s[]="abc";   //栈
   char *p2;         //栈
   char *p3="123456";   //123456\0 在常量区，p3 在栈上。
   static int c=0;    //全局（静态）初始化区
   p1 = (char*)malloc(10);
   p2 = (char*)malloc(20);   //分配得来得 10 和 20 字节的区域就在堆区。
   strcpy(p1,"123456");   //123456\0 放在常量区，编译器可能会将它与 p3 所向"123456"优化成一个地方。
}
```

20、解释堆和栈的区别
答：堆（heap）和栈(stack)的区别
（1）申请方式
stack:由系统自动分配。例如，声明在函数中一个局部变量 int b;系统自动在栈中为 b 开辟空间
heap:需要程序员自己申请，并指明大小，在 c 中 malloc 函数
如 p1=(char*)malloc(10);
在 C++中用 new 运算符
如 p2=(char*)malloc(10);
但是注意 p1、p2 本身是在栈中的。

（2）申请后系统的响应
栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。
堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，
会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

（3）申请大小的限制
栈：在 Windows 下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。
堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

（4）申请效率的比较：
栈:由系统自动分配，速度较快。但程序员是无法控制的。
堆:是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.
另外，在 WINDOWS 下，最好的方式是用 Virtual Alloc 分配内存，他不是在堆，也不是在栈,而是直接在进程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。

（5）堆和栈中的存储内容
栈：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。
当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

（6）存取效率的比较

```
char s1[]="aaaaaaaaaaaaaaa";
char *s2="bbbbbbbbbbbbbbbbb";
```

aaaaaaaaaaa 是在运行时刻赋值的；
而 bbbbbbbbbbb 是在编译时就确定的；
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。
比如：

```
#include
voidmain()
{
char a=1;
char c[]="1234567890";
char *p="1234567890";
a = c[1];
a = p[1];
return;
}
```

对应的汇编代码

```
10:a=c[1];
004010678A4DF1movcl,byteptr[ebp-0Fh]
0040106A884DFCmovbyteptr[ebp-4],cl
11:a=p[1];
0040106D8B55ECmovedx,dwordptr[ebp-14h]
004010708A4201moval,byteptr[edx+1]
004010738845FCmovbyteptr[ebp-4],al
```

第一种在读取时直接就把字符串中的元素读到寄存器 cl 中，而第二种则要先把指针值读到 edx 中，在根据 edx 读取字符，显然慢了。

21、什么是预编译,何时需要预编译?

答：预编译又称为预处理,是做些代码文本的替换工作。处理#开头的指令,比如拷贝#include 包含的文件代码，#define 宏定义的替换,条件编译等，就是为编译做的预备工作的阶段，主要处理#开始的预编译指令，预编译指令指示了在程序正式编译前就由编译器进行的操作，可以放在程序中的任何位置。

c 编译系统在对程序进行通常的编译之前，先进行预处理。c 提供的预处理功能主要有以下三种：1）宏定义 2）文件包含　3）条件编译

1、 总是使用不经常改动的大型代码体。
2、程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

22、关键字 const 是什么含意？

答：我只要一听到被面试者说：“const 意味着常数”，我就知道我正在和一个业余者打交道。去年 Dan Saks 已经在他的文章里完全概括了 const 的所有用法，因此 ESP(译者：Embedded Systems Programming)的每一位读者应该非常熟悉 const 能做什么和不能做什么.如果你从没有读到那篇文章，只要能说出 const 意味着“只读”就可以了。尽管这个答案不是完全的答案，但我接受它作为一个正确的答案。(如果你想知道更详细的答案，仔细读一下 Saks 的文章吧。）如果应试者能正确回答这个问题，我将问他一个附加的问题：下面的声明都是什么意思？

```
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
```

前两个的作用是一样，a 是一个常整型数。第三个意味着 a 是一个指向常整型数的指针（也就是，整型数是不可修改的，但指针可以)。第四个意思 a 是一个指向整型数的常指针（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的)。最后一个意味着 a 是一个指向常整型数的常指针（也就是说，指针指向的整型数是不可修改的，同时指针也是不可修改的)。如果应试者能正确回答这些问题，那么他就给我留下了一个好印象。顺带提一句，也许你可能会问，即使不用关键字 const，也还是能很容易写出功能正确的程序，那么我为什么还要如此看重关键字 const 呢？我也如下的几下理由：

1). 关键字 const 的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。(当然，懂得用 const 的程序员很少会留下的垃圾让别人来清理的。)

2). 通过给优化器一些附加的信息，使用关键字 const 也许能产生更紧凑的代码。

3). 合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少 bug 的出现

23、关键字 volatile 有什么含意 并给出三个不同的例子。

答：一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：

1). 并行设备的硬件寄存器（如：状态寄存器）

2). 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3). 多线程应用中被几个任务共享的变量

回答不出这个问题的人是不会被雇佣的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。嵌入式系统程序员经常同硬件、中断、RTOS 等等打交道，所用这些都要求 volatile 变量。不懂得 volatile 内容将会带来灾难。

假设被面试者正确地回答了这是问题（嗯，怀疑这否会是这样)，我将稍微深究一下，看一下这家伙是不是直正懂得 volatile 完全的重要性。

1). 一个参数既可以是 const 还可以是 volatile 吗？解释为什么。

2). 一个指针可以是 volatile 吗？解释为什么。

3). 下面的函数有什么错误：

```
int square(volatile int *ptr)
{
return *ptr * *ptr;
}
```

下面是答案：

1)．是的。一个例子是只读的状态寄存器。它是 volatile 因为它可能被意想不到地改变。它是 const 因为程序不应该试图去修改它。
2)．是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个 buffer 的指针时。
3)．这段代码的有个恶作剧。这段代码的目的是用来返指针*ptr 指向值的平方，但是，由于*ptr 指向一个 volatile 型参数，编译器将产生类似下面的代码：

```
int square(volatile int *ptr)
{
int a,b;
a = *ptr;
b = *ptr;
return a * b;
}
```

由于*ptr 的值可能被意想不到地该变，因此 a 和 b 可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

```
long square(volatile int *ptr)
{
int a;
a = *ptr;
return a * a;
}
```

24、三种基本的数据模型
答：按照数据结构类型的不同，将数据模型划分为层次模型、网状模型和关系模型。

25、结构与联合有和区别？
答：(1)．结构和联合都是由多个不同的数据类型成员组成，但在任何同一时刻，联合中只存放了一个被选中的成员（所有成员共用一块地址空间），而结构的所有成员都存在（不同成员的存放地址不同）。
(2)．对于联合的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于结构的不同成员赋值是互不影响的

26、描述内存分配方式以及它们的区别?
答：1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。
2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集。
3） 从堆上分配，亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多

27、请说出 const 与#define 相比，有何优点？
答：Const 作用：定义常量、修饰函数参数、修饰函数返回值三个作用。被 Const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。
1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。
2） 有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

28、简述数组与指针的区别？
答：数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。
(1)修改内容上的差别

```
char a[] = “hello”;
```

```
a[0] = 'X';
char *p = "world"; // 注意 p 指向常量字符串
p[0] = 'X'; // 编译器不能发现该错误，运行时错误
```

(2) 用运算符 sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个 指针变量的字节数，而不是 p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节
计算数组和指针的内存容量
void Func(char a[100])
{
 cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

29、分别写出 BOOL,int,float,指针类型的变量 a 与"零"的比较语句。

答：BOOL : if (!a) or if(a)

```
int :     if ( a == 0)
float :   const EXPRESSION EXP = 0.000001
          if ( a < EXP && a >-EXP)
pointer : if ( a != NULL) or if(a == NULL)
```

30、如何判断一段程序是由 C 编译程序还是由 C++编译程序编译的？

答：

```
#ifdef __cplusplus
cout<<"c++";
#else
cout<<"c";
#endif
```

31、论述含参数的宏与函数的优缺点

答：

	带参宏	函数
处理时间	编译时	程序运行时
参数类型	没有参数类型问题	定义实参、形参类型
处理过程	不分配内存	分配内存
程序长度	变长	不变
运行速度	不占运行时间	调用和返回占用时间

32、用两个栈实现一个队列的功能？要求给出算法和思路！

答 、设 2 个栈为 A,B, 一开始均为空.

入队:

将新元素 push 入栈 A;

出队:

(1)判断栈 B 是否为空；

(2)如果不为空，则将栈 A 中所有元素依次 pop 出并 push 到栈 B；

(3)将栈 B 的栈顶元素 pop 出；

这样实现的队列入队和出队的平摊复杂度都还是 O(1), 比上面的几种方法要好

33、嵌入式系统中经常要用到无限循环，你怎么样用 C 编写死循环呢？

答：这个问题用几个解决方案。我首选的方案是：

```
while(1)
{
```

```
}
```
一些程序员更喜欢如下方案：
```
for(;;)
{
}
```
这个实现方式让我为难，因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这个作为方案，我将用这个作为一个机会去探究他们这样做的
基本原理。如果他们的基本答案是：“我被教着这样做，但从没有想到过为什么。”这会给我留下一个坏印象。
第三个方案是用 goto
```
Loop:
...
goto Loop;
```
应试者如给出上面的方案，这说明或者他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。

34、位操作（Bit manipulation）

答： 嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清除 a 的 bit 3。在以上两个操作中，要保持其它位不变。
对这个问题有三种基本的反应
1)不知道如何下手。该被面者从没做过任何嵌入式系统的工作。
2) 用 bit fields。Bit fields 是被扔到 C 语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了的你的代码是不可重用的。我最近不幸看到 Infineon 为其较复杂的通信芯片写的驱动程序，它用到了 bit fields 因此完全对我无用，因为我的编译器用其它的方式来实现 bit fields 的。从道德讲：永远不要让一个非嵌入式的家伙粘实际硬件的边。
3) 用 #defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。最佳的解决方案如下：
```
#define BIT3 (0x1 << 3)
static int a;

void set_bit3(void)
{
    a |= BIT3;
}
void clear_bit3(void)
{
    a &= ~BIT3;
}
```
一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数，这也是可以接受的。我希望看到几个要点：说明常数、|=和&=~操作。

35、访问固定的内存位置（Accessing fixed memory locations）

答：嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为 0x67a9 的整型变量的值为 0xaa66。编译器是一个纯粹的 ANSI 编译器。写代码去完成这一任务。
这一问题测试你是否知道为了访问一绝对地址把一个整型数强制转换（typecast）为一指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下：
```
    int *ptr;
    ptr = (int *)0x67a9;
    *ptr = 0xaa66;
 A more obscure approach is:
```
一个较晦涩的方法是：

```
 *(int * const)(0x67a9) = 0xaa55;
```
即使你的品味更接近第二种方案，但我建议你在面试时使用第一种方案。

36、中断（Interrupts）

答： 中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展—让标准 C 支持中断。具代表事实是，产生了一个新的关键字 __interrupt。下面的代码就使用了__interrupt 关键字去定义了一个中断服务子程序(ISR)，请评论一下这段代码的。

```
__interrupt double compute_area (double radius)
{
    double area = PI * radius * radius;
    printf("\nArea = %f", area);
    return area;
}
```
这个函数有太多的错误了，以至让人不知从何说起了：

1)ISR 不能返回一个值。如果你不懂这个，那么你不会被雇用的。

2) ISR 不能传递参数。如果你没有看到这一点，你被雇用的机会等同第一项。

3) 在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈，有些处理器/编译器就是不允许在 ISR 中做浮点运算。此外，ISR 应该是短而有效率的，在 ISR 中做浮点运算是不明智的。

4) 与第三点一脉相承，printf()经常有重入和性能上的问题。如果你丢掉了第三和第四点，我不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。

37、动态内存分配（Dynamic memory allocation）

答：尽管不像非嵌入式计算机那么常见，嵌入式系统还是有从堆（heap）中动态分配内存的过程的。那么嵌入式系统中，动态分配内存可能发生的问题是什么？

这里，我期望应试者能提到内存碎片，碎片收集的问题，变量的持行时间等等。这个主题已经在 ESP 杂志中被广泛地讨论过了（主要是 P.J. Plauger，他的解释远远超过我这里能提到的任何解释），所有回过头看一下这些杂志吧！让应试者进入一种虚假的安全感觉后，我拿出这么一个小节目：

下面的代码片段的输出是什么，为什么？

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL)
    puts("Got a null pointer");
else
    puts("Got a valid pointer");
```
这是一个有趣的问题。最近在我的一个同事不经意把 0 值传给了函数 malloc，得到了一个合法的指针之后，我才想到这个问题。这就是上面的代码，该代码的输出是"Got a valid pointer"。我用这个来开始讨论这样的一问题，看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要，但解决问题的方法和你做决定的基本原理更重要些。

38、Typedef

答：Typedef 在 C 语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如，思考一下下面的例子：

```
#define dPS struct s *
typedef struct s * tPS;
```
以上两种情况的意图都是要定义 dPS 和 tPS 作为一个指向结构 s 指针。哪种方法更好呢？（如果有的话）为什么？

这是一个非常微妙的问题，任何人答对这个问题（正当的原因）是应当被恭喜的。答案是：typedef 更好。思考下面的例子：

```
dPS p1,p2;
```

tPS p3,p4;
第一个扩展为
struct s * p1, p2;
上面的代码定义 p1 为一个指向结构的指，p2 为一个实际的结构，这也许不是你想要的。第二个例子正确地定义了 p3 和 p4 两个指针。

39、用变量 a 给出下面的定义

答：a) 一个整型数（An integer）

b) 一个指向整型数的指针（A pointer to an integer）
c) 一个指向指针的的指针，它指向的指针是指向一个整型数（A pointer to a pointer to an integer）
d) 一个有 10 个整型数的数组（An array of 10 integers）
e) 一个有 10 个指针的数组，该指针是指向一个整型数的（An array of 10 pointers to integers）
f) 一个指向有 10 个整型数数组的指针（A pointer to an array of 10 integers）
g) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer）
h) 一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数（ An array of ten pointers to functions that take an integer
argument and return an integer ）

答案是：
a) int a; // An integer
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

40、解释局部变量、全局变量和静态变量的含义。
答：

41、写一个“标准”宏

答：交换两个参数值的宏定义为：. #define SWAP(a,b)\
(a)=(a)+(b);\
(b)=(a)-(b);\
(a)=(a)-(b);
输入两个参数，输出较小的一个：#define MIN(A,B) ((A) < (B))? (A) : (B))
表明 1 年中有多少秒（忽略闰年问题）：#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
#define DOUBLE(x) x+x 与 #define DOUBLE(x) ((x) + (x))
i = 5*DOUBLE(5); i 为 30 i = 5*DOUBLE(5); i 为 50
已知一个数组 table，用一个宏定义，求出数据的元素个数
#define NTBL
#define NTBL (sizeof(table)/sizeof(table[0]))

42、A.c 和 B.c 两个 c 文件中使用了两个相同名字的 static 变量,编译的时候会不会有问题?这两个 static 变量会保存到哪里（栈还是堆或者其他的）?

答：static 的全局变量，表明这个变量仅在本模块中有意义，不会影响其他模块。
他们都放在数据区，但是编译器对他们的命名是不同的。
如果要使变量在其他模块也有意义的话，需要使用 extern 关键字。

43、一个单向链表，不知道头节点,一个指针指向其中的一个节点，问如何删除这个指针指向的节点？

答：将这个指针指向的 next 节点值 copy 到本节点，将 next 指向 next->next,并随后删除原 next 指向的节点。

第二部分：程序代码评价或者找错

1、下面的代码输出是什么，为什么？

```
void foo(void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

这个问题测试你是否懂得 C 语言中的整数自动转换原则，我发现有些开发者懂得极少这些东西。不管如何，这无符号整型问题的答案是输出是 ">6"。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。因此-20 变成了一个非常大的正整数，所以该表达式计算出的结果大于 6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是丰常重要的。如果你答错了这个问题，你也就到了得不到这份工作的边缘。

2、评价下面的代码片断：

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

对于一个 int 型不是 16 位的处理器为说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ~0;
```

这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里，好的嵌入式程序员非常准确地明白硬件的细节和它的局限，然而 PC 机程序往往把硬件作为一个无法避免的烦恼。

3、 C 语言同意一些令人震惊的结构,下面的结构是合法的吗，如果是它做些什么？

```
int a = 5, b = 7, c;
c = a+++b;
```

这个问题将做为这个测验的一个愉快的结尾。不管你相不相信，上面的例子是完全合乎语法的。问题是编译器如何处理它？水平不高的编译作者实际上会争论这个问题，根据最处理原则，编译器应当能处理尽可能所有合法的用法。因此，上面的代码被处理成：

```
c = a++ + b;
```

因此，这段代码持行后 a = 6, b = 7, c = 12。
如果你知道答案，或猜出正确答案，做得好。如果你不知道答案，我也不把这个当作问题。我发现这个问题的最大好处是这是一个关于代码编写风格，代码的可读性，代码的可修改性的好的话题。

4、设有以下说明和定义：

```
typedef union {long i; int k[5]; char c;} DATE;
struct data { int cat; DATE cow; double dog;} too;
DATE max;
```

则语句 printf("%d",sizeof(struct date)+sizeof(max));的执行结果是？

答 、结果是：52。DATE 是一个 union，变量公用空间．里面最大的变量类型是 int[5]，占用 20 个字节．所以它的大小是 20
data 是一个 struct，每个变量分开占用空间．依次为 int4 + DATE20 + double8 = 32.
所以结果是 20 + 32 = 52.
当然...在某些 16 位编辑器下，int 可能是 2 字节,那么结果是 int2 + DATE10 + double8 = 20

5、请写出下列代码的输出内容

```
#include<stdio.h>
main()
{
int a,b,c,d;
a=10;
b=a++;
c=++a;
d=10*a++;
printf("b，c，d：%d，%d，%d"，b，c，d）;
return 0;
}
```

答：10，12，120

6、写出下列代码的输出内容

```
#include<stdio.h>
int inc(int a)
{
return(++a);
}
int multi(int*a,int*b,int*c)
{
return(*c=*a**b);
}
typedef int(FUNC1)(int in);
typedef int(FUNC2) (int*,int*,int*);

void show(FUNC2 fun,int arg1, int*arg2)
{
INCp=&inc;
int temp =p(arg1);
fun(&temp,&arg1, arg2);
printf("%d\n",*arg2);
}

main()
{
int a;
show(multi,10,&a);
return 0;
}
```

答：110

7、请找出下面代码中的所以错误

说明：以下代码是把一个字符串倒序，如“abcd”倒序后变为“dcba”

```
1、#include"string.h"
2、main()
3、{
4、 char*src="hello,world";
5、 char* dest=NULL;
6、 int len=strlen(src);
7、 dest=(char*)malloc(len);
8、 char* d=dest;
9、 char* s=src[len];
10、 while(len--!=0)
11、 d++=s--;
12、 printf("%s",dest);
13、 return 0;
14、}
```

答：

方法 1：

```
int main(){
char* src = "hello,world";
int len = strlen(src);
char* dest = (char*)malloc(len+1);//要为\0 分配一个空间
char* d = dest;
char* s = &src[len-1];//指向最后一个字符
while( len-- != 0 )
*d++=*s--;
*d = 0;//尾部要加\0
printf("%s\n",dest);
free(dest);// 使用完，应当释放空间，以免造成内存汇泄露
return 0;
}
```

方法 2：

```
#include <stdio.h>
#include <string.h>
main()
{
char str[]="hello,world";
int len=strlen(str);
char t;
for(int i=0; i<len/2; i++)
{
t=str[i];
str[i]=str[len-i-1]; str[len-i-1]=t;
}
printf("%s",str);
return 0;
}
```

8、请问下面程序有什么错误?

```
int a[60][250][1000],i,j,k;
```

```
   for(k=0;k<=1000;k++)
    for(j=0;j<250;j++)
     for(i=0;i<60;i++)
      a[i][j][k]=0;
```
答案：把循环语句内外换一下

9、请问下面程序会出现什么情况？

```
.    #define Max_CB 500
    void LmiQueryCSmd(Struct MSgCB * pmsg)
     {
     unsigned char ucCmdNum;
     ......

     for(ucCmdNum=0;ucCmdNum<Max_CB;ucCmdNum++)
      {
      ......;
      }
```
答案：死循环

10、以下 3 个有什么区别

```
char * const p; //常量指针，p 的值不可以修改
char const * p；//指向常量的指针，指向的常量值不可以改
const char *p； //和 char const *p
```

11、写出下面的结果

```
char str1[] = "abc";
char str2[] = "abc";

const char str3[] = "abc";
const char str4[] = "abc";

const char *str5 = "abc";
const char *str6 = "abc";

char *str7 = "abc";
char *str8 = "abc";

cout << ( str1 == str2 ) << endl;
cout << ( str3 == str4 ) << endl;
cout << ( str5 == str6 ) << endl;
cout << ( str7 == str8 ) << endl;
```

结果是：0 0 1 1
解答：str1,str2,str3,str4 是数组变量，它们有各自的内存空间；
而 str5,str6,str7,str8 是指针，它们指向相同的常量区域。

12、以下代码中的两个 sizeof 用法有问题吗？

```
void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
```

```
    for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
        if( 'a'<=str[i] && str[i]<='z' )
            str[i] -= ('a'-'A' );
}
char str[] = "aBcDe";
cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
UpperCase( str );
cout << str << endl;
```

答：函数内的 sizeof 有问题。根据语法，sizeof 如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的 str 是一个静态定义的数组，因此其大小为 6，函数内的 str 实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此 sizeof 作用于上只将其当指针看，一个指针为 4 个字节，因此返回 4。

13、写出输出结果

```
main()
{
  int a[5]={1,2,3,4,5};
   int *ptr=(int *)(&a+1);
   printf("%d,%d",*(a+1),*(ptr-1));
}
```

输出：2,5

(a+1）就是 a[1]，(ptr-1)就是 a[4],执行结果是 2，5

&a+1 不是首地址+1，系统会认为加一个 a 数组的偏移，是偏移了一个数组的大小（本例是 5 个 int）

int *ptr=(int *)(&a+1);

则 ptr 实际是&(a[5]),也就是 a+5

原因如下：

&a 是数组指针，其类型为 int (*)[5];

而指针加 1 要根据指针类型加上一定的值，

不同类型的指针+1 之后增加的大小不同

a 是长度为 5 的 int 数组指针，所以要加 5*sizeof(int)

所以 ptr 实际是 a[5]

但是 prt 与(&a+1)类型是不一样的(这点很重要)

所以 prt-1 只会减去 sizeof(int*)

a,&a 的地址是一样的，但意思不一样，a 是数组首地址，也就是 a[0]的地址，&a 是对象（数组）首地址，a+1 是数组下一元素的地址，即 a[1],&a+1 是下一个对象的地址，即 a[5].

14、请问以下代码有什么问题：

```
int  main()
{
char a;
char *str=&a;
strcpy(str,"hello");
printf(str);
return 0;
}
```

没有为 str 分配内存空间，将会发生异常

问题出在将一个字符串复制进一个字符变量指针所指地址。虽然可以正确输出结果，但因为越界进行内在读写而导致程序崩溃。

```
char* s="AAA";
```

```
printf("%s",s);
s[0]='B';
printf("%s",s);
```
有什么错？
"AAA"是字符串常量。s 是指针，指向这个字符串常量，所以声明 s 的时候就有问题。
```
cosnt char* s="AAA";
```
然后又因为是常量，所以对是 s[0]的赋值操作是不合法的。

15、有以下表达式：
```
int a=248; b=4;int const c=21;const int *d=&a;
int *const e=&b;int const *f const =&a;
```
请问下列表达式哪些会被编译器禁止？为什么？
```
*c=32;d=&b;*d=43;e=34;e=&a;f=0x321f;
```
*c 这是个什么东东，禁止
*d 说了是 const， 禁止
e = &a 说了是 const 禁止
const *f const =&a; 禁止

16、交换两个变量的值，不使用第三个变量。
即 a=3,b=5,交换之后 a=5,b=3;
有两种解法，一种用算术算法，一种用^(异或)
```
a = a + b;
b = a - b;
a = a - b;
```
or
```
a = a^b;// 只能对 int,char..
b = a^b;
a = a^b;
```
or
```
a ^= b ^= a;
```
17、下面的程序会出现什么结果

```
.#include <stdio.h>
  #include <stdlib.h>
  void getmemory(char *p)
  {
    p=(char *) malloc(100);
    strcpy(p,"hello world");
  }
  int main( )
  {
    char *str=NULL;
    getmemory(str);
    printf("%s/n",str);
    free(str);
    return 0;
   }
```
程序崩溃，getmemory 中的 malloc 不能返回动态内存， free（）对 str 操作很危险

18、下面的语句会出现什么结果？

```
char szstr[10];
strcpy(szstr,"0123456789");
```

答案：长度不一样，会造成非法的 OS，应该改为 `char szstr[11];`

19、(void *)ptr 和 (*(void))ptr 的结果是否相同？**

答：其中 ptr 为同一个指针

.(void *)ptr 和 (*(void**))ptr 值是相同的

20、问函数既然不会被其它函数调用，为什么要返回 1？

```
int main()
{
 int x=3;
 printf("%d",x);
 return 1;
 }
```

答：mian 中，c 标准认为 0 表示成功，非 0 表示错误。具体的值是某中具体出错信息

21、对绝对地址 0x100000 赋值且想让程序跳转到绝对地址是 0x100000 去执行

```
(unsigned int*)0x100000 = 1234;
```

首先要将 0x100000 强制转换成函数指针,即:

```
 (void (*)())0x100000
```

然后再调用它:

```
*((void (*)())0x100000)();
```

用 typedef 可以看得更直观些:

```
typedef void(*)() voidFuncPtr;
*((voidFuncPtr)0x100000)();
```

22、输出多少？并分析过程

```
unsigned short A = 10;
printf("~A = %u\n", ~A);

char c=128;
printf("c=%d\n",c);
```

第一题，～A ＝0xfffffff5,int 值 为－11，但输出的是 uint。所以输出 4294967285

第二题，c＝0x10,输出的是 int，最高位为 1，是负数，所以它的值就是 0x00 的补码就是 128，所以输出－128。

这两道题都是在考察二进制向 int 或 uint 转换时的最高位处理。

23、分析下面的程序：

```
void GetMemory(char **p,int num)
{
    *p=(char *)malloc(num);
}
int main()
{
    char *str=NULL;
    GetMemory(&str,100);
    strcpy(str,"hello");
    free(str);
```

```
    if(str!=NULL)
    {
        strcpy(str,"world");
    }
    printf("\n str is %s",str);
    getchar();
}
```

问输出结果是什么？希望大家能说说原因，先谢谢了

输出 str is world。

free 只是释放的 str 指向的内存空间,它本身的值还是存在的.

所以 free 之后，有一个好的习惯就是将 str=NULL.

此时 str 指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出 world 来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。

当你 malloc 一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的，只不过。。。。。。。楼上都说过了，最好别这么干。

24、char a[10],strlen(a)为什么等于 15？运行的结果

```
    #include "stdio.h"
    #include "string.h"

    void main()
    {
        char aa[10];
        printf("%d",strlen(aa));
    }
```

sizeof()和初不初始化，没有关系；

strlen()和初始化有关。

```
    char (*str)[20];/*str 是一个数组指针，即指向数组的指针．*/
    char *str[20];/*str 是一个指针数组，其元素为指针型数据．*/
```

25、long a=0x801010;a+5=?

答：0x801010 用二进制表示为：“1000 0000 0001 0000 0001 0000”，十进制的值为 8392720，再加上 5 就是 8392725

26、给定结构 struct A

```
{
      char t:: 4;
      char k:4;
      unsigned short i:8;
      unsigned long m;
};
```

问 sizeof(A) = ?

给定结构 struct A

```
{
      char t:4; 4 位
      char k:4; 4 位
```

```
    unsigned short i:8; 8 位
    unsigned long m; // 偏移 2 字节保证 4 字节对齐
}; // 共 8 字节
```

27、下面的函数实现在一个数上加一个数，有什么错误？请改正。

```
int add_n ( int n )
{
    static int i = 100;
    i += n;
    return i;
}
```

当你第二次调用时得不到正确的结果，难道你写个函数就是为了调用一次？问题就出在 static 上

28、给出下面程序的答案

```
#include<iostream.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
typedef struct  AA
{
       int b1:5;
       int b2:2;
}AA;
void main()
{
       AA aa;
       char cc[100];
        strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
      memcpy(&aa,cc,sizeof(AA));
       cout << aa.b1 <<endl;
       cout << aa.b2 <<endl;
}
```

答案是 -16 和 1

首先 sizeof(AA)的大小为 4,b1 和 b2 分别占 5bit 和 2bit.

经过 strcpy 和 memcpy 后,aa 的 4 个字节所存放的值是:

0,1,2,3 的 ASC 码，即 00110000,00110001,00110010,00110011

所以，最后一步：显示的是这 4 个字节的前 5 位，和之后的 2 位

分别为：10000,和 01

因为 int 是有正负之分　　所以：答案是-16 和 1

29、求函数返回值，输入 x=9999;

```
int func （ x ）
{
    int countx = 0;
    while ( x )
    {
        countx ++;
        x = x&(x-1);
    }
```

```
    return countx;
}
```
结果呢？
知道了这是统计 9999 的二进制数值中有多少个 1 的函数，且有
9999＝9×1024＋512＋256＋15

9×1024 中含有 1 的个数为 2；
512 中含有 1 的个数为 1；
256 中含有 1 的个数为 1；
15 中含有 1 的个数为 4；
故共有 1 的个数为 8，结果为 8。
1000 - 1 = 0111，正好是原数取反。这就是原理。
用这种方法来求 1 的个数是很效率很高的。
不必去一个一个地移位。循环次数最少。

30、分析：

```
struct bit
{   int a:3;
    int  b:2;
    int c:3;
};
int main()
{
  bit s;
  char *c=(char*)&s;
   cout<<sizeof(bit)<<endl;
  *c=0x99;
   cout << s.a <<endl <<s.b<<endl<<s.c<<endl;
     int a=-1;
   printf("%x",a);
  return 0;
}
```
输出为什么是
4
1
-1
-4
ffffffff
因为 0x99 在内存中表示为 100 11 001 ，a = 001，b = 11，c = 100
当 c 为有符合数时，c = 100，最高 1 为表示 c 为负数，负数在计算机用补码表示，所以 c = -4;同理 b = -1;
当 c 为有符合数时，c = 100,即 c = 4,同理 b = 3

31、下面这个程序执行后会有什么错误或者效果：

```
 #define MAX 255
 int main()
{
   unsigned char A[MAX],i;//i 被定义为 unsigned char
   for (i=0;i<=MAX;i++)
      A[i]=i;
```

```
}
```
解答：死循环加数组越界访问（C/C++不进行数组越界检查）
MAX=255
数组 A 的下标范围为:0..MAX-1,这是其一..
其二.当 i 循环到 255 时,循环内执行:
```
  A[255]=255;
```
这句本身没有问题..但是返回 for (i=0;i<=MAX;i++)语句时,
由于 unsigned char 的取值范围在(0..255),i++以后 i 又为 0 了..无限循环下去.

32、写出 sizeof(struct name1)=,sizeof(struct name2)=的结果
```
struct name1{
   char  str;
   short x;
   int   num;
}

struct name2{
   char str;
   int num;
   short x;
}
```

sizeof(struct name1)=8,sizeof(struct name2)=12
在第二个结构中，为保证 num 按四个字节对齐，char 后必须留出 3 字节的空间；同时为保证整个结构的自然对齐（这里是 4 字节对齐），在 x 后还要补齐 2 个字节，这样就是 12 字节。

33、struct s1
```
{
  int i: 8;
  int j: 4;
  int a: 3;
  double b;
};

struct s2
{
  int i: 8;
  int j: 4;
  double b;
  int a:3;
};

printf("sizeof(s1)= %d\n", sizeof(s1));
printf("sizeof(s2)= %d\n", sizeof(s2));
```
result: 16, 24
第一个 struct s1
```
{
  int i: 8;
  int j: 4;
  int a: 3;
```

```
  double b;
};
```
理论上是这样的，首先是 i 在相对 0 的位置，占 8 位一个字节，然后，j 就在相对一个字节的位置，由于一个位置的字节数是 4 位的倍数，因此不用对齐，就放在那里了，然后是 a，要在 3 位的倍数关系的位置上，因此要移一位，在 15 位的位置上放下，目前总共是 18 位，折算过来是 2 字节 2 位的样子，由于 double 是 8 字节的，因此要在相对 0 要是 8 个字节的位置上放下，因此从 18 位开始到 8 个字节之间的位置被忽略，直接放在 8 字节的位置了，因此，总共是 16 字节。
第二个最后会对照是不是结构体内最大数据的倍数，不是的话，会补成是最大数据的倍数

34、在对齐为 4 的情况下

```
struct BBB
{
   long num;
   char *name;
   short int data;
   char ha;
   short ba[5];
}*p;
p=0x1000000;
p+0x200=____;
(Ulong)p+0x200=____;
(char*)p+0x200=____;
```
希望各位达人给出答案和原因，谢谢拉
解答：假设在 32 位 CPU 上，

```
sizeof(long) = 4 bytes
sizeof(char *) = 4 bytes
sizeof(short int) = sizeof(short) = 2 bytes
sizeof(char) = 1 bytes
```

由于是 4 字节对齐，

```
sizeof(struct BBB) = sizeof(*p)
= 4 + 4 + 2 + 1 + 1/*补齐*/ + 2*5 + 2/*补齐*/ = 24 bytes  (经 Dev-C++验证)

p=0x1000000;
p+0x200=____;
    = 0x1000000 + 0x200*24

(Ulong)p+0x200=____;
    = 0x1000000 + 0x200

(char*)p+0x200=____;
    = 0x1000000 + 0x200*4
```

35、找错

```
Void test1()
{
char string[10];
char* str1="0123456789";
strcpy(string, str1);// 溢出，应该包括一个存放'\0'的字符 string[11]
}
```

```
Void test2()
{
char string[10], str1[10];
for(I=0; I<10;I++)
{
str1[i] ='a';
}
strcpy(string, str1);// I，i 没有声明。
}

Void test3(char* str1)
{
char string[10];
if(strlen(str1)<=10)// 改成<10,字符溢出，将 strlen 改为 sizeof 也可以
{
strcpy(string, str1);
}
}
```

36、写出输出结果

```
void g(int**);
int main()
{
int line[10],i;
int *p=line; //p 是地址的地址
for (i=0;i<10;i++)
{
*p=i;
g(&p);//数组对应的值加 1
}
for(i=0;i<10;i++)
printf("%d\n",line[i]);
return 0;
}

void g(int**p)
{
(**p)++;
(*p)++;// 无效
}
```

输出：

```
1
2
3
4
5
6
7
8
```

9
10

37、写出程序运行结果

```
int sum(int a)
{
auto int c=0;
static int b=3;
c+=1;
b+=2;
return(a+b+c);
}

void main()
{
int I;
int a=2;
for(I=0;I<5;I++)
{
printf("%d,", sum(a));
}
}
```

// static 会保存上次结果，记住这一点，剩下的自己写

输出：8,10,12,14,16,

38、评价代码

```
int func(int a)
{
int b;
switch(a)
{
case 1: 30;
case 2: 20;
case 3: 16;
default: 0
}
return b;
}
```

则 func(1)=?

// b 定义后就没有赋值

```
int a[3];
a[0]=0; a[1]=1; a[2]=2;
int *p, *q;
p=a;
q=&a[2];
```

则 a[q-p]=a[2]

解释：指针一次移动一个 int 但计数为 1

39、请问一下程序将输出什么结果？

```
char *RetMenory(void)
{
       char p[] = “hellow world”;
       return p;
}
void Test(void)
{
       char *str = NULL;
       str = RetMemory();
       printf(str);
}
```

RetMenory 执行完毕，p 资源被回收，指向未知地址。返回地址，str 的内容应是不可预测的，打印的应该是 str 的地址

40、写出输出结果

```
typedef struct
  {
     int a:2;
     int b:2;
     int c:1;
  }test;

  test t;
  t.a = 1;
  t.b = 3;
  t.c = 1;

  printf("%d",t.a);
  printf("%d",t.b);
  printf("%d",t.c);
```

t.a 为 01,输出就是 1
t.b 为 11，输出就是－1
t.c 为 1，输出也是-1
3 个都是有符号数 int 嘛。
这是位扩展问题
01
11
1
编译器进行符号扩展

41、对下面程序进行分析

```
void test2()
{
   char string[10], str1[10];
   int i;
   for(i=0; i<10; i++)
   {
      str1[i] = 'a';
   }
```

```
   strcpy( string, str1 );
}
```
解答:如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分;如果面试者指出 strcpy(string, str1)调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分,在此基础上指出库函数 strcpy 工作方式的给 10 分;
str1 不能在数组内结束:因为 str1 的存储为:{a,a,a,a,a,a,a,a,a,a},没有'\0'(字符串结束符),所以不能结束
strcpy(char *s1,char *s2)他的工作原理是,扫描 s2 指向的内存,逐个字符付到 s1 所指向的内存,直到碰到'\0',因为 str1 结尾没有'\0',所以具有不确定性,不知道他后面还会付什么东东。
正确应如下
```
void test2()
{
   char string[10], str1[10];
   int i;
   for(i=0; i<9; i++)
   {
      str1[i] = 'a'+i; //把 abcdefghi 赋值给字符数组
   }
   str[i]='\0';//加上结束符
   strcpy( string, str1 );
}
```

42、分析:
```
int arr[] = {6,7,8,9,10};
int *ptr = arr;
*(ptr++)+=123;
printf(" %d %d ", *ptr, *(++ptr));
```
输出:8 8
过程:对于*(ptr++)+=123;先做加法 6+123,然后++,指针指向 7;对于 printf(" %d %d ", *ptr, *(++ptr));从后往前执行,指针先++,指向 8,然后输出 8,紧接着再输出 8

43、分析下面的代码:
```
char *a = "hello";
char *b = "hello";
if(a= =b)
printf("YES");
else
printf("NO");
```
这个简单的面试题目,我选输出 no(对比的应该是指针地址吧),可在 VC 是 YES 在 C 是 NO
lz 的呢,是一个常量字符串。位于静态存储区,它在程序生命期内恒定不变。如果编译器优化的话,会有可能 a 和 b 同时指向同一个 hello 的。则地址相同。如果编译器没有优化,那么就是两个不同的地址,则不同

44、写出输出结果
```
#include <stdio.h>
void foo(int m, int n)
{
    printf("m=%d, n=%d\n", m, n);
}

int main()
```

```
{
    int b = 3;
    foo(b+=3, ++b);
    printf("b=%d\n", b);
return 0;
}
```
输出：m=7,n=4,b=7(VC6.0)
这种方式和编译器中得函数调用关系相关即先后入栈顺序。不过不同编译器得处理不同。也是因为 C 标准中对这种方式说明为未定义，所以各个编译器厂商都有自己得理解，所以最后产生得结果完全不同。
因为这样，所以遇见这种函数，我们首先要考虑我们得编译器会如何处理这样得函数，其次看函数得调用方式，不同得调用方式，可能产生不同得结果。最后是看编译器优化。

45、找出错误

```
#include   string.h
main(void)
{   char   *src="hello,world";
    char   *dest=NULL;
    dest=(char   *)malloc(strlen(src));
    int   len=strlen(str);
    char   *d=dest;
    char   *s=src[len];
    while(len--!=0)
      d++=s--;
    printf("%s",dest);
}
```
找出错误!!
```
#include   "string.h"
#include "stdio.h"
#include "malloc.h"
main(void)
{
char   *src="hello,world";
    char   *dest=NULL;
    dest=(char   *)malloc(sizeof(char)*(strlen(src)+1));
    int   len=strlen(src);
    char   *d=dest;
    char   *s=src+len-1;
    while(len--!=0)
      *d++=*s--;
*d='\0';
    printf("%s",dest);
}
```

第三部分：编程题

1、读文件 file1.txt 的内容（例如）：

12

34
56
输出到 file2.txt：
56
34
12

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

int main(void)
{
        int MAX = 10;
int *a = (int *)malloc(MAX * sizeof(int));
int *b;

FILE *fp1;
FILE *fp2;

fp1 = fopen("a.txt","r");
if(fp1 == NULL)
{printf("error1");
    exit(-1);
}

    fp2 = fopen("b.txt","w");
if(fp2 == NULL)
{printf("error2");
    exit(-1);
}

int i = 0;
    int j = 0;

while(fscanf(fp1,"%d",&a[i]) != EOF)
{
i++;
j++;
if(i >= MAX)
{
MAX = 2 * MAX;
b = (int*)realloc(a,MAX * sizeof(int));
if(b == NULL)
{
printf("error3");
exit(-1);
}
a = b;
}
}
```

```
for(;--j >= 0;)
   fprintf(fp2,"%d\n",a[j]);

fclose(fp1);
fclose(fp2);

return 0;


}
```

2、输出和为一个给定整数的所有组合

例如 n=5

5=1+4；5=2+3（相加的数不能重复）

则输出

1，4；2，3。

```
#include <stdio.h>

int main(void)
{
unsigned long int i,j,k;

printf("please input the number\n");
scanf("%d",&i);
    if( i % 2 == 0)
        j = i / 2;
else
j = i / 2 + 1;

printf("The result is \n");
    for(k = 0; k < j; k++)
     printf("%d = %d + %d\n",i,k,i - k);
return 0;
}

#include <stdio.h>
void main()
{
unsigned long int a,i=1;
scanf("%d",&a);
if(a%2==0)
{
     for(i=1;i<a/2;i++)
     printf("%d",a,a-i);
}
else
for(i=1;i<=a/2;i++)
        printf(" %d, %d",i,a-i);
}
```

3、递规反向输出字符串的例子,可谓是反序的经典例程.

```
void inverse(char *p)
{
    if( *p = = '\0' )
return;
    inverse( p+1 );
    printf( "%c", *p );
}

int main(int argc, char *argv[])
{
    inverse("abc\0");

    return 0;
}
```

对 1 的另一种做法：

```
#include <stdio.h>
void test(FILE *fread, FILE *fwrite)
{
        char buf[1024] = {0};
        if (!fgets(buf, sizeof(buf), fread))
                return;
        test( fread, fwrite );
        fputs(buf, fwrite);
}
int main(int argc, char *argv[])
{
        FILE *fr = NULL;
        FILE *fw = NULL;
        fr = fopen("data", "rb");
        fw = fopen("dataout", "wb");
        test(fr, fw);
        fclose(fr);
        fclose(fw);
        return 0;
}
```

4、写一段程序，找出数组中第 k 大小的数，输出数所在的位置。例如{2，4，3，4，7}中，第一大的数是 7，位置在 4。第二大、第三大的数都是 4，位置在 1、3 随便输出哪一个均可。函数接口为：int find_orderk(const int* narry,const int n,const int k)

要求算法复杂度不能是 O(n^2）

谢谢！

可以先用快速排序进行排序，其中用另外一个进行地址查找

代码如下，在 VC++6.0 运行通过。给分吧^-^

```
//快速排序

#include<iostream>

usingnamespacestd;
```

```
intPartition (int*L,intlow,int high)
{
inttemp = L[low];
intpt = L[low];

while (low < high)
{
while (low < high && L[high] >= pt)
--high;
L[low] = L[high];
while (low < high && L[low] <= pt)
++low;
L[low] = temp;
}
L[low] = temp;

returnlow;
}

voidQSort (int*L,intlow,int high)
{
if (low < high)
{
intpl = Partition (L,low,high);

QSort (L,low,pl - 1);
QSort (L,pl + 1,high);
}
}

intmain ()
{
intnarry[100],addr[100];
intsum = 1,t;

cout << "Input number:" << endl;
cin >> t;

while (t != -1)
{
narry[sum] = t;
addr[sum - 1] = t;
sum++;

cin >> t;
}

sum -= 1;
QSort (narry,1,sum);
```

```
for (int i = 1; i <= sum;i++)
cout << narry[i] << '\t';
cout << endl;

intk;
cout << "Please input place you want:" << endl;
cin >> k;

intaa = 1;
intkk = 0;
for (;;)
{
if (aa == k)
break;
if (narry[kk] != narry[kk + 1])
{
aa += 1;
kk++;
}

}

cout << "The NO." << k << "number is:" << narry[sum - kk] << endl;
cout << "And it's place is:" ;
for (i = 0;i < sum;i++)
{
if (addr[i] == narry[sum - kk])
cout << i << '\t';
}


return0;
}
```

5、两路归并排序

```
Linklist *unio(Linklist *p,Linklist *q){
linklist *R,*pa,*qa,*ra;
pa=p;
qa=q;
R=ra=p;
while(pa->next!=NULL&&qa->next!=NULL){
if(pa->data>qa->data){
ra->next=qa;
qa=qa->next;
}
else{
ra->next=pa;
pa=pa->next;
}
```

```
}
if(pa->next!=NULL)
ra->next=pa;
if(qa->next!=NULL)
ra->next==qa;
return R;
}
```

6、用递归算法判断数组 a[N]是否为一个递增数组。

递归的方法，记录当前最大的，并且判断当前的是否比这个还大，大则继续，否则返回 false 结束：

```
bool fun( int a[], int n )
{
if( n= =1 )
return true;
if( n= =2 )
return a[n-1] >= a[n-2];
return fun( a,n-1) && ( a[n-1] >= a[n-2] );
}
```

7、单连表的建立，把'a'--'z'26 个字母插入到连表中，并且倒叙，还要打印！

方法 1：

```
typedef struct val
{   int date_1;
    struct val *next;
}*p;

void main(void)
{   char c;

    for(c=122;c>=97;c--)
       { p.date=c;
         p=p->next;
        }

    p.next=NULL;
}
}
```

方法 2：

```
node *p = NULL;
node *q = NULL;

node *head = (node*)malloc(sizeof(node));
head->data = ' ';head->next=NULL;

node *first = (node*)malloc(sizeof(node));
first->data = 'a';first->next=NULL;head->next = first;
p = first;

int longth = 'z' - 'b';
int i=0;
```

```
while ( i<=longth )
{
node *temp = (node*)malloc(sizeof(node));
temp->data = 'b'+i;temp->next=NULL;q=temp;

head->next = temp; temp->next=p;p=q;
i++;
}

print(head);
```

8、请列举一个软件中时间换空间或者空间换时间的例子。

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

--->空优

```
void swap(int a,int b)
{
a=a+b;b=a-b;a=a-b;
}
```

9、outputstr 所指的值为 123456789

```
int continumax(char *outputstr, char *inputstr)
{
char *in = inputstr, *out = outputstr, *temp, *final;
int count = 0, maxlen = 0;

while( *in != '\0' )
{
if( *in > 47 && *in < 58 )
{
for(temp = in; *in > 47 && *in < 58 ; in++ )
count++;
}
else
in++;

if( maxlen < count )
{
maxlen = count;
count = 0;
final = temp;
}
}
for(int i = 0; i < maxlen; i++)
{
*out = *final;
out++;
final++;
```

```
}
*out = '\0';
return maxlen;
}
```

10、不用库函数,用 C 语言实现将一整型数字转化为字符串

方法 1:

```
int getlen(char *s){
    int n;
    for(n = 0; *s != '\0'; s++)
          n++;
    return n;
}
void reverse(char s[])
{
   int c,i,j;
   for(i = 0,j = getlen(s) - 1; i < j; i++,j--){
       c = s[i];
       s[i] = s[j];
       s[j] = c;
   }
}
void itoa(int n,char s[])
{
   int i,sign;
   if((sign = n) < 0)
        n = -n;
   i = 0;
   do{/*以反序生成数字*/
      s[i++] = n%10 + '0';/*get next number*/
   }while((n /= 10) > 0);/*delete the number*/

   if(sign < 0)
      s[i++] = '-';

   s[i] = '\0';
   reverse(s);
}
```

方法 2:

```
#include <iostream>
using namespace std;

void itochar(int num);

void itochar(int num)
{
int i = 0;
int j ;
char stra[10];
char strb[10];
```

```
while ( num )
{
stra[i++]=num%10+48;
num=num/10;
}
stra[i] = '\0';
for( j=0; j < i; j++)
{
strb[j] = stra[i-j-1];
}
strb[j] = '\0';
cout<<strb<<endl;

}
int main()
{
int num;
cin>>num;
itochar(num);
return 0;
}
```

11、求组合数： 求 n 个数（1....n）中 k 个数的组合....

如：combination(5,3)

要求输出：543，542，541，532，531，521，432，431，421，321，

```
#include<stdio.h>

int pop(int *);
int push(int );
void combination(int ,int );

int stack[3]={0};
top=-1;

int main()
{
int n,m;
printf("Input two numbers:\n");
while( (2!=scanf("%d%*c%d",&n,&m)) )
{
fflush(stdin);
printf("Input error! Again:\n");
}
combination(n,m);
printf("\n");
}
void combination(int m,int n)
{
int temp=m;
push(temp);
```

```
while(1)
{
if(1==temp)
{
if(pop(&temp)&&stack[0]==n) //当栈底元素弹出&&为可能取的最小值，循环退出
break;
}
else if( push(--temp))
{
printf("%d%d%d  ",stack[0],stack[1],stack[2]);//§&auml;¨ì¤@?
pop(&temp);
}
}
}
int push(int i)
{
stack[++top]=i;
if(top<2)
return 0;
else
return 1;
}
int pop(int *i)
{
*i=stack[top--];
if(top>=0)
return 0;
else
return 1;
}
```

12、用指针的方法，将字符串“ABCD1234efgh”前后对调显示

```
#include <stdio.h>
#include <string.h>
#include <dos.h>
int main()
{
    char str[] = "ABCD1234efgh";
    int length = strlen(str);
    char * p1 = str;
    char * p2 = str + length - 1;
    while(p1 < p2)
    {
        char c = *p1;
        *p1 = *p2;
        *p2 = c;
        ++p1;
        --p2;
    }
    printf("str now is %s\n",str);
```

```
    system("pause");
    return 0;
}
```

13、有一分数序列：1/2,1/4,1/6,1/8……，用函数调用的方法，求此数列前 20 项的和

```
#include <stdio.h>
double getValue()
{
    double result = 0;
    int i = 2;
    while(i < 42)
    {
        result += 1.0 / i;//一定要使用 1.0 做除数，不能用 1，否则结果将自动转化成整数，即 0.000000
        i += 2;
    }
    return result;
}
int main()
{
    printf("result is %f\n", getValue());
    system("pause");
    return 0;
}
```

14、有一个数组 a[1000]存放 0--1000;要求每隔二个数删掉一个数，到末尾时循环至开头继续进行，求最后一个被删掉的数的原始下标位置。

以 7 个数为例：

{0,1,2,3,4,5,6,7} 0-->1-->2（删除）-->3-->4-->5(删除)-->6-->7-->0（删除），如此循环直到最后一个数被删除。

方法 1：数组

```
#include <iostream>
using namespace std;
#define null 1000

int main()
{
int arr[1000];
for (int i=0;i<1000;++i)
arr[i]=i;
int j=0;
int count=0;
while(count<999)
{
while(arr[j%1000]==null)
j=(++j)%1000;
j=(++j)%1000;
while(arr[j%1000]==null)
j=(++j)%1000;
j=(++j)%1000;
while(arr[j%1000]==null)
```

```
j=(++j)%1000;
arr[j]=null;
++count;
}
while(arr[j]==null)
j=(++j)%1000;

cout<<j<<endl;
return 0;
}
```
方法 2：链表
```
#include<iostream>
using namespace std;
#define null 0
struct node
{
int data;
node* next;
};
int main()
{
node* head=new node;
head->data=0;
head->next=null;
node* p=head;
for(int i=1;i<1000;i++)
{
node* tmp=new node;
tmp->data=i;
tmp->next=null;
head->next=tmp;
head=head->next;
}
head->next=p;
while(p!=p->next)
{
p->next->next=p->next->next->next;
p=p->next->next;
}
cout<<p->data;
return 0;
}
```
方法 3：通用算法
```
#include <stdio.h>
#define MAXLINE 1000   //元素个数
/*
MAXLINE   元素个数
a[]       元素数组
R[]       指针场
suffix    下标
index     返回最后的下标序号
```

```
values    返回最后的下标对应的值
start     从第几个开始
K         间隔
*/
int find_n(int a[],int R[],int K,int& index,int& values,int s=0) {
   int suffix;
   int front_node,current_node;
   suffix=0;
      if(s==0) {
      current_node=0;
      front_node=MAXLINE-1;
      }
      else {
      current_node=s;
      front_node=s-1;
      }
        while(R[front_node]!=front_node) {
            printf("%d\n",a[current_node]);
            R[front_node]=R[current_node];
            if(K==1) {
              current_node=R[front_node];
              continue;
            }
            for(int i=0;i<K;i++){
               front_node=R[front_node];
            }
            current_node=R[front_node];
        }
 index=front_node;
 values=a[front_node];

 return 0;
}
int main(void) {
int a[MAXLINE],R[MAXLINE],suffix,index,values,start,i,K;
suffix=index=values=start=0;
K=2;

for(i=0;i<MAXLINE;i++) {
a[i]=i;
R[i]=i+1;
}
R[i-1]=0;
find_n(a,R,K,index,values,2);
printf("the value is %d,%d\n",index,values);
return 0;
}
```

15、实现 strcmp

```
int StrCmp(const char *str1, const char *str2)
```

做是做对了，没有抄搞，比较乱

```
int StrCmp(const char *str1, const char *str2)
{
    assert(str1 && srt2);
    while (*str1 && *str2 && *str1 == *str2) {
        str1++, str2++;
    }
    if (*str1 && *str2)
        return (*str1-*str2);
    elseif (*str1 && *str2==0)
        return 1;
    elseif (*str1 = = 0 && *str2)
        return -1;
    else
        return 0;
}

int StrCmp(const char *str1, const char *str2)
{
        //省略判断空指针(自己保证)
while(*str1 && *str1++ = = *str2++);
return *str1-*str2;
}
```

16、实现子串定位

```
int FindSubStr(const char *MainStr, const char *SubStr)
```

做是做对了，没有抄搞，比较乱

```
int MyStrstr(const char* MainStr, const char* SubStr)
{
const char *p;
const char *q;
const char * u = MainStr;

//assert((MainStr!=NULL)&&( SubStr!=NULL));//用断言对输入进行判断
while(*MainStr) //内部进行递增
{
p = MainStr;
q = SubStr;
while(*q && *p && *p++ == *q++);
if(!*q )
{
return MainStr - u +1 ;//MainStr 指向当前起始位，u 指向
}
MainStr ++;
}
return -1;
}
```

17、已知一个单向链表的头，请写出删除其某一个结点的算法，要求，先找到此结点，然后删除。

```
slnodetype *Delete(slnodetype *Head,int key){}中 if(Head->number==key)
```

```
{
Head=Pointer->next;
free(Pointer);
break;
}
Back = Pointer;
       Pointer=Pointer->next;
if(Pointer->number==key)
{
           Back->next=Pointer->next;
free(Pointer);
break;
}
void delete(Node* p)
{
    if(Head = Node)

    while(p)
}
```

18、有 1,2,....一直到 n 的无序数组,求排序算法,并且要求时间复杂度为 O(n),空间复杂度 O(1),使用交换,而且一次只能交换两个数.（华为）

```
#include<iostream.h>

int main()
{
    int a[]  = {10,6,9,5,2,8,4,7,1,3};
    int len = sizeof(a) / sizeof(int);
    int temp;

    for(int i = 0; i < len; )
    {
temp = a[a[i] - 1];
a[a[i] - 1] = a[i];
a[i] = temp;

if ( a[i] == i + 1)
  i++;
    }
    for (int j = 0; j < len; j++)
      cout<<a[j]<<",";

    return 0;
}
```

19、写出程序把一个链表中的接点顺序倒排

```
typedef struct linknode
{
int data;
struct linknode *next;
```

```
}node;
//将一个链表逆置
node *reverse(node *head)
{
node *p,*q,*r;
p=head;
q=p->next;
while(q!=NULL)
{
r=q->next;
q->next=p;
p=q;
q=r;
}

head->next=NULL;
head=p;
return head;
}
```

20、写出程序删除链表中的所有接点

```
void del_all(node *head)
{
node *p;
while(head!=NULL)
{
p=head->next;
free(head);
head=p;
}
cout<<"释放空间成功!"<<endl;
}
```

21、两个字符串，s,t;把 t 字符串插入到 s 字符串中，s 字符串有足够的空间存放 t 字符串

```
void insert(char *s, char *t, int i)
{
char *q = t;
char *p =s;
if(q == NULL)return;
while(*p!='\0')
{
p++;
}
while(*q!=0)
{
*p=*q;
p++;
q++;
}
*p = '\0';
```

```
}
```

22、写一个函数，功能：完成内存之间的拷贝

memcpy source code:

```
270 void* memcpy( void *dst, const void *src, unsigned int len )
271 {
272    register char *d;
273    register char *s;
27
275    if (len == 0)
276       return dst;
277
278    if (is_overlap(dst, src, len, len))
279       complain3("memcpy", dst, src, len);
280
281    if ( dst > src ) {
282       d = (char *)dst + len - 1;
283       s = (char *)src + len - 1;
284       while ( len >= 4 ) {
285          *d-- = *s--;
286          *d-- = *s--;
287          *d-- = *s--;
288          *d-- = *s--;
289          len -= 4;
290       }
291       while ( len-- ) {
292          *d-- = *s--;
293       }
294    } else if ( dst < src ) {
295       d = (char *)dst;
296       s = (char *)src;
297       while ( len >= 4 ) {
298          *d++ = *s++;
299          *d++ = *s++;
300          *d++ = *s++;
301          *d++ = *s++;
302          len -= 4;
303       }
304       while ( len-- ) {
305          *d++ = *s++;
306       }
307    }
308    return dst;
309 }
```

23、公司考试这种题目主要考你编写的代码是否考虑到各种情况，是否安全（不会溢出）

各种情况包括：

1、参数是指针，检查指针是否有效

2、检查复制的源目标和目的地是否为同一个，若为同一个，则直接跳出

3、读写权限检查

４、安全检查，是否会溢出
memcpy 拷贝一块内存，内存的大小你告诉它
strcpy 是字符串拷贝，遇到'\0'结束

```
/* memcpy ——拷贝不重叠的内存块 */
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
void* pbFrom = (byte*)pvFrom;
ASSERT(pvTo != NULL && pvFrom != NULL); //检查输入指针的有效性
ASSERT(pbTo>=pbFrom+size || pbFrom>=pbTo+size);//检查两个指针指向的内存是否重叠
while(size-->0)
*pbTo++ == *pbFrom++;
return(pvTo);
}
```

24、两个字符串，s,t;把 t 字符串插入到 s 字符串中，s 字符串有足够的空间存放 t 字符串

```
void insert(char *s, char *t, int i)
{
memcpy(&s[strlen(t)+i],&s[i],strlen(s)-i);
memcpy(&s[i],t,strlen(t));
s[strlen(s)+strlen(t)]='\0';
}
```

25、编写一个 C 函数，该函数在一个字符串中找到可能的最长的子字符串，且该字符串是由同一字符组成的。

```
char * search(char *cpSource, char ch)
{
        char *cpTemp=NULL, *cpDest=NULL;
        int iTemp, iCount=0;
        while(*cpSource)
        {
                if(*cpSource == ch)
                {
                        iTemp = 0;
                        cpTemp = cpSource;
                        while(*cpSource == ch)
++iTemp, ++cpSource;
                        if(iTemp > iCount)
iCount = iTemp, cpDest = cpTemp;
       if(!*cpSource)
break;
                }
                ++cpSource;
 }
 return cpDest;
}
```

26、请编写一个 C 函数，该函数在给定的内存区域搜索给定的字符，并返回该字符所在位置索引值。

```
int search(char *cpSource, int n, char ch)
```

```
{
        int i;
        for(i=0; i<n && *(cpSource+i) != ch; ++i);
        return i;
}
```

27、给定字符串 A 和 B,输出 A 和 B 中的最大公共子串。

比如 A="aocdfe" B="pmcdfa" 则输出"cdf"

```
*/
//Author: azhen
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

char *commanstring(char shortstring[], char longstring[])
{
int i, j;

char *substring=malloc(256);

if(strstr(longstring, shortstring)!=NULL)              //如果……，那么返回 shortstring
return shortstring;

for(i=strlen(shortstring)-1;i>0; i--)                 //否则，开始循环计算
{
for(j=0; j<=strlen(shortstring)-i; j++){
memcpy(substring, &shortstring[j], i);
substring[i]='\0';
if(strstr(longstring, substring)!=NULL)
return substring;
}
}
return NULL;
}


main()
{
char *str1=malloc(256);
char *str2=malloc(256);
char *comman=NULL;

gets(str1);
gets(str2);

if(strlen(str1)>strlen(str2))                         //将短的字符串放前面
comman=commanstring(str2, str1);
else
comman=commanstring(str1, str2);
```

```
printf("the longest comman string is: %s\n", comman);
}
```

28、写一个函数比较两个字符串 str1 和 str2 的大小，若相等返回 0，若 str1 大于
str2 返回 1，若 str1 小于 str2 返回－1

```
int strcmp ( const char * src,const char * dst)
{
	int ret = 0 ;
	while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
{
		++src;
++dst;
}
	if ( ret < 0 )
		ret = -1 ;
	else if ( ret > 0 )
		ret = 1 ;
	return( ret );
}
```

29、求 1000！的未尾有几个 0（用素数相乘的方法来做，如 72=2*2*2*3*3）；
求出 1->1000 里,能被 5 整除的数的个数 n1,能被 25 整除的数的个数 n2,能被 125 整除的数的个数 n3,能被 625 整除的数的个数 n4.
1000!末尾的零的个数=n1+n2+n3+n4;

```
#include<stdio.h>
#define NUM 1000

int find5(int num){
int ret=0;
while(num%5==0){
num/=5;
ret++;
}
return ret;
}
int main(){
int result=0;
int i;
for(i=5;i<=NUM;i+=5)
{
result+=find5(i);
}
printf(" the total zero number is %d\n",result);
return 0;
}
```

30、有双向循环链表结点定义为：

```
struct node
{ int data;
struct node *front,*next;
};
```

有两个双向循环链表 A，B，知道其头指针为：pHeadA,pHeadB，请写一函数将两链表中 data 值相同的结点删除

```
BOOL DeteleNode(Node *pHeader, DataType Value)
{
if (pHeader == NULL) return;

BOOL bRet = FALSE;
Node *pNode = pHead;
while (pNode != NULL)
{
if (pNode->data == Value)
{
if (pNode->front == NULL)
{
pHeader = pNode->next;
pHeader->front = NULL;
}
else
{
if (pNode->next != NULL)
{
pNode->next->front = pNode->front;
}
pNode->front->next = pNode->next;
}

Node *pNextNode = pNode->next;
delete pNode;
pNode = pNextNode;

bRet = TRUE;
//不要 break 或 return，删除所有
}
else
{
pNode = pNode->next;
}
}

return bRet;
}

void DE(Node *pHeadA, Node *pHeadB)
{
if (pHeadA == NULL || pHeadB == NULL)
{
return;
}

Node *pNode = pHeadA;
```

```
while (pNode != NULL)
{
if (DeteleNode(pHeadB, pNode->data))
{
if (pNode->front == NULL)
{
pHeadA = pNode->next;
pHeadA->front = NULL;
}
else
{
pNode->front->next = pNode->next;
if (pNode->next != NULL)
{
pNode->next->front = pNode->front;
}
}
Node *pNextNode = pNode->next;
delete pNode;
pNode = pNextNode;
}
else
{
pNode = pNode->next;
}
}
}
```

31、编程实现：找出两个字符串中最大公共子字符串,如"abccade","dgcadde"的最大子串为"cad"

```
int GetCommon(char *s1, char *s2, char **r1, char **r2)
{
int len1 = strlen(s1);
int len2 = strlen(s2);
int maxlen = 0;

for(int i = 0; i < len1; i++)
{
for(int j = 0; j < len2; j++)
{
if(s1[i] == s2[j])
{
int as = i, bs = j, count = 1;
while(as + 1 < len1 && bs + 1 < len2 && s1[++as] == s2[++bs])
count++;

if(count > maxlen)
{
maxlen = count;
*r1 = s1 + i;
*r2 = s2 + j;
```

```
}
}
}
}
```

32、编程实现：把十进制数(long 型)分别以二进制和十六进制形式输出，不能使用 printf 系列库函数

```
char* test3(long num) {
char* buffer = (char*)malloc(11);
buffer[0] = '0';
buffer[1] = 'x';
buffer[10] = '\0';

char* temp = buffer + 2;
for (int i=0; i < 8; i++) {
temp[i] = (char)(num<<4*i>>28);
temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
}
return buffer;
}
```

33、输入 N，打印 N*N 矩阵

比如 N = 3，打印：

```
1  2  3
8  9  4
7  6  5
```

N = 4，打印：

```
1   2   3   4
12  13  14  5
11  16  15  6
10  9   8   7
```

解答：

```
1 #define N 15
int s[N][N];
void main()
{
int k = 0, i = 0, j = 0;
int a = 1;
for( ; k < (N+1)/2; k++ )
{
while( j < N-k ) s[i][j++] = a++; i++; j--;
while( i < N-k ) s[i++][j] = a++; i--; j--;
while( j > k-1 ) s[i][j--] = a++; i--; j++;
while( i > k )   s[i--][j] = a++; i++; j++;
}
for( i = 0; i < N; i++ )
{
```

```
for( j = 0; j < N; j++ )
cout << s[i][j] << '\t';
cout << endl;
}
}
2 define MAX_N  100
int matrix[MAX_N][MAX_N];

/*
 *（x,y）：第一个元素的坐标
 * start：第一个元素的值
 * n：矩阵的大小
 */
void SetMatrix(int x, int y, int start, int n) {
    int i, j;

    if (n <= 0)    //递归结束条件
        return;
    if (n == 1) {  //矩阵大小为 1 时
        matrix[x][y] = start;
        return;
    }
    for (i = x; i < x + n-1; i++)   //矩阵上部
        matrix[y][i] = start++;

    for (j = y; j < y + n-1; j++)   //右部
        matrix[j][x+n-1] = start++;

    for (i = x+n-1; i > x; i--)     //底部
        matrix[y+n-1][i] = start++;

    for (j = y+n-1; j > y; j--)     //左部
        matrix[j][x] = start++;

    SetMatrix(x+1, y+1, start, n-2);   //递归
}

void main() {
   int i, j;
   int n;

   scanf("%d", &n);
   SetMatrix(0, 0, 1, n);

   //打印螺旋矩阵
   for(i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
printf("%4d", matrix[i][j]);
      printf("\n");
   }
```

```
}
```

34、斐波拉契数列递归实现的方法如下：

```
 int  Funct( int n )
{
   if(n==0) return 1;
   if(n==1) return 1;
   retrurn  Funct(n-1) + Funct(n-2);
}
```

请问，如何不使用递归，来实现上述函数？

请教各位高手！

解答：

```
int  Funct( int n )  //  n 为非负整数
{
   int a=0;
   int b=1;
   int c;
   if(n==0) c=1;
   else if(n==1) c=1;
   else for(int i=2;i<=n;i++)  //应该 n 从 2 开始算起
   {
     c=a+b;
     a=b;
     b=c;
   }
   return c;
}
```

解答：

现在大多数系统都是将低字位放在前面，而结构体中位域的申明一般是先声明高位。

100 的二进制是 001 100 100

低位在前　高位在后

001----s3

100----s2

100----s1

所以结果应该是 1

如果先申明的在低位则：

001----s1

100----s2

100----s3

结果是 4

1、原题跟 little-endian，big-endian 没有关系

2、原题跟位域的存储空间分配有关，到底是从低字节分配还是从高字节分配，从 Dev C++和 VC7.1 上看，都是从低字节开始分配，并且连续分配，中间不空，不像谭的书那样会留空位

3、原题跟编译器有关，编译器在未用堆栈空间的默认值分配上有所不同，Dev C++未用空间分配为 01110111b，VC7.1 下为 11001100b,所以在 Dev C++下的结果为 5，在 VC7.1 下为 1。

注：PC 一般采用 little-endian，即高高低低，但在网络传输上，一般采用 big-endian，即高低低高，华为是做网络的，所以可能考虑 big-endian 模式，这样输出结果可能为 4

35、判断一个字符串是不是回文

```
int IsReverseStr(char *aStr)
```

```
{
int i,j;
int found=1;
if(aStr==NULL)
return -1;
j=strlen(aStr);
for(i=0;i<j/2;i++)
if(*(aStr+i)!=*(aStr+j-i-1))
{
found=0;
break;
}
return found;
}
```

36、Josephu 问题为：设编号为 1，2，… n 的 n 个人围坐一圈，约定编号为 k（1<=k<=n）的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

数组实现：

```
#include <stdio.h>
#include <malloc.h>
int Josephu(int n, int m)
{
  int flag, i, j = 0;
  int *arr = (int *)malloc(n * sizeof(int));
  for (i = 0; i < n; ++i)
    arr[i] = 1;
  for (i = 1; i < n; ++i)
  {
    flag = 0;
    while (flag < m)
    {
      if (j == n)
        j = 0;
      if (arr[j])
        ++flag;
      ++j;
    }
    arr[j - 1] = 0;
    printf("第%4d 个出局的人是：%4d 号\n", i, j);
  }
  free(arr);
  return j;
}
int main()
{
  int n, m;
  scanf("%d%d", &n, &m);
  printf("最后胜利的是%d 号！\n", Josephu(n, m));
```

```
  system("pause");
  return 0;
}
```
链表实现：
```
#include <stdio.h>
#include <malloc.h>
typedef struct Node
{
  int index;
  struct Node *next;
}JosephuNode;
int Josephu(int n, int m)
{
  int i, j;
  JosephuNode *head, *tail;
  head = tail = (JosephuNode *)malloc(sizeof(JosephuNode));
  for (i = 1; i < n; ++i)
  {
    tail->index = i;
    tail->next = (JosephuNode *)malloc(sizeof(JosephuNode));
    tail = tail->next;
  }
  tail->index = i;
  tail->next = head;

  for (i = 1; tail != head; ++i)
  {
    for (j = 1; j < m; ++j)
    {
      tail = head;
      head = head->next;
    }
    tail->next = head->next;
    printf("第%4d 个出局的人是：%4d 号\n", i, head->index);
    free(head);
    head = tail->next;
  }
  i = head->index;
  free(head);
  return i;
}
int main()
{
  int n, m;
  scanf("%d%d", &n, &m);
  printf("最后胜利的是%d 号！\n", Josephu(n, m));
  system("pause");
  return 0;
}
```

37、已知 strcpy 函数的原型是：

```
char * strcpy(char * strDest,const char * strSrc);
```

1.不调用库函数，实现 strcpy 函数。

2.解释为什么要返回 char *。

解说：

1.strcpy 的实现代码

```
char * strcpy(char * strDest,const char * strSrc)
{
        if ((strDest==NULL)||(strSrc==NULL)) file://[/1]
                throw "Invalid argument(s)"; //[2]
        char * strDestCopy=strDest;  file://[/3]
        while ((*strDest++=*strSrc++)!='\0'); file://[/4]
        return strDestCopy;
}
```

错误的做法：

[1]

(A)不检查指针的有效性，说明答题者不注重代码的健壮性。

(B)检查指针的有效性时使用((!strDest)||(!strSrc))或(!(strDest&&strSrc))，说明答题者对 C 语言中类型的隐式转换没有深刻认识。在本例中 char *转换为 bool 即是类型隐式转换，这种功能虽然灵活，但更多的是导致出错概率增大和维护成本升高。所以 C++专门增加了 bool、true、false 三个关键字以提供更安全的条件表达式。

(C)检查指针的有效性时使用((strDest==0)||(strSrc==0))，说明答题者不知道使用常量的好处。直接使用字面常量（如本例中的 0）会减少程序的可维护性。0 虽然简单，但程序中可能出现很多处对指针的检查，万一出现笔误，编译器不能发现，生成的程序内含逻辑错误，很难排除。而使用 NULL 代替 0，如果出现拼写错误，编译器就会检查出来。

[2]

(A)return new string("Invalid argument(s)");，说明答题者根本不知道返回值的用途，并且他对内存泄漏也没有警惕心。从函数中返回函数体内分配的内存是十分危险的做法，他把释放内存的义务抛给不知情的调用者，绝大多数情况下，调用者不会释放内存，这导致内存泄漏。

(B)return 0;，说明答题者没有掌握异常机制。调用者有可能忘记检查返回值，调用者还可能无法检查返回值（见后面的链式表达式）。妄想让返回值肩负返回正确值和异常值的双重功能，其结果往往是两种功能都失效。应该以抛出异常来代替返回值，这样可以减轻调用者的负担、使错误不会被忽略、增强程序的可维护性。

[3]

(A)忘记保存原始的 strDest 值，说明答题者逻辑思维不严密。

[4]

(A)循环写成 while (*strDest++=*strSrc++);，同[1](B)。

(B)循环写成 while (*strSrc!='\0') *strDest++=*strSrc++;，说明答题者对边界条件的检查不力。循环体结束后，strDest 字符串的末尾没有正确地加上'\0'。

第四部分：附加部分

1、位域 ：

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态， 用一位二进位即可。为了节省存储空间，并使处理简便， C 语言又提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进位划分为几个不同的区域， 并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。 这样就可以把几个不同的对象用一个字节的二进制位域来表示。一、位域的定义和位域变量的说明位域定义与结构定义相仿，其形式为：

struct 位域结构名

{ 位域列表 };
其中位域列表的形式为： 类型说明符 位域名：位域长度
例如：

```
struct bs
{
int a:8;
int b:2;
int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。
例如：

```
struct bs
{
int a:8;
int b:2;
int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。对于位域的定义尚有以下几点说明：

1．一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
unsigned a:4
unsigned :0 /*空域*/
unsigned b:4 /*从下一单元开始存放*/
unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

2．由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。

3．位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
{
int a:1
int :2 /*该 2 位不能使用*/
int b:3
int c:2
};
```

从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。

二、位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名•位域名 位域允许用各种格式输出。

```
main(){
struct bs
{
unsigned a:1;
```

```
unsigned b:3;
unsigned c:4;
} bit,*pbit;
bit.a=1;
bit.b=7;
bit.c=15;
pri
```

改错：

```
#include <stdio.h>

int main(void) {

    int **p;
    int arr[100];

    p = &arr;

    return 0;
}
```

解答：

搞错了,是指针类型不同,

```
int **p; //二级指针
&arr; //得到的是指向第一维为 100 的数组的指针
#include <stdio.h>
int main(void) {
int **p, *q;
int arr[100];
q = arr;
p = &q;
return 0;
}
```

附录 A

C 语言语法

C语言的ISO标准的附件B给出了一套完整的语言语法规则。本附录再现了这些规则，而且我把这些规则编写得更易读[①]。在每条规则中，语法项的名称会出现在左侧并以黑体字的形式显示出来。符号 | 、*、+、[、]、(和)具有下列的含义：

- 项目$_1$ | 项目$_2$表示项目$_1$和项目$_2$可以两者选其一。
- 项目*表示项目可以重复零次或多次。
- 项目+表示项目可以重复一次或多次。
- [项目]表示项目是可选的。
- (和)用于可选择项组。

但是，当把这些符号设置成为**courier粗体**时，它们具有通常C语言中的含义。虽然大多数规则相当清楚，但是有一些需要更深入的解释说明。如果需要，还会有注释。

A.1 记号

记号 关键字 | 标识符 | 常量 | 字符串字面量 | 运算符 | 标点符号

预处理 头文件名 | 标识符 | 预处理数 | 字符常量 | 字符串字面量 | 运算符 | 标点符号 | 每种不属于上述字符的非空白字符

> “记号”是组成程序不可分割的符号。预处理器识别一些编译器不识别的记号，因此记号和预处理记号之间有区别。

A.2 关键字

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

A.3 标识符

标识符 非数字 (非数字 | 数字)*

非数字 **_** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

数字 **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

A.4 常量

常量 浮点常量 | 整型常量 | 枚举常量 | 字符常量

浮点常量 小数常量 [指数部分] [浮点后缀] | 数字+ 指数部分 [浮点后缀]

小数的常量 数字* **.** 数字+ | 数字+ **.**

指数部分 (**e** | **E**) [**+** | **-**] 数字+

① 这些资料经ANSI许可改编自American National Standards Institute ANSI/ISO 9899©1990。这个标准的副本可从ANSI购买（ANSI，11 West 42nd Street, New York, NY 10036）。

浮点后缀　**f** | **l** | **F** | **L**

默认情况下，浮点常量是以double格式存储的。在浮点常量末尾的字母f或F通知编译器要以float型存储常量。l或L则通知编译器以long double型存储常量。

整型常量　十进制常量 [整数后缀] | 八进制常量 [整数后缀] |
十六进制常量 [整数后缀]

十进制常量　非零数字 数字*

八进制常量　**0** 八进制数字*

注意，0是正式区分作为八进制常量的，不是十进制常量。当然，这个特殊的举措没有什么差异，因为0在任何情况下的含义相同。

十六进制常量　(**0x** | **0X**) 十六进制数字+

非零的数字　**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

八进制数字　**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

十六进制数字　**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

整数后缀　无符号后缀 [长后缀] | 长后缀 [无符号后缀]

588

无符号后缀　**u** | **U**

长后缀　**l** | **L**

整型常量末尾的字母u或U通知编译器把常量作为unsinged int型来存储。l或L则通知编译器把常量作为long int型来存储。当常量后跟有两种字母时（顺序任意），都把常量作为unsigned long int型来存储。

枚举常量　标识符

枚举常量用于枚举元规则中（见A.11节）。

字符常量　**'***c*字符+**'** | **L'***c*字符+**'**

如果出现了L，常量表示的是宽字符。

c字符　除去**'**、****以外的任何字符，或换行符 | 转义序列

小心正确地解释上述的规则。规则说明字符常量不包含换行符，但是它没说明字符常量不能包含'或\字符。这两者始终会作为转义序列的内容出现在字符常量中。

转义序列　简单转义序列 | 八进制转义序列 | 十六进制转义序列

简单转义序列　**\'** | **\"** | **\?** | **** | **\a** | **\b** | **\f** | **\n** | **\r** | **\t** | **\v**

八进制转义序列　**** 八进制数字 [八进制数字] [八进制数字]

十六进制转义序列　**\x** 十六进制数字+

由于历史原因，八进制转义序列限定为三个数字。另一方面，十六进制转义序列可以是任意数量的数字。

A.5　字符串字面量

字符串字面量　**"***s*字符***"** | **L"***s*字符***"**

如果出现了L，字符串字面量就为宽字符串。

s字符　除去**"**、****以外的任何字符，或换行符 | 转义序列

这条规这不表示字符串常量不能含有"或\字符。这两者始终会作为转义序列的内容出现在字符常量中。

A.6　运算符

运算符　**[** | **]** | **(** | **)** | **.** | **->** | **++** | **--** | **&** | ***** | **+** | **-** | **~** | **!** | **sizeof** | **/** | **%** | **<<** | **>>** | **<** | **>** | **<=** | **>=** | **==** | **!=** | **^** | **|** | **&&** | **||** | **?** | **:** | **=** | ***=** | **/=** | **%=** | **+=** | **-=** | **<<=** | **>>=** | **&=** | **^=** | **|=** | **,** | **#** | **##**

为了方便，把预处理器运算符#和##也包含到C语言的普通运算符组中了。

A.7　标点符号

标点符号　**[** | **]** | **(** | **)** | **{** | **}** | ***** | **,** | **:** | **=** | **;** | **...** | **#**

一些标点符号也是运算符，这要依赖于实际内容。例如，记号=当用在声明中时是标点符号，主要是为了把变量和它的初始符分离开，或者是为了把枚举常量和它的值分离开。而当记号=用于表达式时，它就是赋值运算符。记号...（省略号）用于写带有可变长度实参列表的函数。 589

A.8　头文件名

头文件名　**<** *h*字符$^+$ **>** | **"***q*字符$^+$**"**

*h*字符　除了换行符和**>**的任何字符。

*q*字符　除了换行符和**"**的任何字符。

头文件名几乎可以包含几无任何字符。允许如此灵活的原因是头文件名常常含有依赖操作系统的信息（例如路径）。

A.9　预处理数

预处理数　[**.**] 数字 (数字 | 非数字 | (**e** | **E**) (**+** | **-**) | **.**)*

在预处理期间，对要监测的数应用这条简单规则，这条规则允许一些不合法的数（比如0x0y）溜掉。但是，这些不合法的数稍后会由编译器检查出来，所以不会有害的。

A.10　表达式

基本表达式　标识符 | 常量 | 字符串字面量 | **(** 表达式 **)**

基本表达式是不可分割的表达式。不是因为它是单独的标识符、常量或字符串字面量，就是因为它是用括号闭合的。所有其他的表达式都服从于C语言的优先级和结合性规则，这些规则都嵌入到随后的19条规则中了。

后缀表达式　基本表达式 (**[** 表达式 **]** | **(** [参数表达式列表] **)** | **.** 标识符 | **->** 标识符 | **++** | **--**)*

参数表达式列表　赋值表达式 (**,** 赋值表达式)*

为了避免作为参数分隔符的逗号标点和逗号运算符的混淆，函数调用中的实际参数必须是“赋值表达式”，而不能是任意表达式。

一元表达式　(**++** | **--** **sizeof**)* (后缀表达式 | 一元运算符 强制类型转换表达式 | **sizeof (**类型名**)**)

一元运算符　**&** | ***** | **+** | **-** | **~** | **!**

强制类型转换表达式　(**(** 类型名 **)**)* 一元表达式

乘法类表达式　强制类型转换表达式 ((***** | **/** | **%**)强制类型转换表达式)*

加法类表达式　乘法类表达式 ((**+** | **-**) 乘法类表达式)*

移位表达式　加法类表达式 ((**<<** | **>>**) 加法类表达式)*

关系表达式　移位表达式 ((**<** | **>** | **<=** | **>=**) 移位表达式)*

判等表达式　关系表达式 ((**==** | **!** **=**) 关系表达式)*

与表达式　判等表达式 (**&** 判等表达式)*

异或表达式　与表达式 (**^** 与表达式)* 590

或表达式　异或表达式 (**|** 异或表达式)*

逻辑与表达式　或表达式 (**&&** 或表达式)*

逻辑或表达式　逻辑与表达式 (**||**逻辑与表达式)*

条件表达式　逻辑或表达式 (**?** 表达式 **:** 条件表达式)*

赋值表达式　(一元表达式 赋值运算符)* 条件表达式

赋值运算符　= | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=

表达式　赋值表达式 (, 赋值表达式)*

常量表达式　条件表达式

把常量表达式定义成条件表达式，而不是通常的表达式，这是因为C禁止在常量表达式中有赋值运算符和逗号运算符。（虽然语法规则中没有显示出来，但是C语言也不允许自增、自减运算符和函数调用。）

A.11　声明

声明　声明说明符 [初始声明符列表];

声明说明符　(存储类型说明符 | 类型说明符 | 类型限定符)+

前述的规则有些误导，因为它说明声明可以包含多于一个的存储类型说明符。实际上，只允许一个真正的存储类型，而且它必须在类型说明符和类型限定符之前。规则的正确理解是可以用typedef（由于语法目的所以考虑成是存储类型说明符）开始，后边跟着存储类型。类型说明符和类型限定符才是真的可以像规则显示的那样混合，这样会引发诸如int const unsigned volatile long这样的奇异组合。

初始声明符列表　初始声明符 (, 初始声明符)*

初始声明符　声明符 [= 初始化式]

存储类型说明符　typedef | extern | static | auto | register

为了简化语法规则，所以把typedef与真正的存储类型混在一起了。

类型说明符　void | char | short | int | long | float | double | signed | unsigned | 结构或联合说明符 | 枚举说明符 | 类型定义名

结构或联合说明符　(struct | union) (说明符 | [说明符] { 结构声明+ })

结构声明　说明符限定符列表 结构声明符列表 ;

说明符限定符列表　(类型说明符 | 类型限定符)+

结构声明符列表　结构声明符 (, 结构声明符)*

结构声明符　声明符 | [声明符] : 常量表达式

在前述的规则中常量表达式说明位域的宽度。如果出现了常量表达式，则可以忽略声明符，并产生一个未命名的位域。

枚举说明符　enum (标识符 | [标识符] { 枚举元列表 })

591

枚举元列表　枚举元 (, 枚举元)*

枚举元　枚举常量 [= 常量表达式]

类型限定符　const | volatile

声明符　(* 类型限定符*)* 直接声明符

直接声明符　(说明符 | (声明符)) ([[常量表达式]] | (参数类型列表) | ([标识符列表]))*

参数类型列表　参数声明 (, 参数声明)* [, ...]

在参数列表末尾出现的，...表明可能跟随额外的可变数量的参数。

参数声明　声明说明符 [声明符 | 抽象声明符]

标识符列表　标识符 (, 标识符)*

类型名　说明符限定符列表 [抽象符声明符]

类型名用于一元表达式和强制类型转换表达式规则中（见**表达式**）。

抽象声明符　(* 类型限定符*)+ | (* 类型限定符*)* 直接抽象声明符

普通声明符包含名字和关于名字性质的信息；而抽象声明符说明了性质，但是忽略名字。函数原型void f (int **, float[]);就使用了抽象声明符**和[]来帮助描述f的参数类型。

直接抽象声明符　(抽象声明符) | [(抽象声明符)] ([[常量表达式]] | ([参数类型列表]))+

类型定义名　标识符

初始化式　赋值表达式 | **{** *初始化式列表* [**,**] **}**
　　不，这不是错误。初始化式列表可以真的后边跟随（多余的）逗号。
初始化式列表　*初始化式* (**,** *初始化式*)*

A.12　语句

语句　标号语句 | 复合语句 | 表达式语句 | 选择语句 | 循环语句 | 跳转语句
标号语句　标识符 **:** 语句 | **case** 常量表达式 **:** 语句 | **default :** 语句
　　最后两种格式的标号语句只允许出现在switch语句中。
复合语句　**{** *声明** *语句** **}**
表达式语句　[表达式] **;**
　　由于语法目的，空语句被看成是表达式语句，这种表达式语句是缺少表达式的。
选择语句　**if (** 表达式 **)** 语句 [**else** 语句] | **switch (** 表达式 **)** 语句　592
　　虽然没有严格要求，但是switch语句体事实上始终是复合语句。虽然会忽略在声明中的初始化式，但是复合语句可以有声明。
循环语句　**while (** 表达式 **)** 语句 |
　　do 语句 **while (** 表达式 **) ;** |
　　for ([表达式] **;** [表达式] **;** [表达式] **)** 语句
跳转语句　**goto** 标识符 **;** | **continue ;** | **break ;** | **return** [表达式] **;**

A.13　外部定义

翻译单元　外部声明+
外部声明　函数定义 | 声明
函数定义　[声明说明符] 声明符 声明* 复合语句
　　声明说明符描述函数的返回类型。声明符给出了函数名和参数列表。声明（只出现在经典C风格的函数定义中）说明参数的类型。复合语句是函数体。

A.14　预处理指令

预处理文件　[组]
组　([预处理记号] 换行 | *if* 部分 | 控制行)+
*if*部分　*if* 组 *elif* 组* [*else*组] *endif* 行
*if*组　**# if** 常量表达式 换行 [组] |
　　# ifdef 标识符 换行 [组] |
　　# ifndef 标识符 换行 [组]
*elif*组　**# elif** 常量表达式 换行 [组]
*else*组　**# else** 换行 [组]
*endif*行　**# endif** 换行
控制行　**# include** 预处理记号 换行 |
　　# define 标识符 替换列表 换行 |
　　# define 标识符 **(** [标识符列表] **)** 替换列表 换行 |
　　# undef 标识符 换行 |
　　# line 预处理记号 换行 |
　　# error [预处理记号] 换行 |
　　# pragma [预处理记号] 换行 |
　　# 换行
lparen　没有前述空白的左圆括号字符。
替换列表　[预处理记号]
预处理记号　预处理记号+
换行　换行符　593

附录 B

C 语言运算符

优先级	名　　称	符　　号	结合性
1	数组下标	[]	左结合性
1	函数调用	()	左结合性
1	结构和联合的成员	. ->	左结合性
1	自增（后缀）	++	左结合性
1	自减（后缀）	--	左结合性
2	自增（前缀）	++	右结合性
2	自减（前缀）	--	右结合性
2	取地址	&	右结合性
2	间接寻址	*	右结合性
2	一元正号	+	右结合性
2	一元负号	-	右结合性
2	按位求反	~	右结合性
2	逻辑非	!	右结合性
2	计算内存长度	sizeof	右结合性
3	强制类型转换	()	右结合性
4	乘法类的	* / %	左结合性
5	加法类的	+ -	左结合性
6	按位移位	<< >>	左结合性
7	关系	< > <= >=	左结合性
8	判等	== !=	左结合性
9	按位与	&	左结合性
10	按位异或	^	左结合性
11	按位或	|	左结合性
12	逻辑与	&&	左结合性
13	逻辑或	||	左结合性
14	条件	?:	右结合性
15	赋值	= *= /= %= += -= <<= >>= &= ^= |=	右结合性
16	逗号	,	左结合性

让你不再害怕指针

前言:复杂类型说明

要了解指针,多多少少会出现一些比较复杂的类型,所以我先介绍一下如何完全理解一个复杂类型,要理解复杂类型其实很简单,一个类型里会出现很多运算符,他们也像普通的表达式一样,有优先级,其优先级和运算优先级一样,所以我总结了一下其原则:

从变量名处起,根据运算符优先级结合,一步一步分析.

下面让我们先从简单的类型开始慢慢分析吧:

```
int  p;          //这是一个普通的整型变量

int  *p;         //首先从 P 处开始,先与*结合,所以说明 P 是一
                 //个指针,然后再与 int 结合,说明指针所指向
                 //的内容的类型为 int 型.所以 P 是一个返回整
                 //型数据的指针

int  p[3];       //首先从 P 处开始,先与[]结合,说明 P 是一个数
                 //组,然后与 int 结合,说明数组里的元素是整
                 //型的,所以 P 是一个由整型数据组成的数组

int  *p[3];      //首先从 P 处开始,先与[]结合,因为其优先级
```

```
                //比*高,所以 P 是一个数组,然后再与*结合,说明
                //数组里的元素是指针类型,然后再与 int 结合,
                //说明指针所指向的内容的类型是整型的,所以
                //P 是一个由返回整型数据的指针所组成的数组

int  (*p) [3]; //首先从 P 处开始,先与*结合,说明 P 是一个指针
                //然后再与[]结合(与"()"这步可以忽略,只是为
                //了改变优先级),说明指针所指向的内容是一个
                //数组,然后再与 int 结合,说明数组里的元素是
                //整型的.所以 P 是一个指向由整型数据组成的数
                //组的指针

int  **p;       //首先从 P 开始,先与*结合,说是 P 是一个指针,然
                //后再与*结合,说明指针所指向的元素是指针,然
                //后再与 int 结合,说明该指针所指向的元素是整
                //型数据.由于二级指针以及更高级的指针极少用
                //在复杂的类型中,所以后面更复杂的类型我们就
                //不考虑多级指针了,最多只考虑一级指针.

int  p(int);   //从 P 处起,先与()结合,说明 P 是一个函数,然后进入
                //()里分析,说明该函数有一个整型变量的参数
                //然后再与外面的 int 结合,说明函数的返回值是
                //一个整型数据
```

```
Int (*p)(int); //从P处开始,先与指针结合,说明P是一个指针,然后与
               //()结合,说明指针指向的是一个函数,然后再与()里的
               //int 结合,说明函数有一个 int 型的参数,再与最外层的
               //int 结合,说明函数的返回类型是整型,所以 P 是一个指
               //向有一个整型参数且返回类型为整型的函数的指针

int *(*p(int))[3];     //可以先跳过,不看这个类型,过于复杂
               //从P开始,先与()结合,说明P是一个函数,然后进
               //入()里面,与 int 结合,说明函数有一个整型变量
               //参数,然后再与外面的*结合,说明函数返回的是
               //一个指针,,然后到最外面一层,先与[]结合,说明
               //返回的指针指向的是一个数组,然后再与*结合,说
               //明数组里的元素是指针,然后再与 int 结合,说明指
               //针指向的内容是整型数据.所以P是一个参数为一个
               //整数据且返回一个指向由整型指针变量组成的数组
               //的指针变量的函数.
```

说到这里也就差不多了,我们的任务也就这么多,理解了这几个类型,其它的类型对我们来说也是小菜了,不过我们一般不会用太复杂的类型,那样会大大减小程序的可读性,请慎用,这上面的几种类型已经足够我们用了.

1、细说指针

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。要搞清一个指针需要搞清指针的四方面的内容：指针的类型、指针所指向的类型、指针的值或者叫指针所指向的内存区、指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

```
(1) int*ptr;
(2) char*ptr;
(3) int**ptr;
(4) int (*ptr) [3];
(5) int* (*ptr) [4];
```

1.指针的类型

从语法的角度看，你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。让我们看看例一中各个指针的类型：

```
(1) int*ptr; //指针的类型是 int*
(2) char*ptr; //指针的类型是 char*
(3) int**ptr; //指针的类型是 int**
(4) int (*ptr) [3]; //指针的类型是 int (*) [3]
```

(5) int*(*ptr)[4]; //指针的类型是 int*(*)[4]

怎么样？找出指针的类型的方法是不是很简单？

2. 指针所指向的类型

当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。例如:

(1) int*ptr;　　//指针所指向的类型是 int

(2) char*ptr;　　//指针所指向的的类型是 char

(3) int**ptr;　　//指针所指向的的类型是 int*

(4) int(*ptr)[3];　//指针所指向的的类型是 int()[3]

(5) int*(*ptr)[4]; //指针所指向的的类型是 int*()[4]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对 C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

3. 指针的值----或者叫指针所指向的内存区或地址

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在 32 位程序里，所有类型的指针的值都是一个 32 位

整数，因为 32 位程序里内存地址全都是 32 位长。 指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为 sizeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是 XX，就相当于说该指针指向了以 XX 为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？（重点注意）

4 指针本身所占据的内存区

指针本身占了多大的内存？你只要用函数 sizeof(指针的类型)测一下就知道了。在 32 位平台里，指针本身占据了 4 个字节的长度。

指针本身占据的内存这个概念在判断一个指针表达式（后面会解释）是否是左值时很有用。

2、指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的，以单元为单位。例如:

例二:

```
char a[20];
int *ptr=(int *)a; //强制类型转换并不会改变 a 的类型
ptr++;
```

在上例中，指针 ptr 的类型是 int*,它指向的类型是 int，它被初始化为指向整型变量 a。接下来的第 3 句中，指针 ptr 被加了 1，编译器是这样处理的：它把指针 ptr 的值加上了 sizeof(int)，在 32 位程序中，是被加上了 4，因为在 32 位程序中，int 占 4 个字节。由于地址是用字节做单位的，故 ptr 所指向的地址由原来的变量 a 的地址向高地址方向增加了 4 个字节。由于 char 类型的长度是一个字节，所以，原来 ptr 是指向数组 a 的第 0 号单元开始的四个字节，此时指向了数组 a 中从第 4 号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子:

例三:

```
int array[20]={0};
int *ptr=array;
for(i=0; i<20; i++)
{
  (*ptr)++;
  ptr++;
```

```
}
```

这个例子将整型数组中各个单元的值加 1。由于每次循环都将指针 ptr 加 1 个单元，所以每次循环都能访问数组的下一个单元。

再看例子:

例四:

```
char a[20]="You_are_a_girl";
int *ptr=(int *)a;
ptr+=5;
```

在这个例子中，ptr 被加上了 5，编译器是这样处理的：将指针 ptr 的值加上 5 乘 sizeof(int)，在 32 位程序中就是加上了 5 乘 4=20。由于地址的单位是字节，故现在的 ptr 所指向的地址比起加 5 后的 ptr 所指向的地址来说，向高地址方向移动了 20 个字节。在这个例子中，没加 5 前的 ptr 指向数组 a 的第 0 号单元开始的四个字节，加 5 后，ptr 已经指向了数组 a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，ptr 是被减去 5，那么处理过程大同小异，只不过 ptr 的值是被减去 5 乘 sizeof(int)，新的 ptr 指向的地址将比原来的 ptr 所指向的地址向低地址方向移动了 20 个字节。

下面请允许我再举一个例子:(一个误区)

例五:

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

```
  char a[20]=" You_are_a_girl";
  char *p=a;
  char **ptr=&p;

  //printf("p=%d\n",p);
  //printf("ptr=%d\n",ptr);
  //printf("*ptr=%d\n",*ptr);
  printf("**ptr=%c\n",**ptr);

  ptr++;

  //printf("ptr=%d\n",ptr);
  //printf("*ptr=%d\n",*ptr);
  printf("**ptr=%c\n",**ptr);
}
```

误区一、输出答案为 Y 和 o

误解: ptr 是一个 char 的二级指针, 当执行 ptr++; 时, 会使指针加一个 sizeof(char), 所以输出如上结果, 这个可能只是少部分人的结果.

误区二、输出答案为 Y 和 a

误解: ptr 指向的是一个 char *类型, 当执行 ptr++; 时, 会使指针加一个 sizeof(char *) (有可能会有人认为这个值为 1, 那就会得到误区一的答案, 这个值应该是 4, 参考前面内容), 即&p+4; 那进行一次取值运算不

就指向数组中的第五个元素了吗?那输出的结果不就是数组中第五个元素了吗?答案是否定的.

正解: ptr 的类型是 char **,指向的类型是一个 char *类型,该指向的地址就是 p 的地址(&p),当执行 ptr++;时,会使指针加一个 sizeof(char *),即&p+4;那*(&p+4)指向哪呢,这个你去问上帝吧,或者他会告诉你在哪?所以最后的输出会是一个随机的值,或许是一个非法操作.

总结一下:

一个指针 ptrold 加(减)一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加(减少)了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向高(低)地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

指针和指针进行加减:

两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的结果指向一个不知所向的地方，而且毫无意义。两个指针可以进行减法操作，但必须类型相同，一般用在数组方面，不多说了。

3、运算符&和*

这里&是取地址运算符，*是间接运算符。

&a 的运算结果是一个指针，指针的类型是 a 的类型加个*，指针所指向的类型是 a 的类型，指针所指向的地址嘛，那就是 a 的地址。

*p 的运算结果就五花八门了。总之*p 的结果是 p 所指向的东西，这个东西有这些特点：它的类型是 p 指向的类型，它所占用的地址是 p 所指向的地址。

例六：

```
int a=12;  int b;    int *p;   int **ptr;
p=&a;      //&a 的结果是一个指针，类型是 int*，指向的类型是
           //int，指向的地址是 a 的地址。
*p=24;     //*p 的结果，在这里它的类型是 int，它所占用的地址是
           //p 所指向的地址，显然，*p 就是变量 a。
ptr=&p;    //&p 的结果是个指针，该指针的类型是 p 的类型加个*，
           //在这里是 int **。该指针所指向的类型是 p 的类型，这
           //里是 int*。该指针所指向的地址就是指针 p 自己的地址。
*ptr=&b;   //*ptr 是个指针，&b 的结果也是个指针，且这两个指针
           //的类型和所指向的类型是一样的，所以用&b 来给*ptr 赋
           //值就是毫无问题的了。
**ptr=34;  //*ptr 的结果是 ptr 所指向的东西，在这里是一个指针，
           //对这个指针再做一次*运算，结果是一个 int 类型的变量。
```

4、指针表达式

一个表达式的结果如果是一个指针，那么这个表达式就叫指针表式。

下面是一些指针表达式的例子:

例七:

```
int a,b;
int array[10];
int *pa;
pa=&a;              //&a 是一个指针表达式。
Int **ptr=&pa;      //&pa 也是一个指针表达式。
*ptr=&b;            //*ptr 和&b 都是指针表达式。
pa=array;
pa++;               //这也是指针表达式。
```

例八:

```
char *arr[20];
char **parr=arr; //如果把 arr 看作指针的话，arr 也是指针表达式
char *str;
str=*parr;          //*parr 是指针表达式
str=*(parr+1);      //*(parr+1)是指针表达式
str=*(parr+2);      //*(parr+2)是指针表达式
```

由于指针表达式的结果是一个指针,所以指针表达式也具有指针所具有的四个要素:指针的类型,指针所指向的类型,指针指向的内存区,指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针 pa，既然 pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

5、数组和指针的关系

数组的数组名其实可以看作一个指针。看下例:

例九:

```
intarray[10]={0,1,2,3,4,5,6,7,8,9},value;
value=array[0];      //也可写成: value=*array;
value=array[3];      //也可写成: value=*(array+3);
value=array[4];      //也可写成: value=*(array+4);
```

上例中，一般而言数组名 array 代表数组本身，类型是 int[10]，但如果把 array 看做指针的话，它指向数组的第 0 个单元，类型是 int* ，所指向的类型是数组单元的类型即 int。因此*array 等于 0 就一点也不奇怪了。同理，array+3 是一个指向数组第 3 个单元的指针，所以*(array+3)等于 3。其它依此类推。

例十:

```
char *str[3]={
  "Hello,thisisasample!",
  "Hi,goodmorning.",
  "Helloworld"
};
chars[80];
strcpy(s,str[0]);   //也可写成 strcpy(s,*str);
strcpy(s,str[1]);   //也可写成 strcpy(s,*(str+1));
strcpy(s,str[2]);   //也可写成 strcpy(s,*(str+2));
```

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 str 当作一个指针的话，它指向数组的第 0 号单元，它的类型是 char **，它指向的类型是 char *。

*str 也是一个指针，它的类型是 char *，它所指向的类型是 char，它指向的地址是字符串"Hello,thisisasample!"的第一个字符的地址，即'H'的地址。注意:字符串相当于是一个数组,在内存中以数组的形式储存,只不过字符串是一个数组常量,内容不可改变,且只能是右值.如果看成指针的话,他即是常量指针,也是指针常量.

str+1 也是一个指针，它指向数组的第 1 号单元，它的类型是 char**，它指向的类型是 char*。

(str+1)也是一个指针，它的类型是 char，它所指向的类型是 char，它指向 "Hi,goodmorning."的第一个字符'H'

下面总结一下数组的数组名(数组中储存的也是数组)的问题:
声明了一个数组 TYPE array[n]，则数组名称 array 就有了两重含义:
第一，它代表整个数组，它的类型是 TYPE[n]；第二 ，它是一个常量指针，该指针的类型是 TYPE*，该指针指向的类型是 TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第 0 号单元，该指针自己占有单独的内存区，注意它和数组第 0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 array++的表达式是错误的。

在不同的表达式中数组名 array 可以扮演不同的角色。

在表达式 sizeof(array)中，数组名 array 代表数组本身，故这时 sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第 0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式 array+n（其中 n=0，1，2，.....）中，array 扮演的是指针，故 array+n 的结果是一个指针，它的类型是 TYPE *，它指向的类型是 TYPE，它指向数组第 n 号单元。故 sizeof(array+n)测出的是指针类型的大小。在 32 位程序中结果是 4

例十一:

```
int array[10];
int (*ptr)[10];
ptr=&array;:
```

上例中 ptr 是一个指针，它的类型是 int(*)[10]，他指向的类型是 int[10] ，我们用整个数组的首地址来初始化它。在语句 ptr=&array 中，array 代表数组本身。

本节中提到了函数 sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者。例如:

```
int(*ptr)[10];
```

则在 32 位程序中，有:

```
sizeof(int(*)[10])==4
sizeof(int[10])==40
```

sizeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

6、指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十二:

```
struct MyStruct
{
  int a;
  int b;
  int c;
};
struct MyStruct ss={20,30,40};
        //声明了结构对象 ss，并把 ss 的成员初始化为 20，30 和 40。
struct MyStruct *ptr=&ss;
        //声明了一个指向结构对象 ss 的指针。它的类型是
        //MyStruct *,它指向的类型是 MyStruct。
int *pstr=(int*)&ss;
        //声明了一个指向结构对象 ss 的指针。但是 pstr 和
        //它被指向的类型 ptr 是不同的。
```

请问怎样通过指针 ptr 来访问 ss 的三个成员变量?

答案:

```
ptr->a;     //指向运算符，或者可以这们(*ptr).a,建议使用前者
ptr->b;
```

ptr->c;

又请问怎样通过指针 pstr 来访问 ss 的三个成员变量?

答案:

*pstr;　　//访问了 ss 的成员 a。

*(pstr+1);　　//访问了 ss 的成员 b。

*(pstr+2)　　//访问了 ss 的成员 c。

虽然我在我的 MSVC++6.0 上调式过上述代码，但是要知道，这样使用 pstr 来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元:　(将结构体换成数组)

例十三:

```
int array[3]={35,56,37};
int *pa=array;
```

通过指针 pa 访问数组 array 的三个单元的方法是:

*pa;　　//访问了第 0 号单元

*(pa+1);　　//访问了第 1 号单元

*(pa+2);　　//访问了第 2 号单元

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。

所有的 C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干个"填充字节"，这就导

致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使*pstr 访问到了结构对象 ss 的第一个成员变量 a，也不能保证*(pstr+1)就一定能访问到结构成员 b。因为成员 a 和成员 b 之间可能会有若干填充字节，说不定*(pstr+1)就正好访问到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，嘿，这倒是个不错的方法。不过指针访问结构成员的正确方法应该是象例十二中使用指针 ptr 的方法。

7、指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
int fun1(char *,int);
int (*pfun1)(char *,int);
pfun1=fun1;
int a=(*pfun1)("abcdefg",7); //通过函数指针调用函数。
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为实参。

例十四：

```
int fun(char *);
inta;
char str[]="abcdefghijklmn";
a=fun(str);
int fun(char *s)
{
int num=0;
for(int i=0;;)
{
    num+=*s;s++;
}
return num;
}
```

这个例子中的函数fun统计一个字符串中各个字符的ASCII码值之和。前面说了，数组的名字也是一个指针。在函数调用中，当把 str 作为实参传递给形参 s 后，实际是把 str 的值传递给了 s，s 所指向的地址就和 str 所指向的地址一致，但是 str 和 s 各自占用各自的存储空间。在函数体内对 s 进行自加 1 运算，并不意味着同时对 str 进行了自加 1 运算。

8、指针类型转换

当我们初始化一个指针或给一个指针赋值时，赋值号的左边是一个指针，赋值号的右边是一个指针表达式。在我们前面所举的例子中，绝大多数情况下，指针的类型和指针表达式的类型是一样的，指针所指向的类型和指针表达式所指向的类型是一样的。

例十五：

```
float f=12.3;
float *fptr=&f;
int *p;
```

在上面的例子中,假如我们想让指针 p 指向实数 f,应该怎么办?是用下面的语句吗?

```
p=&f;
```

不对。因为指针 p 的类型是 int *，它指向的类型是 int。表达式 &f 的结果是一个指针，指针的类型是 float *,它指向的类型是 float。两者不一致，直接赋值的方法是不行的。至少在我的 MSVC++6.0 上，对指针的赋值语句要求赋值号两边的类型一致，所指向的类型也一致，其它的编译器上我没试过，大家可以试试。为了实现我们的目的，需要进行"强制类型转换":

```
p=(int*)&f;
```

如果有一个指针 p，我们需要把它的类型和所指向的类型改为 TYEP *TYPE， 那么语法格式是： (TYPE *)p;

这样强制类型转换的结果是一个新指针，该新指针的类型是

TYPE *，它指向的类型是 TYPE，它指向的地址就是原指针指向的地址。而原来的指针 p 的一切属性都没有被修改。（切记）

一个函数如果使用了指针作为形参，那么在函数调用语句的实参和形参的结合过程中，必须保证类型一致 ，否则需要强制转换

例十六：

```
void fun(char*);
int a=125,b;
fun((char*)&a);
void fun(char*s)
{
charc;
c=*(s+3);*(s+3)=*(s+0);*(s+0)=c;
c=*(s+2);*(s+2)=*(s+1);*(s+1)=c;
}
```

注意这是一个 32 位程序，故 int 类型占了四个字节，char 类型占一个字节。函数 fun 的作用是把一个整数的四个字节的顺序来个颠倒。注意到了吗？在函数调用语句中，实参&a 的结果是一个指针，它的类型是 int *，它指向的类型是 int。形参这个指针的类型是 char *，它指向的类型是 char。这样，在实参和形参的结合过程中，我们必须进行一次从 int *类型到 char *类型的转换。结合这个例子，我们可以这样来想象编译器进行转换的过程：编译器先构造一个临时指针 char *temp，然后执行 temp=(char *)&a，最后再把 temp 的值传递给 s。所以最后的结果是：s 的类型是 char *,它指向的类型是 char，它指向的地址就是

a 的首地址。

我们已经知道，指针的值就是指针指向的地址，在 32 位程序中，指针的值其实是一个 32 位整数。那可不可以把一个整数当作指针的值直接赋给指针呢？就象下面的语句：

```
unsigned int a;
TYPE *ptr;          //TYPE 是 int，char 或结构类型等等类型。
a=20345686;
ptr=20345686;       //我们的目的是要使指针 ptr 指向地址 20345686
ptr=a;              //我们的目的是要使指针 ptr 指向地址 20345686
```

编译一下吧。结果发现后面两条语句全是错的。那么我们的目的就不能达到了吗？不，还有办法：

```
unsigned int a;
TYPE *ptr;          //TYPE 是 int，char 或结构类型等等类型。
a=N                 //N 必须代表一个合法的地址;
ptr=(TYPE*)a;       //呵呵，这就可以了。
```

严格说来这里的(TYPE *)和指针类型转换中的(TYPE *)还不一样。这里的(TYPE*)的意思是把无符号整数 a 的值当作一个地址来看待。上面强调了 a 的值必须代表一个合法的地址，否则的话，在你使用 ptr 的时候，就会出现非法操作错误。

想想能不能反过来，把指针指向的地址即指针的值当作一个整数取出来。完全可以。下面的例子演示了把一个指针的值当作一个整数取出来，然后再把这个整数当作一个地址赋给一个指针：

例十七：

```
int a=123,b;
int *ptr=&a;
char *str;
b=(int)ptr;        //把指针 ptr 的值当作一个整数取出来。
str=(char*)b;      //把这个整数的值当作一个地址赋给指针 str。
```

现在我们已经知道了，可以把指针的值当作一个整数取出来，也可以把一个整数值当作地址赋给一个指针。

9、指针的安全问题

看下面的例子:

例十八:

```
char s='a';
int *ptr;
ptr=(int *)&s;
*ptr=1298;
```

指针 ptr 是一个 int *类型的指针，它指向的类型是 int。它指向的地址就是 s 的首地址。在 32 位程序中，s 占一个字节，int 类型占四个字节。最后一条语句不但改变了 s 所占的一个字节，还把和 s 相临的高地址方向的三个字节也改变了。这三个字节是干什么的？只有编译程序知道，而写程序的人是不太可能知道的。也许这三个字节里存储了非常重要的数据，也许这三个字节里正好是程序的一条代码，而由于你对指针的马虎应用，这三个字节的值被改变了！这会造成崩溃性的错误。

让我们再来看一例:

例十九:

```
char a;
int *ptr=&a;
ptr++;
*ptr=115;
```

该例子完全可以通过编译，并能执行。但是看到没有？第 3 句对指针 ptr 进行自加 1 运算后，ptr 指向了和整形变量 a 相邻的高地址方向

的一块存储区。这块存储区里是什么？我们不知道。有可能它是一个非常重要的数据，甚至可能是一条代码。而第 4 句竟然往这片存储区里写入一个数据！这是严重的错误。所以在使用指针时，程序员心里必须非常清楚：我的指针究竟指向了哪里。在用指针访问数组的时候，也要注意不要超出数组的低端和高端界限，否则也会造成类似的错误。

在指针的强制类型转换：ptr1=(TYPE *)ptr2 中，如果 sizeof(ptr2 的类型)大于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是安全的。如果 sizeof(ptr2 的类型)小于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是不安全的。至于为什么，读者结合例十八来想一想，应该会明白的。

10、结束语

现在你是否已经觉得指针再也不是你所想的那么害怕了，如果你的回答是：对，我不怕了！哈哈，恭喜你，你已经掌握 C 语言的精华了，C 中唯一的难点就是指针，指针搞定其它小菜而已，重要的是实践，好吧，让我们先暂停 C 的旅程吧，开始我们的 C++编程，C 是对底层操作非常方便的语言，但开发大型程序本人觉得还是没有 C++方便，至少维护方面不太好做。而且 C++是面向对象的语言，现在基本已经是面向对象的天下了，所以建议学 C++。C++是一门难学易用的语言，要真正掌握 C++可不是那么容易的，将基本的学完后，就学数据结构吧，算法才是永恒的，程序设计语言层出不穷，永远学不完。学完之后就认真啃下 STL 这根骨头吧，推荐书籍---------范型编程与 STL 和 STL 源码剖析。如果你达到了这样要求，再一次恭喜你，你已经是个程序高手了，甚至可以说是个算法高手，因为 STL 里有大量的精华而高效的算法。唉，已经该说再见的时候了，让我们一起用我们的语言来谱写我们的人生吧，最后笑个，哈哈，睡觉了。好累，都 2：00 了

解析 C语言中的 sizeof　2009-03-07 14:34

一、 sizeof的概念

sizeof是 C语言的一种单目操作符，如 C语言的其他操作符 ++、 -等。它并不是函数。 sizeof操作符以字节形式给出了其操作数的存储大小。操作数可以是一个表达式或括在括号内的类型名。操作数的存储大小由操作数的类型决定。

二、 sizeof的使用方法

1、用于数据类型

sizeof使用形式： sizeof（type）

数据类型必须用括号括住。如 sizeof（int）。

2、用于变量

sizeof使用形式： sizeof（var_name）或 sizeof var_name

变量名可以不用括号括住。如 sizeof (var_name)， sizeof var_name等都是正确形式。带括号的用法更普遍，大多数程序员采用这种形式。

注意： sizeof操作符不能用于函数类型，不完全类型或位字段。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、 void类型等。

如 sizeof(max)若此时变量 max定义为 int max(),sizeof(char_v) 若此时 char_v定义为 char char_v [MAX]且 MAX未知， sizeof(void)都不是正确形式。

三、 sizeof的结果

sizeof操作符的结果类型是 size_t,它在头文件中 typedef为 unsigned int类型。该类型保证能容纳实现所建立的最大对象的字节大小。

1、若操作数具有类型 char、 unsigned char或 signed char，其结果等于 1。

ANSI C正式规定字符类型为 1字节。

2、 int、 unsigned int 、short int、 unsigned short 、long int 、unsigned long 、 float、 double、 long double类型的 sizeof 在 ANSI C中没有具体规定，大小依赖于实现，一般可能分别为 2、 2、 2、 2、 4、 4、 4、 8、 10。

3、当操作数是指针时， sizeof依赖于编译器。例如 Microsoft C/C++7.0中，near类指针字节数为 2， far、 huge类指针字节数为 4。一般 Unix的指针字节数为 4。

4、当操作数具有数组类型时，其结果是数组的总字节数。

5、联合类型操作数的 sizeof是其最大字节成员的字节数。结构类型操作数的 sizeof是这种类型对象的总字节数，包括任何垫补在内。

让我们看如下结构：

struct {char b; double x;} a;

在某些机器上 sizeof（a）=12，而一般 sizeof（char）+sizeof（double）=9。

这是因为编译器在考虑对齐问题时，在结构中插入空位以控制各成员对象的地址对齐。如 double类型的结构成员 x要放在被 4整除的地址。

6、如果操作数是函数中的数组形参或函数类型的形参， sizeof给出其指针的大小。

四、 sizeof与其他操作符的关系

sizeof的优先级为 2级，比 /、 %等 3级运算符优先级高。它可以与其他操

作符一起组成表达式。如 i*sizeof（int）；其中 i为 int类型变量。

五、 sizeof的主要用途

1、 sizeof操作符的一个主要用途是与存储分配和 I/O系统那样的例程进行通信。例如：

void *malloc（size_t size）；

size_t fread(void * ptr,size_t size,size_t mmemb,FILE * stream)。

2、 sizeof的另一个的主要用途是计算数组中元素的个数。例如：

void * memset（void * s,int c,sizeof(s)）。

六、建议

由于操作数的字节数在实现时可能出现变化，建议在涉及到操作数字节大小时用 sizeof来代替常量计算。

1．用预处理指令#define 声明一个常数，

用以表明 1 年中有多少秒（忽略闰年问题）

#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL

我在这想看到几件事情：

; #define 语法的基本知识（例如：不能以分号结束，括号的使用，等等）

; 懂得预处理器将为你计算常数表达式的值，因此，直接写出你是如何计算一年中有多少秒而不是计算出实际的值，是更清晰而没有代价的。

; 意识到这个表达式将使一个 16 位机的整型数溢出-因此要用到长整型符号 L,告诉编译器这个常数是的长整型数。

; 如果你在你的表达式中用到 UL（表示无符号长整型），那么你有了一个好的起点。记住，第一印象很重要。

2．写一个"标准"宏 MIN

这个宏输入两个参数并返回较小的一个。

```
#define MIN(A,B)  ((A)  <= (B) ? (A) : (B))
```

这个测试是为下面的目的而设的：

; 标识#define 在宏中应用的基本知识。这是很重要的，因为直到嵌入(inline)操作符变为标准 C 的一部分，宏是方便产生嵌入代码的唯一方法，对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

; 三重条件操作符的知识。这个操作符存在 C 语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码，了解这个用法是很重要的。

; 懂得在宏中小心地把参数用括号括起来

; 我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

```
least = MIN(*p++, b);
```

3. 预处理器标识#error 的目的是什么？

如果你不知道答案，请看参考文献 1。这问题对区分一个正常的伙计和一个书呆子是很有用的。只有书呆子才会读 C 语言课本的附录去找出象这种问题的答案。当然如果你不是在找一个书呆子，那么应试者最好希望自己不要知道答案。

4.死循环（Infinite loops）

嵌入式系统中经常要用到无限循环，你怎么样用 C 编写死循环呢？

这个问题用几个解决方案。我首选的方案是：

```
while(1)
{
?}
```

一些程序员更喜欢如下方案：

```
for(;
{
?}
```

这个实现方式让我为难，因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这

个作为方案，我将用这个作为一个机会去探究他们这样做的基本原理。如果他们的基本答案是："我被教着这样做，但从没有想到过为什么。"这会给我留下一个坏印象。

第三个方案是用 goto

```
Loop:
...
goto Loop;
```

应试者如给出上面的方案，这说明或者他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。

5.数据声明（Data declarations）

用变量 a 给出下面的定义

a) 一个整型数（An integer）

b)一个指向整型数的指针（ A pointer to an integer）

c)一个指向指针的的指针，它指向的指针是指向一个整型数（ A pointer to a pointer to an intege）r

d)一个有 10 个整型数的数组（ An array of 10 integers）

e) 一个有 10 个指针的数组，该指针是指向一个整型数的。（An array of 10 pointers to integers）

f) 一个指向有 10 个整型数数组的指针（ A pointer to an array of 10 integers）

g) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer）

h)一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数（ An array of ten pointers to functions that take an integer argument and return an integer ）

答案是：

```
a) int a; // An integer
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer
```

人们经常声称这里有几个问题是那种要翻一下书 才能回答的问题，我同意这种说法。当我写这篇文章时，为了确定语法的正确性，我的确查了一下书。但是当我被面试的时候，我期望被问到这个问题（或者相近的 问题）。因为在被面试的这段时间里，我确定我知道这个问题的答案。应试者如果不知道所有的答案（或至少大部分答案），那么也就没有为这次面试做准备，如果 该面试者没有为这次面试做准备，那么他又能为什么出准备呢？

6. 关键字 static 的作用是什么？

这个简单的问题很少有人能回答完全。在 C 语言中，关键字 static 有三个明显的作用：

; 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。

；在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。
；在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。
大多数应试者能正确回答第一部分，一部分能正确回答第二部分，同是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数据和代码范围的好处和重要性。

7．关键字 const 有什么含意？

我只要一听到被面试者说："const 意味着常数"，我就知道我正在和一个业余者打交道。去年 Dan Saks 已经在他的文章里完全概括了 const 的所有用法，因此 ESP(译者：Embedded Systems Programming)的每一位读者应该非常熟悉 const 能做什么和不能做什么.如果你从没有读到那篇文章，只要能说出 const 意味着"只读"就可 以了。尽管这个答案不是完全的答案，但我接受它作为一个正确的答案。（如果你想知道更详细的答案，仔细读一下 Saks 的文章吧。）

如果应试者能正确回答这个问题，我将问他一个附加的问题：

下面的声明都是什么意思？

```
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
```

/******/

前两个的作用是一样，a 是一个常整型数。第三个意味着 a 是一个指向常整型数的指针（也就是，整型数是不可修改的，但指针可以）。第四个意思 a 是一个指向整 型数的常指针（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的）。最后一个意味着 a 是一个指向常整型数的常指针（也就是说，指针指向的整型 数是不可修改的，同时指针也是不可修改的）。如果应试者能正确回答这些问题，那么他就给我留下了一个好印象。顺带提一句，也许你可能会问，即使不用关键字 const，也还是能很容易写出功能正确的程序，那么我为什么还要如此看重关键字 const 呢？我也如下的几下理由：
；关键字 const 的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理 其它人留下的垃圾，你就会很快学会感谢这点多余的信息。（当然，懂得用 const 的程序员很少会留下的垃圾让别人来清理的。）
；通过给优化器一些附加的信息，使用关键字 const 也许能产生更紧凑的代码。
；合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少 bug 的出现。

8. 关键字 volatile 有什么含意?并给出三个不同的例子。

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取

这个变量的值，而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：
; 并行设备的硬件寄存器（如：状态寄存器）
; 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)
; 多线程应用中被几个任务共享的变量
回答不出这个问题的人是不会被雇佣的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等等打交道，所有这些都要求用到 volatile 变量。不懂得 volatile 的内容将会带来灾难。

假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，看一下这家伙是不是直正懂得 volatile 完全的重要性。
; 一个参数既可以是 const 还可以是 volatile 吗？解释为什么。
; 一个指针可以是 volatile 吗？解释为什么。
; 下面的函数有什么错误： int square(volatile int *ptr)

```
{
return *ptr * *ptr;
}
```

下面是答案：
; 是的。一个例子是只读的状态寄存器。它是 volatile 因为它可能被意想不到地改变。它是 const 因为程序不应该试图去修改它。
; 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个 buffer 的指针时。
; 这段代码有点变态。这段代码的目的是用来返指针*ptr 指向值的平方，但是，由于*ptr 指向一个 volatile 型参数，编译器将产生类似下面的代码：

```
int square(volatile int *ptr)
{
int a,b;
a = *ptr;
b = *ptr;
return a * b;
}
```

由于*ptr 的值可能被意想不到地该变，因此 a 和 b 可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

```
long square(volatile int *ptr)
{
int a;
a = *ptr;
return a * a;
}
```

9.位操作（Bit manipulation）

嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清除 a 的 bit 3。在以上两个操作中，要保持其它位不变。

对这个问题有三种基本的反应
; 不知道如何下手。该被面者从没做过任何嵌入式系统的工作。

；用 bit fields。Bit fields 是被扔到 C 语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了的你的代码是不可重用的。我最近不幸看到 Infineon 为其较复杂的通信芯片写的驱动程序，它用到了 bit fields 因此完全对我无用，因为我的编译器用其它的方式来实现 bit fields 的。从道德讲：永远不要让一个非嵌入式的家伙粘实际硬件的边。
；用 #defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。最佳的解决方案如下： #define BIT3 (0x1 << 3)

```
static int a;
void set_bit3(void) {
a |= BIT3;
}
void clear_bit3(void) {
a &= ~BIT3;
}
```

一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数，这也是可以接受的。我希望看到几个要点：说明常数、|=和&=~操作。

10.访问固定的内存位置（Accessing fixed memory locations）

嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为 0x67a9 的整型变量的值为 0xaa66。编译器是一个纯粹的 ANSI 编译器。写代码去完成这一任务。

这一问题测试你是否知道**为了访问一绝对地址把一个整型数强制转换（typecast）为一指针**是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下：

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
A more obscure approach is:
一个较晦涩的方法是：
*(int * const)(0x67a9) = 0xaa55;
```

即使你的品味更接近第二种方案，但我建议你在面试时使用第一种方案。

11.中断（Interrupts）

中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展-让标准 C 支持中断。具代表事实是，产生了一个新的关键字 __interrupt。下面的代码就使用了__interrupt 关键字去定义了一个中断服务子程序(ISR)，请评论一下这段代码的。

```
__interrupt double compute_area (double radius)
{
double area = PI * radius * radius;
printf("\nArea = %f", area);
return area;
}
```

这个函数有太多的错误了，以至让人不知从何说起了：
; ISR 不能返回一个值。如果你不懂这个，那么你不会被雇用的。
; ISR 不能传递参数。如果你没有看到这一点，你被雇用的机会等同第一项。
; 在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈，有些处理器/编译器就是不允许在 ISR 中做浮点运算。此外，ISR 应该是短而有效率的，在 ISR 中做浮点运算是不明智的。
; 与第三点一脉相承，printf()经常有重入和性能上的问题。如果你丢掉了第三和第四点，我不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。
*****

12．下面的代码输出是什么，为什么？

```
void foo(void)
{
unsigned int a = 6;
int b = -20;
(a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

这 个问题测试你是否懂得 C 语言中的整数自动转换原则，我发现有些开发者懂得极少这些东西。不管如何，这无符号整型问题的答案是输出是 ">6"。原因是当**表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型**。因此-20 变成了一个非常大的正整数，所以该表达式 计算出的结果大于 6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是丰常重要的。如果你答错了这个问题，你也就到了得不到这份工作的边缘。

13. 评价下面的代码片断：

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

对于一个 int 型不是 16 位的处理器为说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ~0;
```

这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里，好的嵌入式程序员非常准确地明白硬件的细节和它的局限，然而 PC 机程序往往把硬件作为一个无法避免的烦恼。

到了这个阶段，应试者或者完全垂头丧气了或者信心满满志在必得。如果显然应试者不是很好，那么这个测试就在这里结束了。但如果显然应试者做得不错，那么我 就扔出下面的追加问题，这些问题是比较难的，我想仅仅非常优秀的应试者能做得不错。提出这些问题，我希望更多看到应试者应付问题的方法，而不是答案。不管 如何，你就当是这个娱乐吧...

14.动态内存分配（Dynamic memory allocation）

尽管不像非嵌入式计算机那么常见，嵌入式系统还是有从堆（heap）中动态分配内存的过程的。那么嵌入式系统中，动态分配内存可能发生的问题是什么？

这里，我期望应试者能提到内存碎片，碎片收集的问题，变量的持行时间等等。这个主题

已经在 ESP 杂志中被广泛地讨论过了（主要是 P.J. Plauger, 他的解释远远超过我这里能提到的任何解释），所有回过头看一下这些杂志吧！让应试者进入一种虚假的安全感觉后，我拿出这么一个小节目：

下面的代码片段的输出是什么，为什么？

```
char *ptr;
if ((ptr = (char *)malloc(0)) ==
NULL)
else
    puts("Got a null pointer");

puts("Got a valid pointer");
```

这是一个有趣的问题。最近在我的一个同事不经意把 0 值传给了函数 malloc，得到了一个合法的指针之后，我才想到这个问题。这就是上面的代码，该代码的输出是"Got a valid pointer"。我用这个来开始讨论这样的一问题，看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要，但解决问题的方法和你做决定的基本原理更重要些。

15 Typedef

Typedef 在 C 语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如，思考一下下面的例子：

```
#define dPS struct s *
typedef struct s * tPS;
```

以上两种情况的意图都是要定义 dPS 和 tPS 作为一个指向结构 s 指针。哪种方法更好呢？（如果有的话）为什么？

这是一个非常微妙的问题，任何人答对这个问题（正当的原因）是应当被恭喜的。答案是：typedef 更好。思考下面的例子：

```
dPS p1,p2;
tPS p3,p4;
```

第一个扩展为

```
struct s * p1, p2;
```

上面的代码定义 p1 为一个指向结构的指，p2 为一个实际的结构，这也许不是你想要的。第二个例子正确地定义了 p3 和 p4 两个指针。

16．晦涩的语法

C 语言同意一些令人震惊的结构,下面的结构是合法的吗，如果是它做些什么？

```
int a = 5, b = 7, c;
c = a++ +b;
```

这个问题将做为这个测验的一个愉快的结尾。不管你相不相信，上面的例子是完全合乎语法的。问题是编译器如何处理它？水平不高的编译作者实际上会争论这个问题，根据最处理原则，编译器应当能处理尽可能所有合法的用法。因此，上面的代码被处理成：

```
c = a++ + b;
```

因此，这段代码持行后 a = 6, b = 7, c = 12。

如果你知道答案，或猜出正确答案，做得好。如果你不知道答案，我也不把这个当作问题。我发现这个问题的最大好处是这是一个关于代码编写风格，代码的可读性，代码的可修改性的好的话题。

C程序设计的常用算法

一、求两个整数的最大公约数、最小公倍数

分析：求最大公约数的算法思想：(最小公倍数=两个整数之积/最大公约数)

(1) 对于已知两数 m，n，使得 m>n；

(2) m 除以 n 得余数 r；

(3) 若 r=0，则 n 为求得的最大公约数，算法结束；否则执行(4)；

(4) m←n，n←r，再重复执行(2)。

例如: 求 m=14 ,n=6 的最大公约数. m n r

14 6 2

6 2 0

```
void main()
{
   int a,r,n,m,t;
   printf("please input two numbers:\n");
   scanf("%d,%d",&m,&n);
   a=n*m;
   if (m<n)
   {
      t=n;
      n=m;
      m=t;
   }
   r=m%n;
   while (r!=0)
   {
      m=n;
      n=r;
      r=m%n;
   }
   printf("最大公约数:%d\n",n);
   printf("最小公倍数:%d\n",a/n);
}
```

二、判断素数

只能被 1 或本身整除的数称为素数 基本思想：把 m 作为被除数，将 2 到(int)sqrt(m)之间所有的整数作为除数，如果都除不尽，m 就是素数，否则就不是。（可用以下程序段实现）

```
void main()
{
   int m,i,k;
   printf("please input a number:\n");
   scanf("%d",&m);
   k=sqrt(m);
```

```
    for(i=2;i<k;i++)
    {
        if(m%i==0) break;
    }
    if(i>=k)
        printf("该数是素数");
    else
        printf("该数不是素数");
}
```

三、排序算法

1.冒泡法：

冒泡法原理详见教材例 7.3，教材 134 页

```
#include <stdio.h>
void BubbleSort(int* pData,int Count)
{
    int iTemp;
    for(int i=1;i<Count;i++)
    {
        for(int j=Count-1;j>=i;j--)
        {
            if(pData[j]<pData[j-1])
            {
                iTemp = pData[j-1];
                pData[j-1] = pData[j];
                pData[j] = iTemp;
            }
        }
    }
}

void main()
{
    int data[] = {10,9,8,7,6,5,4};
    BubbleSort(data,7);
    for (int i=0;i<7;i++)
    {
        printf(“%d\n”, data[i]);
    }
}
```

2.比较交换法

比较交换法的程序最清晰简单，每次用当前的元素一一的同其后的元素比较并交换

```
#include <stdio.h>
void ExchangeSort(int* pData,int Count)
{
   int iTemp;
   for(int i=0;i<Count-1;i++)
   {
      for(int j=i+1;j<Count;j++)
      {
         if(pData[j]<pData[i])
         {
            iTemp = pData[i];
            pData[i] = pData[j];
            pData[j] = iTemp;
         }
      }
   }
}

void main()
{
   int data[] = {10,9,8,7,6,5,4};
   ExchangeSort(data,7);
   for (int i=0;i<7;i++)
   {
      printf("%d\n", data[i]);
      }
}
```

3.选择法：

选择法原理详见教材例 10.9，教材 241 页

此处为另一种写法，原理相同

```
#include <stdio.h>
void SelectSort(int* pData,int Count)
{
   int iTemp;
   int iPos;
   for(int i=0;i<Count-1;i++)
   {
      iTemp = pData[i];
      iPos = i;
      for(int j=i+1;j<Count;j++)
      {
```

```
            if(pData[j]<iTemp)
            {
                iTemp = pData[j];
                iPos = j;
            }
        }
        pData[iPos] = pData[i];
        pData[i] = iTemp;
    }
}

void main()
{
    int data[] = {10,9,8,7,6,5,4};
    SelectSort(data,7);
    for (int i=0;i<7;i++)
    {
        printf("%d\n", data[i]);
        }
}
```

4. 插入法

插入法较为复杂，它的基本工作原理是抽出牌，在前面的牌中寻找相应的位置插入，然后继续下一张

```
#include <stdio.h>
void InsertSort(int* pData,int Count)
{
    int iTemp;
    int iPos;
    for(int i=1;i<Count;i++)
    {
        iTemp = pData[i];
        iPos = i-1;
        while((iPos>=0) && (iTemp<pData[iPos]))
        {
            pData[iPos+1] = pData[iPos];
            iPos--;
        }
        pData[iPos+1] = iTemp;
    }
}

void main()
{
```

```
   int data[] = {10,9,8,7,6,5,4};
   InsertSort(data,7);
   for (int i=0;i<7;i++)
   {
   printf("%d\n", data[i]);
      }
}
```

四、查找法

1. 顺序查找法（在一列数中查找某数 x）

基本思想：一列数放在数组 a[1]---a[n]中，待查找的数放在 x 中，把 x与 a数组中的元素从头到尾一一进行比较查找。用变量 p表示 a数组元素下标， p初值为 1，使 x与 a[p]比较，如果 x不等于 a[p]，则使 p=p+ 1，不断重复这个过程；一旦 x等于 a[p]则退出循环；另外，如果 p大于数组长度，循环也应该停止。

```
void main()
{
   int a[10],p,x,i;
   printf("please input the array:\n");
   for(i=0;i<10;i++)
   {
      scanf("%d",&a);
   }
   printf("please input the number you want find:\n");
   scanf("%d",&x);
   printf("\n");
   p=0;
   while(x!=a[p]&&p<10)
      p++;
   if(p>=10)
      printf("the number is not found!\n");
   else
      printf("the number is found the no%d!\n",p);
}
```

2．折半查找法（只能对有序数列进行查找）

基本思想：设 n 个有序数（从小到大）存放在数组中，要查找的数为 x。用变量 bot、top、mid 分别表示查找数据范围的底部(数组下界)、顶部(数组的上界)和中间，mid=(top+bot)/2，折半查找的算法如下：

（1）x=a(mid)，则已找到退出循环，否则进行下面的判断；

（2）x<a(mid)，x 必定落在 bot 和 mid-1 的范围之内，即 top=mid-1；

（3）x>a(mid)，x 必定落在 mid+1 和 top 的范围之内，即 bot=mid+1；

（4）在确定了新的查找范围后，重复进行以上比较，直到找到或者 bot<=top。

将上面的算法写成如下程序：

```
void main()
{
   int a[10],mid,bot,top,x,i,find;
   printf("please input the array:\n");
   for(i=0;i<10;i++)
      scanf("%d",&a);

   printf("please input the number you want find:\n");
   scanf("%d",&x);
   printf("\n");
   bot=0;
   top=9;
   find=0;
   while(bot<top&&find==0)
   {
      mid=(top+bot)/2;
      if(x==a[mid])
      {
            find=1;break;
      }
      else if(x<a[mid])
            top=mid-1;
      else
            bot=mid+1;
   }
   if (find==1)
      printf("the number is found the no%d!\n",mid);
   else
      printf("the number is not found!\n");
}
```

附录 D

标准库函数

本附录描述了标准C支持的库函数[①]。使用此附录时，请记住下列要点。

- 为了简洁清楚，这里删除了一些细节。如果想看全部内容，请参考标准。本书的其他地方已经对一些函数（特别是printf函数、scanf函数以及它们的变异函数）进行了详细介绍，所以这里只对这类函数做简短的描述。为了获得关于某个函数更详细的信息（包括如何使用这个函数的示例），请见函数描述右下角用楷体列出的节号。
- 每个函数描述结尾都有其他与之相关函数的列表。**相似函数**非常接近于正在描述的函数。**相关函数**经常会和在描述的函数联合使用。（例如，calloc函数和realloc函数与malloc函数“类似”，而free函数则与malloc函数“相关”。）**也可参见**的函数和在描述的函数没有紧密联系，但是却可能有影响。
- 如果把函数行为的某些方面描述为由**实现定义的**，那么这就意味着此函数依赖于C库的实现方式。函数将始终行为一致，但是结果却可能会由于系统的不同而千差万别。（换句话说，请参考手册了解可能发生的问题。）另一方面，**未定义的**行为是一个不好的消息：不但函数的行为可能会因系统不同而不同，而且程序也可能会行为异常甚至崩溃。
- <math.h>中许多函数的描述提到了**定义域错误**和**取值范围错误**。在本附录的末尾对这两种错误进行了定义。

601

- 下列库函数的行为是会受到当前地区影响的：
  - 字符处理函数（除了isdigit函数和isxdigit函数）。
  - 格式化输入/输出函数。
  - 多字节字符和字符串函数。
  - 字符串转换函数。
  - Strcoll函数、strftime函数和strxfrm函数。

  例如，isalpha函数实际上检测字符是否在a到z之间或者在A到Z之间。在某些区域内也把其他字符看成是字母次序的。本附录描述了在"C"（默认的）地区内库函数的行为。
- 一些函数实际上是宏。然而，这些宏的用法和函数完全一样，所以这里不对它们区别对待。

---

abort 异常终止程序 <stdlib.h>

```
void abort(void);
```

产生SIGABRT信号。如果无法捕获信号（或者如果信号处理函数返回），那么程序会异常终止，并且返回由实现定义的代码来说明不成功的终止。是否清洗输出缓冲区，是否关闭打开的流，以及是否移除临时文件都是由实现定义的。

相似函数 exit函数、raise函数

相关函数 assert函数、signal函数

也可参见 atexit函数 26.2节

---

abs 整数的绝对值 <stdlib.h>

```
int abs(int j);
```

返回 整数j的绝对值。如果不能表示j的绝对值，那么函数的行为是未定义的。

---

① 这些材料经ANSI许可改编自American National Standards Institude ANSI/ISO 9899©1990。这个标准的副本可从ANSI购买（ANSI, 11 West 42nd Street, New York, NY 10036）。

相似函数　fabs函数、labs函数　26.2节

---

acos　反余弦　<math.h>

```
double acos(double x);
```

返回　x的反余弦值。返回值的范围在0到π之间。如果x的值不在–1到+1之间，那么就会发生定义域错误。

相关函数　asin函数、atan函数、atan2函数、cos函数、sin函数、tan函数　23.3节　602

---

asctime　把日期和时间转换成ASCII码　<time.h>

```
char *asctime(const struct tm *timeptr);
```

返回　指向以空字符结尾的字符串的指针，其格式如下所示：

```
Mon Jul 15 12:30:45 1996\n
```

此格式的构造来源于timeptr指向的结构中的分解时间。

相似函数　ctime函数、strftime函数

相关函数　diffime函数、gmtime函数、localtime函数、mktime函数、time函数　26.3节

---

asin　反正弦　<math.h>

```
double asin(double x);
```

返回　x的反正弦值。返回值的范围在-π/2到π/2之间。如果x的值不在-1到+1之间，那么就会发生定义域错误。

相关函数　acos函数、atan函数、atan2函数、cos函数、sin函数、tan函数　23.3节

---

asserrt　诊断表达式的真值　<assert.h>

```
void assert(int expression);
```

如果expression的值非零，那么assert函数什么也不做。如果expression的值为零，那么assert函数向stderr写信息（说明expression的文本，含有assert函数的源文件名，以及assert函数的行数），然后通过调用abort函数终止程序。为了使assert函数无效，要在包含<assert.h>之前定义宏NDEBUG。

相关函数　abort函数　24.1节

---

atan　反正切　<math.h>

```
double atan(double x);
```

返回　x的反正切值。返回值的范围在–π/2到π/2之间。

相似函数　atan2函数

相关函数　acos函数、asin函数、cos函数、sin函数、tan函数　23.3节

---

atan2　商的反正切　<math.h>

```
double atan2(double y, double x);
```

返回　y/x的反正切值。返回值的范围在–π到π之间。如果x和y的值都为零，那么就会发生定义域错误。

相似函数　atan函数

相关函数　acos函数、asin函数、cos函数、sin函数、tan函数　23.3节　603

---

atexit　在程序退出处注册要调用的函数　<stdlib.h>

```
int atexit(void (*func)(void));
```

注册由func指向的函数作为终止函数。如果程序正常终止（通过return或exit，而不是abort），那么将调用函数。可以重复调用atexit函数来注册多个终止函数。最后一个注册的函数将是在终止前第一个被调用的函数。

返回　如果成功，返回零。如果不成功，则返回非零（达到由实现定义的限制）。

相关函数　exit函数

也可参见　abort函数　26.2节

---

atof　把字符串转换成浮点数　<stdlib.h>

```
double atof(const char *nptr);
```

返回 对应字符串最长初始部分的double型值，此字符串是由nptr指向的，且字符串最长初始部分具有浮点数的格式。如果无法表示此数，那么函数的行为将是未定义的。

相似函数 strtod函数

相关函数 atoi函数、atol函数

也可参见 strtol函数、strtoul函数　26.2节

---

atoi 把字符串转换成整数　<stdlib.h>

```
int atoi(const char *nptr);
```

返回 对应字符串最长初始部分的整数，此字符串是由nptr指向的，且字符串最长初始部分具有整数的格式。如果无法表示此数，那么函数的行为将是未定义的。

相似函数 atol函数、strtol函数、strtoul函数

相关函数 atof函数

也可参见 strtod函数　26.2节

---

atol 把字符串转换成长整数　<stdlib.h>

```
long int atol(const char *nptr);
```

返回 对应字符串最长初始部分的长整数，此字符串是由nptr指向的，且字符串最长初始部分具有整数的格式。如果无法表示此数，那么函数的行为将是未定义的。

相似函数 atoi函数、strtol函数、strtoul函数

相关函数 atof函数

604 **也可参见** strtod函数　26.2节

---

bsearch 二分检索　<stdlib.h>

```
void *bsearch(const void *key, const void *base,
              size_t memb, size_t size,
              int (*compar)(const void *,
                            const void *));
```

在有序数组中搜索由key指向的值。其中，数组存储在base地址上，且此数组有nmemb个元素，每个元素大小为size个字节。compar指向“比较函数”。换句话说当传递指向关键字的指针和数组元素时，比较函数必须返回负整数、零或正整数，这主要依赖于关键字是小于、等于还是大于数组元素。

返回 指向数组元素的指针，此数组元素是用来测试是否等于关键字的。如果没有找到关键字，那么返回空指针。

相关函数 qsort函数　26.2节

---

calloc 分配并清除内存块　<stdlib.h>

```
void *calloc(size_t nmemb, size_t size);
```

为带有nmemb个元素的数组分配内存块，其中每个数组元素占size个字节。通过设置所有位为零来清除内存块。

返回 指向内存块开始处的指针。如果不能分配所要求大小的内存块，那么返回空指针。

相似函数 malloc函数、realloc函数

相关函数 free函数　17.3节

---

ceil 上整数　<math.h>

```
double ceil(double x);
```

返回 大于或等于x的最小整数。

相似函数 floor函数　23.3节

---

clearerr 清除流错误　<stdio.h>

```
void clearerr(FILE *stream);
```

为stream指向的流清除文件尾指示器和错误指示器。

605 **相关函数** feof函数、ferror函数、rewind函数　22.3节

clock　处理器时钟　<time.h>

```
clock_t clock(void);
```

返回　从程序开始执行起所经过的处理器时间（按照“时钟嘀嗒”来衡量的）。（用CLOCKS_PER_SEC除以此时间来转换成秒。）如果时间无效或者无法表示，那么返回(clock_t)-1。

相似函数　time函数

也可参见　difftime函数　26.3节

cos　余弦　<math.h>

```
double cos(double x);
```

返回　x的余弦值（按照弧度衡量的）。

也可参见　acos函数、asin函数、atan函数、atan2函数、sin函数、tan函数　23.3节

cosh　双曲余弦　<math.h>

```
double cosh(double x);
```

返回　x的双曲余弦值。如果x的数过大，那么可能会发生取值范围错误。

相关函数　sinh函数、tanh函数

也可参见　acos函数、asin函数、atan函数、atan2函数、cos函数、sin函数、tan函数　23.3节

ctime　把日期和时间转换成字符串　<time.h>

```
char *ctime(const time_t *timer);
```

返回　指向字符串的指针，此字符串描述了本地时间，此时间等价于timer指向的日历时间。等价于asctime(localtime(timer))。

相似函数　asctime函数、strftime函数

相关函数　difftime函数、gmtime函数、localtime函数、mktime函数、time函数　26.3节

difftime　时间差　<time.h>

```
double difftime(time_t time1, time_t time0);
```

返回　time0（较早的时间）和time1之间的差值，此值按秒来衡量。

相关函数　asctime函数、ctime函数、gmtime函数、localtime函数、mktime函数、strftime函数、time函数

也可参见　clock函数　26.3节

div　整数除法　<stdlib.h>

```
div_t div(int numer, int denom);
```

返回　含有quot（numer除以denom时的商）和rem（余数）的结构。如果无法表示结果，那么函数的行为是未定义的。

相似函数　ldiv函数　26.2节

606

exit　退出程序　<stdlib.h>

```
void exit(int status);
```

调用所有用atexit函数注册的函数，清洗全部输出缓冲区，关闭所有打开的流，移除任何由tmpfile产生的文件，并终止程序。status的值说明程序是否正常终止。status唯一可移植的值是0和EXIT_SUCCESS（两者都说明成功终止）以及EXIT_FAILURE（不成功的终止）。status的其他值都是由实现定义的。

相似函数　abort函数

相关函数　atexit函数　9.5节、26.2节

exp　指数　<math.h>

```
double exp(double x);
```

返回　*e*的x次幂的值（即e^x）。如果x的数过大，那么可能会发生取值范围错误。

相似函数　pow函数

相关函数　log函数

也可参见　log10函数　23.3节

fabs　浮点数的绝对值　<math.h>

```
double fabs(double x);
```

返回　x的绝对值。

相似函数　abs函数、labs函数　23.3节

fclose　关闭文件　<stdio.h>

```
int fclose(FILE *stream);
```

关闭由stream指向的流。清洗保留在流缓冲区内的任何未写的输出。如果是自动分配，那么就释放缓冲区。

返回　如果成功，就返回零。如果检测到错误，就返回EOF。

相关函数　fopen函数、freopen函数

607　也可参见　fflush函数　22.2节

feof　检测文件末尾　<stdio.h>

```
int feof(FILE *stream);
```

返回　如果为stream指向的流设置了文件尾指示器，那么返回非零值。否则返回零。

相似函数　ferror函数

相关函数　clearerr函数、fseek函数、rewind函数　22.3节

ferror　检测文件错误　<stdio.h>

```
int ferror(FILE *stream);
```

返回　如果为stream指向的流设置了文件错误指示器，那么返回非零值。否则返回零。

相似函数　feof函数

相关函数　clearerr函数、rewind函数　22.3节

fflush　清洗文件缓冲区　<stdio.h>

```
int fflush(FILE *stream);
```

把任何未写入的数据写到和stream相关的缓冲区中，其中stream指向用于输出或更新的已打开的流。如果stream是空指针，那么fflush函数清洗存储在缓冲区中的所有未写入的流。

返回　如果成功就返回零。如果检测到错误，就返回EOF。

也可参见　fclose函数、setbuf函数、setvbuf函数　22.2节

fgetc　从文件中读取字符　<stdio.h>

```
int fgetc(FILE *stream);
```

从stream指向的流中读取字符。

返回　读到的字符。如果fgetc函数遇到流的末尾，则设置流的文件尾指示器并且返回EOF。如果读取发生错误，fgetc函数设置流的错误指示器并且返回EOF。

相似函数　getc函数、getchar函数

相关函数　fputc函数、putc函数、ungetc函数

也可参见　putchar函数　22.4节

fgetpos　获得文件位置　<stdio.h>

608

```
int fgetpos(FILE *stream, fpos_t *pos);
```

把stream指向的流的当前位置存储到pos指向的对象中。

返回　如果成功就返回零。如果调用失败，则返回非零值，并且把由实现定义的错误码存储到errno中。

相似函数　ftell函数

相关函数　fsetpos函数

也可参见　fseek函数、rewind函数　22.7节

fgets　从文件中读取字符串　<stdio.h>

```
char *fgets(char *s, int n, FILE *stream);
```

从stream指向的流中读取字符，并且把读入的字符存储到s指向的数组中。遇到第一个换行符已经读取了n-1个字符，或到了文件末尾时，读取操作都会停止。fgets函数会在字符串后添加一个空字符。

返回　s（指向数组的指针，此数组存储着输入）。如果读取操作错误或fgets函数在存储任何字符之前遇到了流的末尾，都会返回空指针。

相似函数　gets函数

相关函数　fputs函数

也可参见　puts函数　22.5节

---

floor　向下取整　<math.h>

```
double floor(double x);
```

返回　小于或等于x的最大整数。

相似函数　ceil函数　23.3节

---

fmod　浮点模数　<math.h>

```
double fmod(double x, double y);
```

返回　x除以y的余数。如果y为零，是发生定义域错误还是fmod函数返回零是由实现定义的。

也可参见　div函数、ldiv函数　23.3节

---

fopen　打开文件　<stdio.h>

```
FILE *fopen(const char *filename, const char *mode);
```

打开文件以及和它相关的流，文件名是由filename指向的。mode说明文件打开的方式。为流清除错误指示器和文件尾指示器。

609

返回　文件指针。在执行下一次关于文件的操作时会用到此指针。如果无法打开文件则返回空指针。

相似函数　freopen函数

相关函数　fclose函数、setbuf函数、setvbuf函数　22.2节

---

fprintf　格式化写文件　<stdio.h>

```
int fprintf(FILE *stream, const char *format, ...);
```

向stream指向的流写输出。format指向的字符串说明了后续参数显示的格式。

返回　写入的字符数量。如果发生错误就返回负值。

相似函数　printf函数、sprintf函数、vfprintf函数、vprintf函数、vsprintf函数

相关函数　fscanf函数

也可参见　scanf函数、sscanf函数　22.3节

---

fputc　向文件写字符　<stdio.h>

```
int fputc(int c, FILE *stream);
```

把字符c写到stream指向的流中。

返回　c（写入的字符）。如果写发生错误，fputc函数会为stream设置错误指示器，并且返回EOF。

相似函数　putc函数、putchar函数

相关函数　fgetc函数、getc函数

也可参见　getchar函数　22.4节

---

fputs　向文件写字符串　<stdio.h>

```
int fputs(const char *s, FILE *stream);
```

把s指向的字符串写到stream指向的流中。

返回　如果成功，返回非负值。如果写发生错误，则返回EOF。

相似函数　puts函数

相关函数　fgets函数

也可参见　gets函数　22.5节

---

fread　从文件读块　<stdio.h>

```
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

试着从stream指向的流中读取nmemb个元素，每个元素大小为size个字节，并且把读入的元素存储到ptr指向的数组中。

610

返回　实际读入的元素（不是字符）数量。如果fread遇到文件末尾或检测到读取错误，那么此数将会小于nmemb。如果nmemb或size为零，则返回值为零。

相关函数　fwrite函数　22.6节

---

free　释放内存块　<stdlib.h>

```
void free (void *ptr);
```

释放地址为ptr的内存块（除非ptr为空指针时调用无效）。块必须通过calloc函数、malloc函数或realloc函数进行分配。

相关函数　calloc函数、malloc函数、realloc函数　17.4节

---

freopen　重新打开文件　<stdio.h>

```
FILE *freopen(const char *filrname, const char *mode,
              FILE *stream);
```

在freopen函数关闭和stream相关的文件后，打开名为filename且与stream相关的文件。Mode参数具有和fopen函数调用中相同的含义。

返回　如果操作成功，返回stream的值。如果无法打开文件则返回空指针。

相似函数　fopen函数

相关函数　fclose函数、setbuf函数、setvbuf函数　22.2节

---

frexp　分解成小数和指数　<math.h>

```
double frexp(double value, int *exp);
```

按照下列形式把value分解成小数部分*f*和指数部分*n*：

$$value = f \times 2^n$$

其中*f*是规范化的，因此$0.5 \leqslant f < 1$或者$f = 0$。把*n*存储在exp指向的整数中。

返回　*f*，即value的小数部分。

相关函数　ldexp函数

也可参见　modf函数　23.3节

---

fscanf　格式化读文件　<stdio.h>

611

```
int fscanf(FILE *stream, const char *format, ...);
```

向stream指向的流读入任意数量的数据项。format指向的字符串说明了读入项的格式。跟在format后边的参数指向数据项存储的位置。

返回　成功读入并且存储的数据项数量。如果发生错误或在可以读数据项前到达了文件末尾，那么就返回EOF。

相似函数　scanf函数、sscanf函数

相关函数　fprintf函数、vfprintf函数

也可参见　printf函数、sprintf函数、vprintf函数、vsprintf函数　22.3节

---

fseek　文件查找　<stdio.h>

```
int fseek(FILE *stream, long int offset, int whence);
```

为stream指向的流改变文件位置指示器。如果whence是SEEk_SET，那么新位置是在文件开始处加上offset个字节。如果whence是SEEK_CUR，那么新位置是在当前位置加上offset个字节。如果whence是SEEK_END，那么新位置是在文件末尾加上offset个字节。对于文本流而言，offset必须是零，或者whence必须是SEEK_SET并且offset的值是由前一次的ftell函数调用获得的。而对于二进制流来说，fseek函数不可以支持whence是SEEK_END的调用。

返回　如果操作成功就返回零。否则返回非零值。

相似函数　fsetpos函数、rewind函数

相关函数　ftell函数

也可参见　fgetpos函数　22.7节

fsetpos　设置文件位置　<stdio.h>

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

根据pos（前一次fgetpos函数调用获得的）指向的值来为stream指向的流设置文件位置指示器。

返回　如果成功就返回零。如果调用失败，返回非零值，并且把由实现定义的错误码存储在errno中。

相似函数　fseek函数、rewind函数

相关函数　fgetpos函数

也可参见　ftell函数　22.7节

---

ftell　确定文件位置　<stdio.h>

```
long int ftell(FILE *stream);
```

返回　返回stream指向的流的当前文件位置指示器。如果调用失败，返回-1L，并且把由实现定义的错误码存储在errno中。 612

相似函数　fgetpos函数

相关函数　fseek函数

也可参见　fsetpos函数、rewind函数　22.7节

---

fwrite　向文件写块　<stdio.h>

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

从ptr指向的数组中写nmemb个元素到stream指向的流中，且每个元素大小为size个字节。

返回　实际写入的元素（不是字符）的数量。如果fwrite函数检测到写错误，则这个数将会小于nmemb。

相关函数　fread函数　22.6节

---

getc　从文件读入字符　<stdio.h>

```
int getc(FILE *stream);
```

从stream指向的流中读入一个字符。注意：getc函数通常是作为宏来实现的。它可能计算stream不只一次。

返回　读入的字符。如果getc函数遇到流的末尾，那么它会设置流的文件尾指示器并且返回EOF。如果读取发生错误，那么getc函数设置流的错误指示器并且返回EOF。

相似函数　fgetc函数、getchar函数

相关函数　fputc函数、putc函数、ungetc函数

也可参见　putchar函数　22.4节

---

getchar　读入字符　<stdio.h>

```
int getchar(void);
```

从stdin流中读入一个字符。注意：getchar函数通常是作为宏来实现的。

返回　读入的字符。如果getc函数遇到输入流的末尾，那么它会设置stdin流的文件尾指示器并且返回EOF。如果读取发生错误，那么getc函数设置stdin流的错误指示器并且返回EOF。

相似函数　fgetc函数、getc函数

相关函数　putchar函数、ungetc函数

也可参见　fputc函数、putc函数　7.3节、22.4节

---

getenv　获取外部环境字符串　<stdlib.h>

```
char *getenv(const char *name);
```

613

为了检查是否有任意字符串匹配name指向的字符串，搜索操作系统的外部环境列表。

返回　与匹配名相关的字符串的指针。如果没有找到匹配则返回空指针。

也可参见　system函数　26.2节

---

gets　读入字符串　<stdio.h>

```
char *gets(char *s);
```

从stdin流中读入多个字符，并且把这些读入的字符存储到s指向的数组中。

返回　s（即存储输入的数组的指针）。如果读取发生错误或gets函数在存储任何字符之前遇到流的末尾，那么返回空指针。
相似函数　fgets函数
相关函数　puts函数
也可参见　fputs函数　13.3节、22.5节

---

gmtime　转换成格林威治标准时间　<time.h>

```
struct tm *gmtime(const time_t *timer);
```

返回　指向结构的指针，此结构包含的分解的UTC（协调世界时间—从前的格林威治时间）值等价于timer指向的日历时间。如果UTC无效，则返回空指针。
相似函数　localtime函数
相关函数　asctime函数、ctime函数、difftime函数、mktime函数、strftime函数、time函数　26.3节

---

isalnum　测试是字母或数字　<ctype.h>

```
int isalnum(int c);
```

返回　如果isalnum是字母或数字，返回非零值；否则返回零。（如果isalph(c)或isdigit(c)为真，则c是字母或数字。）
相关函数　isalpha函数、isdigit函数
也可参见　islower函数、isupper函数　23.4节

---

isalpha　测试字母　<ctype.h>

```
int isalpha(int c);
```

返回　如果isalnum是字母，返回非零值；否则返回零。（如果islower(c)或isupper(c)为真，则c是字母。）
相似函数　islower函数、isupper函数
相关函数　isalnum函数
也可参见　tolower函数、toupper函数　23.4节

614

---

iscntrl　测试控制字符　<ctype.h>

```
int iscntrl(int c);
```

返回　如果c是控制字符，返回非零值；否则返回零。
相关函数　isgraph函数、isprint函数、isspace函数　23.4节

---

isdigit　测试数字　<ctype.h>

```
int isdigit(int c);
```

返回　如果c是数字，返回非零值；否则返回零。
相似函数　isxdigit函数
相关函数　isalnum函数　23.4节

---

isgraph　测试图形字符　<ctype.h>

```
int isgraph(int c);
```

返回　如果c是显示字符（除了空格），返回非零值；否则返回零。
相似函数　isprint函数
相关函数　iscntrl函数、isspace函数　23.4节

---

islower　测试小写字母　<ctype.h>

```
int islower(int c);
```

返回　如果c是小写字母，返回非零值；否则返回零。
相似函数　isalpha函数、isupper函数
相关函数　tolower函数、toupper函数
也可参见　isalnum函数　23.4节

---

isprint　测试显示字符　<ctype.h>

```
int isprint(int c);
```

返回 如果c是显示字符（包括空格），返回非零值；否则返回零。

相似函数 isgraph函数

相关函数 iscntrl函数、isspace函数 23.4节 615

---

ispunct 测试标点字符 <ctype.h>

```
int ispunct(int c);
```

返回 如果c是标点符号字符，返回非零值；否则返回零。除了空格、字母和数字字符以外，所有显示字符都可以看成是标点符号。

也可参见 isalnum函数、isgraph函数、isprint函数 23.4节

---

isspace 测试空白字符 <ctype.h>

```
int isspace(int c);
```

返回 如果c是空白字符，返回非零值；否则返回零。空白字符有空格（' '）、换页符（'\f'）、换行符（'\n'）、回车符（'\r'），横向制表符（'\t'）和纵向制表符（'\v'）。

也可参见 iscntrl函数、isgraph函数、isprint函数 23.4节

---

isupper 测试大写字母 <ctype.h>

```
int isupper(int c);
```

返回 如果c是大写字母，返回非零值；否则返回零。

相似函数 isalpha函数、islower函数

也可参见 tolower函数、toupper函数 23.4节

---

isxdigit 测试十六进制数字 <ctype.h>

```
int isxdigit(int c);
```

返回 如果c是十六进制数字（0-9、a-f、A-F），返回非零值；否则返回零。

相似函数 isdigit函数 23.4节

---

labs 长整数的绝对值 <stdlib.h>

```
longint labs(long int j);
```

返回 j的绝对值。如果不能表示j的绝对值，那么函数的行为是未定义的。

相似函数 abs函数、fabs函数 26.2节

---

ldexp 联合小数和指数 <math.h>

```
double ldexp(double x, int exp);
```

616

返回 $x \times 2^{exp}$的值。可能会发生取值范围错误。

相关函数 frexp函数 23.3节

---

ldiv 长整数除法 <stdlib.h>

```
ldiv_t ldiv(long int numer, long int denom);
```

返回 含有quot（numer除以denom的商）和rem（余数）的结构。如果无法表示结果，那么函数的行为是未定义的。

相似函数 div函数 26.2节

---

localeconv 获取区域转换 <locale.h>

```
struct lconv *localeconv(void);
```

返回 指向结构的指针，此结构含有当前区域信息。

相关函数 setlocale函数 25.1节

---

localtime 转换成区域时间 <time.h>

```
struct tm *localtime(const time_t *timer);
```

返回 指向结构的指针，此结构含有的分解时间等价于timer指向的日历时间。

相似函数 gmtime函数

相关函数 astime函数、ctime函数、difftime函数、mktime函数、strftime函数、time函数 26.3节

log　自然对数　<math.h>

```
double log(double x);
```

返回　基数为*e*的x的对数（即lnx）。如果x是负数，会发生定义域错误；如果x是零，则会发生取值范围错误。

相似函数　log10函数

相关函数　exp函数

也可参见　pow函数　23.3节

log10　常用对数　<math.h>

```
double log10(double x);
```

返回　基数为10的x的对数。如果x是负数，会发生定义域错误；如果x是零，则会发生取值范围错误。

相似函数　log函数

617

也可参见　exp函数、pow函数　23.3节

longjmp　非区域跳转　<setjmp.h>

```
void longjmp(jmp_buf env, int val);
```

恢复存储在env中的外部环境，并且从初始保存env的setjmp调用中返回。如果val非零，它将是setjmp的返回值；如果val为1，则setjmp返回1。

相关函数　setjmp函数

也可参见　signal函数　24.4节

malloc　分配内存块　<stdlib.h>

```
void *malloc(size_t size);
```

分配size个字节的内存块。不清除内存块。

返回　指向内存块开始处的指针。如果无法分配要求尺寸的内存块，那么返回空指针。

相似函数　calloc函数、realloc函数

相关函数　free函数　17.2节

mblen　计算多字节字符的长度　<stdlib.h>

```
int mblen(const char *s, size_t n);
```

如果s是空指针，则初始化移位状态。

返回　如果s是空指针，返回非零值还是零值依赖于多字节字符是否是依赖状态编码。如果s指向空字符则返回零；如果接下来n个或几个字节形成了一个有效的字符，那么返回s指向的多字节字符中的字节数量；否则返回-1。

相关函数　mbtowc函数、wctomb函数

也可参见　mbstowcs函数、setlocale函数、wcstombs函数　25.2节

mbstowcs　把多字节字符串转换成宽字符串　<stdlib.h>

```
size_t mbstowcs(wchar_t *pwcs, const char *s,
                size_t n);
```

把s指向的多字节字符序列转换为宽字符序列，并把不多于n个的编码存储到pwcs指向的数组中。如果遇到空字符则转换结束。空字符会被转换成为零值码。

返回　修改的数组元素的个数，无论如何也不包括终止码。如果遇到无效的多字节字符，则返回（size_t）-1。

618

相关函数　wctombs函数

也可参见　mblen函数、mbtowc函数、setlocale函数、wctomb函数　25.2节

mbtowc　把多字节字符转换成宽字符　<stdlib.h>

```
int mbtowc(wchar_t *pwcs, const char *s, size_t n);
```

如果s是空指针，则初始化移位状态。如果s不是空指针，把s指向的多字节字符转换成宽字符码。最多将检查n个字节的多字节字符。如果多字节字符有效，并且pwc不是空指

针，则把码存储到pwc指向的对象中。

返回　如果s是空指针，则返回非零值还是零值依赖于多字节字符是否是依赖状态编码。如果s指向空字符，则返回零。如果接下来n个或几个字节形成了一个有效的字符，那么返回s指向的多字节字符中的字节数量。如果不是这样，则返回–1。

相关函数　mblen函数、wctomb函数

也可参见　mbstowcs函数、setlocale函数、wcstombs函数　25.2节

---

memchr　搜索内存块字符　<string.h>

```
void *memchr(const void *s, int c, size_t n);
```

返回　指向字符的指针，此字符是s所指向对象的前n个字符中第一个遇到的字符c。如果没有找到c，则返回空指针。

相似函数　strchr函数

也可参见　strpbrk函数、strrchr函数、strstr函数　23.5节

---

memcmp　比较内存块　<string.h>

```
int memcmp(const void *s1, const void *s2, size_t n);
```

返回　负整数、零还是正整数依赖于s1所指向对象的前n个字符是小于、等于还是大于s2所指向对象的前n个字符。

相似函数　strcmp函数、strcoll函数、strncmp函数　23.5节

---

memcpy　复制内存块　<string.h>

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

把s2所指向对象的n个字符复制到s1所指向的对象中。如果对象重叠，则不可能正确地工作。

返回　s1（指向目的的指针）。

相似函数　memmove函数、strcpy函数、strncpy函数　23.5节　619

---

memmove　复制内存块　<string.h>

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

把s2所指向对象的n个字符复制到s1所指向的对象中。如果对象重叠，即使memmove函数比memcpy函数速度慢，但是memmove函数还将正确地工作。

返回　s1（指向目的的指针）。

相似函数　memcpy函数、strcpy函数、strncpy函数　23.5节

---

memset　初始化内存块　<string.h>

```
void *memset(void *s, int c, size_t n);
```

把c存储到s指向的内存块的前n个字符中。

返回　s（指向内存块的指针）。

相似函数　memcpy函数、memmove函数　23.5节

---

mktime　转换成日历时间　<time.h>

```
time_t mktime(struct tm *timeptr);
```

把分解的区域时间（存储在由timeptr指向的结构中）转换成为日历时间。结构的成员不要求一定在合法的取值范围内。而且，会忽略tm_wday（星期的天号）的值和tm_yday（年份的天号）的值。调整其他成员到正确的取值范围内之后，mktime函数把值存储在tm_wday和tm_yday中。

返回　日历时间对应timeptr指向的结构。如果无法表示日历时间，则返回(time_t)-1。

相关函数　astime函数、ctime函数、difftime函数、gmtime函数、localtime函数、strftime函数、time函数　26.3节

---

modf　分解成整数和小数部分　<math.h>

```
double modf(double value, double *iptr);
```

把value分解成整数部分和小数部分。把整数部分存储到iptr指向的double型对象中。

返回　value的小数部分。

620

也可参见　frexp函数　23.3节

perror　显示错误信息　<stdio.h>

```
void perror(const char *s);
```

向Stderr流中写下列信息：

字符串：出错信息

这里的字符串是s所指向的字符串。出错信息是由实现定义的，它与strerror(errno)函数调用返回的信息相匹配。

相关函数　strerror函数　24.2节

pow　幂　<math.h>

```
double pow(double x, double y);
```

返回　x的y次幂。发生定义域错误的情况有（1）当x是负数并且y的值不是整数时；或者（2）当x为零且y是小于或等于零，无法表示结果时。取值范围错误也是可能发生的。

相似函数　exp函数、sqrt函数

也可参见　log函数、log10函数　23.3节

printf　格式化写　<stdio.h>

```
int printf(const char *format, ...);
```

向stdout流写输出。format指向的字符串说明了后续参数显示的格式。

返回　写入的字符数量。如果发生错误就返回负值。

相似函数　fprintf函数、sprintf函数、vfprintf函数、vprintf函数、vsprintf函数

相关函数　scanf函数

也可参见　fscanf函数、sscanf函数　3.1节、22.3节

putc　向文件写字符　<stdio.h>

```
int putc(int c, FILE *stream);
```

把字符c写到stream指向的流中。注意：putc函数通常作为宏来实现的。它可能不只计算stream一次。

返回　c（写入的字符）。如果写发生错误，putc函数会设置流的错误指示器，并且返回EOF。

相似函数　fputc函数、putchar函数

相关函数　fgetc函数、getc函数

621

也可参见　getchar函数　22.4节

putchar　写字符　<stdio.h>

```
int putchar(int c);
```

把字符c写到stdout流中。注意：putchar函数通常作为宏来实现的。

返回　c（写入的字符）。如果写发生错误，putchar函数设置流的错误指示器，并且返回EOF。

相似函数　fputc函数、putc函数

相关函数　getchar函数

也可参见　fgetc函数、getc函数　7.3节、22.4节

puts　写字符串　<stdio.h>

```
int puts(const char *s);
```

把s指向的字符串写到strout流中，然后写一个换行符。

返回　如果成功返回非负值。如果写发生错误则返回EOF。

相似函数　fputs函数

相关函数　gets函数

也可参见　fgets函数　13.3节、22.5节

qsort　排序数组　<stdlib.h>

```
void qsort(void *base, size_t memb, size_t size,
           int (*compar)(const void *, const void *));
```

对base指向的数组排序。数组有nmemb个元素，每个元素大小为size个字节。compar

是指向“比较函数”的指针。当把指向两个数组元素的指针传递过来时，函数依赖于第一个数组元素是否小于、等于或者大于第二个数组元素，应该返回负数、零或正整数。

相关函数　bsearch函数　17.7节、26.2节

---

raise　产生信号　<signal.h>

```
int raise(int sig);
```

产生数为sig的信号。

返回　如果成功，返回零；否则返回非零值。

相似函数　abort函数

相关函数　signal函数　24.3节　622

---

rand　产生伪随机数　<stdlib.h>

```
int rand(void);
```

返回　0到RAND_MAX（包括RAND_MAX在内）之间的伪随机整数。

相关函数　srand函数　26.2节

---

realloc　调整内存块　<stdlib.h>

```
void *realloc(void *ptr, size_t size);
```

假设ptr指向先前由calloc函数、malloc函数或realloc函数获得内存块。realloc函数分配size个字节的内存块，并且如果需要还会复制旧内存块的内容。

返回　指向新内存块开始处的指针。如果无法分配要求尺寸的内存块，那么返回空指针。

相似函数　calloc函数、malloc函数

相关函数　free函数　17.3节

---

remove　移除文件　<stdio.h>

```
int remove(const char *filename);
```

删除文件，此文件名由filename指向。

返回　如果成功就返回零；否则返回非零值。

也可参见　rename函数　22.2节

---

rename　重命名文件　<stdio.h>

```
int rename(const char *old, const char *new);
```

改变文件的名字。old和new指向的字符串分别包含旧的文件名和新的文件名。

返回　如果改名成功就返回零。如果操作失败，就返回非零值（可能因为旧文件目前是打开的）。

也可参见　remove函数　22.2节

---

rewind　返回到文件头　<stdio.h>

```
void rewind(FILE *stream);
```

为stream指向的流设置文件位置指示器到文件的开始处。为流清除错误指示器和文件尾指示器。　623

相似函数　fseek函数、fsetpos函数

相关函数　clearerr函数

也可参见　feof函数、ferror函数、fgetpos函数、ftell函数　22.7节

---

scanf　格式化读　<stdio.h>

```
int scanf(const char *format, ...);
```

从stdin流读取任意数量数据项。format指向的字符串说明了读入项的格式。跟随在format后边的参数指向数据项要存储的地方。

返回　成功读入并且存储的数据项数量。如果发生错误或在可以读入任意数据项之前到达了文件末尾，就返回EOF。

相似函数　fscanf函数、sscanf函数

相关函数　printf函数、vprintf函数

也可参见　fprintf函数、sprintf函数、vfprintf函数、vsprintf函数　3.2节、22.3节

setbuf　设置缓冲区　<stdio.h>

```
void setbuf(FILE *stream, char *buf);
```

如果buf不是空指针，那么setbuf的调用就等价于：

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

（BUFSIZ是在<stdio.h>中定义的宏。）否则，它等价于：

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

相似函数　setvbuf函数
相关函数　fopen函数、freopen函数
也可参见　fflush函数　22.2节

---

setjmp　准备非局部跳转　<setjmp.h>

```
int setjmp(jmp_buf env);
```

为了稍候用于longjmp函数调用，所以把当前外部环境存储到env中。

返回　当直接调用时，返回为零。当从longjmp函数调用中返回时，返回非零值。
相关函数　longjmp函数
也可参见　signal函数　24.4节

---

setlocale　设置地区　<locale.h>

```
char *setlocale(int category, const char *locale);
```

624 设置程序的地区部分。category说明哪部分有效。locale指向表示新地区的字符串。

返回　如果locale是空指针，就返回一个指向与当前地区的category相关的字符串的指针。否则，返回一个指向与新地区的category相关的字符串的指针。如果操作失败，则返回空指针。
相关函数　localeeconv函数　25.1节

---

setvbuf　设置缓冲区　<stdio.h>

```
int setvbuf(FILE *stream, char *buf, int mode,size_t size);
```

改变由stream指向的流的缓冲。mode的值可以是_IOFBF（满缓冲）、_IOLBF（行缓冲）或者_IONBF（不缓冲）。如果buf是空指针，那么若需要则自动分配缓冲区。否则，buf指向用作缓冲区的内存块。size是内存块中字节的数量。注意：必须在打开流之后但对流的任何操作执行之前，调用setvbuf函数。

返回　如果操作成功，就返回零。如果mode无效或者无法满足要求，则返回非零值。
相似函数　setbuf函数
相关函数　fopen函数、freopen函数
也可参见　fflush函数　22.2节

---

signal　安装信号处理函数　<signal.h>

```
void (*signal(int sig, void (*func)(int)))(int);
```

安装func指向的函数作为数sig的信号处理函数。

返回　指向此信号前一个处理函数的指针。如果无法安装处理函数，则返回SIG_ERR。
相关函数　abort函数、raise函数　24.3节

---

sin　正弦　<math.h>

```
double sin(double x);
```

返回　x的正弦值（按照弧度衡量的）。
相关函数　acos函数、asin函数、atan函数、atan2函数、cos函数、tan函数　23.3节

---

sinh　双曲正弦　<math.h>

```
double sinh(double x);
```

625 返回　x的双曲正弦值（按照弧度衡量的）。如果x的数过大，那么可能会发生取值范围错误。
相关函数　cosh函数、tanh函数
也可参见　acos函数、asin函数、atan函数、atan2函数、cos函数、sin函数、tan函数　23.3节

sprintf　格式串写　<stdio.h>

```
int sprintf(char *s, const char *format, ...);
```

与fprintf函数和printf函数很类似，但是sprintf函数不是把字符写入流，而是把字符存储到s指向的数组中。format指向的字符串说明了后续参数显示的格式。在输出的末尾存储一个空字符到数组中。

返回　存储到数组中的字符数量，不计空字符。

相似函数　fprintf函数、printf函数、vfprintf函数、vprintf函数、vsprintf函数

相关函数　sscanf函数

也可参见　fscanf函数、scanf函数　22.8节

---

sqrt　平方根　<math.h>

```
double sqrt(double x);
```

返回　x的平方根。如果x是负数，则会发生定义域错误。

相似函数　pow函数　23.3节

---

srand　启动伪随机数产生器　<stdlib.h>

```
void srand(unsigned int seed);
```

使用seed来初始化由rand函数调用而产生的伪随机序列。

相关函数　rand函数　26.2节

---

sscanf　格式串读　<stdio.h>

```
int sscanf(const char *s, const char *format, ...);
```

与fscanf函数和scanf函数很类似，但是sprintf函数不是从流读取字符，而是从s指向的字符串中读取字符。format指向的字符串说明了读入项的格式。跟随在format后的参数指向数据项要存储的地方。

返回　成功读入并且存储的数据项数量。如果在可以读入任意数据项之前到达了字符串末尾，就返回EOF。

相似函数　fscanf函数、scanf函数

相关函数　sprintf函数、vsprintf函数

也可参见　fprintf函数、printf函数、vfprintf函数、vprintf函数　22.8节　626

---

strcat　字符串的连接　<string.h>

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

把s2指向的字符串连接到s1指向的字符串后边。

返回　s1（指向连接后字符串的指针）。

相似函数　strncat函数　13.5节、23.5节

---

strchr　搜索字符串中字符　<string.h>

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

返回　指向字符的指针，此字符是s所指向的字符串的前n个字符中第一个遇到的字符c。如果没有找到c，则返回空指针。

相似函数　memchr函数

也可参见　strpbrk函数、strrchr函数、strstr函数　23.5节

---

strcmp　比较字符串　<string.h>

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

返回　负数、零还是正整数，依赖于s1所指向的字符串是小于、等于还是大于s2所指的字符串。

相似函数　memcmp函数、strcoll函数、strncmp函数　13.5节、23.5节

---

strcoll　采用指定地区的比较序列进行字符串比较　<string.h>

```
int strcoll(const char *s1, const char *s2);
```

返回　负数、零还是正整数，依赖于s1所指向的字符串是小于、等于还是大于s2所指的字符串。根据当前地区的LC_COLLATE类型规则来执行比较操作。

相似函数　memcmp函数、strcmp函数、strncmp函数
相关函数　strxfrm函数　23.5节

strcpy　字符串复制　<string.h>

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

把s2指向的字符串复制到s1所指向的数组中。

返回　s1（指向目的的指针）。

627

相似函数　memcpy函数、memmove函数、strncpy函数　13.5节、23.5节

strcspn　搜索集合中不在初始范围内的字符串　<string.h>

```
size_t strcspn(const char *s1, const char *s2);
```

返回　最长的初始字符段的长度，此初始字符段由s1指向的，但是不包含s2指向的字符串中的任何字符。

相关函数　strspn函数　23.5节

strerror　把错误数转换成为字符串　<string.h>

```
char *strerror(int errnum);
```

返回　指向字符串的指针，此字符串含有的出错消息对应errnum的值。

相关函数　perror函数　24.2节

strftime　把格式化的日期和时间写到字符串中　<time.h>

```
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

在format指向的字符串的控制下把字符存储到s指向的数组中。格式串可能含有不用改变就进行复制的普通字符和转换说明符，其中转换说明符要用timeptr指向的结构中的值进行替换。maxsize参数限制了可以存储的字符的数量（包括空字符）。

返回　如果要存储的字符数量（包括空字符）超过了maxsize，那么返回零；否则，返回存储的字符数量（不包括空字符）。

相似函数　asctime函数、ctime函数
相关函数　difftime函数、gmtime函数、localtime函数、mktime函数、time函数　26.3节

strlen　字符串长度　<string.h>

```
size_t strlen(const char *s);
```

返回　s指向的字符串长度，不包括空字符。　13.5节、23.5节

strncat　有限制的字符串的连接　<string.h>

628

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

把来自s2所指向的数组的字符连接到s1指向的字符串后边。当遇到空字符或已经复制了n个字符时，复制操作停止。

返回　s1（指向连接后字符串的指针）。

相似函数　strcat函数　23.5节

strncmp　有限制的字符串比较　<string.h>

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

返回　负整数、零还是正整数，依赖于s1所指向的数组的前n个字符是小于、等于还是大于s2所指向的数组的前n个字符。如果在其中某个数组中遇到空字符，比较都会停止。

相似函数　memcmp函数、strcmp函数、strcoll函数　23.5节

strncpy　有限制的字符串复制　<string.h>

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

把s2指向的数组的前n个字符复制到s1所指向的数组中。如果在s2指向的数组中遇到一个空字符，那么strncpy函数为s1指向的数组添加空字符直到写完n个字符的总数量。

返回　s1（指向目的的指针）。

相似函数　memcpy函数、memmove函数、strcpy函数　23.5节

strpbrk　为一组字符之一搜索字符串　<string.h>

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

返回　指向字符的指针，此字符是s1所指向字符串中与s2所指向字符串中的字符相匹配的最左侧的字符。如果没有找到匹配字符，则返回空指针。

也可参见　memchr函数、strchr函数、strrchr函数、strstr函数　23.5节

strrchr　反向搜索字符串中字符　<string.h>

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

返回　指向字符的指针，此字符是s所指向字符串中最后一个遇到的字符c。如果没有找到c，则返回空指针。

也可参见　memchr函数、strchr函数、strpbrk函数、strstr函数　23.5节　629

strspn　搜索集合中在初始范围内的字符串　<string.h>

```
size_t strspn(const char *s1, const char *s2);
```

返回　最长的初始字符段的长度，此初始字符段是由s1指向的且与s2指向的字符串中的全部字符一致的字符段。

相关函数　strcspn函数　23.5节

strstr　搜索子字符串　<string.h>

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

返回　指针，此指针指向s1字符串中的字符第一次出现在s2字符串中的位置。如果没有发现匹配，就返回空指针。

也可参见　memchr函数、strchr函数、strpbrk函数、strrchr函数　23.5节

strtod　把字符串转换成双精度数　<stdlib.h>

```
double strtod(const char *nptr, char **endptr);
```

函数会跳过nptr所指向的字符串中的空白字符，然后把后续字符都转换成为double型的值。如果endptr不是空指针，那么strtod就修改endptr指向的对象，从而使endptr指向第一个剩余字符。如果没有发现double型的值，或者有错误的格式，那么strtod函数把nptr存储到endptr指向的对象中。如果要表示的数过大或者过小，函数就把ERANGE存储到errno中。

返回　转换的数。如果没有转换可以执行，就返回零。如果要表示的数过大，则返回正的或负的HUGE_VAL，这要依赖于数的符号而定。如果要表示的数过小，则返回零。

相似函数　atof函数

相关函数　strtol函数、strtoul函数

也可参见　atoi函数、atol函数　26.2节

strtok　搜索字符串记号　<string.h>

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

在s1指向的字符串中搜索“记号”。组成此记号的字符不在s2指向的字符串中。如果存在记号，则把跟在记号后边的字符变为空字符。如果s1是空指针，则将继续由strtok函数最近一次调用开始的搜索。在上一个记号尾部的空字符之后立即开始搜索。

返回　指向记号的第一个字符的指针。如果没有发现记号，就返回空指针。

也可参见　memchr函数、strchr函数、strpbrk函数、strrchr函数、strstr函数　23.5节　630

strtol　把字符串转换成长整数　<stdlib.h>

```
long int strtol(const char *nptr, char **endptr,
                int base);
```

函数跳过nptr指向字符串中的空白字符，然后把后续字符转换成long int型的值。如果base是2~36之间的数，则把它用作数的基数。如果base为零，除非数是以0（八进制）或者0x/0X（十六进制）开头的，否则就把数设定为十进制的。如果endptr不是空指针，那么strtol函数会修改endptr指向的对象以便endptr可以指向第一个剩余字符。如果没有发现long int型的值，或者它有错误的格式，那么strtol函数会把nptr存储

到endptr指向的对象中。如果没有能表示的数，函数会把ERANGE存储到errno中。

返回　转换的数。如果没有转换可以执行，则返回零。如果无法表示数，则依赖于数的符号返回LONG_MAX或者LONG_MIN。

相似函数　atoi函数、atol函数、strtoul函数

相关函数　strtod函数

也可参见　atof函数　26.2节

---

strtoul　把字符串转换成无符号长整数　<stdlib.h>

```
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```

strtuol函数和strtol函数一样，只不过前者会把字符串转换成为无符号长整数。

返回　转换的数。如果没有转换可以执行，则返回零。如果无法表示数，则返回ULONG_MAX。

相似函数　atoi函数、atol函数、strtol函数

相关函数　strtod函数

也可参见　atof函数　26.2节

---

strxfrm　转换指定地区的字符串　<string.h>

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

函数转换由s2指向的字符串，把结果的前n个字符（包括空字符）放到s1指向的数组中。调用带有两个转换的字符串的strcmp函数应该会产生相同的结果（负数、零或正数），就像调用带有原始字符串的strcol函数。

返回　转换的字符串的长度（可能超过n）。

631　相似函数　strcmp函数、strcoll函数　23.5节

System　执行操作系统命令　<stdlib.h>

```
int system(const char *string);
```

把string指向的字符串传递给操作系统的命令处理器（命令解释程序）来执行。

返回　当string是空指针时，如果命令处理器有效，则返回非零值。如果string不是空指针，则返回由实现定义的值。

也可参见　getenv函数　26.2节

---

tan　正切　<math.h>

```
double tan(double x);
```

返回　x的正切值（按照弧度衡量的）。

相关函数　acos函数、asin函数、atan函数、atan2函数、cos函数、sin函数　23.3节

---

tanh　双曲正切　<math.h>

```
double tanh(double x);
```

返回　x的双曲正切值。

相关函数　cosh函数、sinh函数

也可参见　acos函数、asin函数、atan函数、atan2函数、cos函数、sin函数、tan函数　23.3节

---

time　当前时间　<time.h>

```
time_t time(time_t *timer);
```

返回　当前的日历时间。如果日历时间无效，则返回(time_t)-1。如果timer不是空指针，也把返回值存储到timer指向的对象中。

相似函数　clock函数

相关函数　asctime函数、ctime函数、difftime函数、gmtime函数、localtime函数、mktime函数、strftime函数　26.3节

---

tmpfile　创建临时文件　<stdio.h>

```
FILE *tmpfile(void);
```

创建临时文件，此文件在被关闭或者程序结束时会被自动删除。按照"wb+"模式打开文件。

返回　文件指针。当执行对此文件的后续操作时候用到此指针。如果无法创建文件，则返回空

指针。

相关函数　tmpnam函数、fopen函数　22.2节

---

tmpnam　产生临时文件名　<stdio.h>

```
char *tmpnam(char *s);
```

产生临时文件名。如果s是空指针，那么tmpnam把文件名存储在静态变量中。否则，它会把文件名复制到s指向的字符数组中。（数组必须足够长可以存储L_tmpnam个字符，这里的L_tmpnam是在<stdio.h>头文件中定义的宏。）

返回　指向文件名的指针。

相关函数　tmpfile函数　22.2节　632

---

tolower　转换成小写字母　<ctype.h>

```
int tolower(int c);
```

返回　如果c是大写字母，则返回相应的小写字母。如果c不是大写字母，则返回无变化的c。

相似函数　toupper函数

相关函数　islower函数、isupper函数

也可参见　isalpha函数　23.4节

---

toupper　转换成大写字母　<ctype.h>

```
int toupper(int c);
```

返回　如果c是小写字母，则返回相应的大写字母。如果c不是小写字母，则返回无变化的c。

相似函数　tolower函数

相关函数　islower函数、isupper函数

也可参见　isalpha函数　23.4节

---

ungetc　未读取的字符　<stdio.h>

```
int ungetc(int c, FILE *stream);
```

把字符c回退到stream指向的流中，并且清除流的文件尾指示器。由连续的ungetc函数调用回退的字符数量有变化。只能保证第一次调用成功。调用文件定位函数（fseek函数、fsetpos函数或者rewind函数）会导致回退的字符丢失。

返回　c（回退的字符）。如果没有读取操作或者文件定位操作就试图回退过多的字符，那么函数将会返回EOF。

相关函数　fgetc函数、getc函数、getchar函数　22.4节　633

---

va_arg　从可变实际参数列表中获取参数　<stdarg.h>

```
类型 va_arg(va_list ap, 类型);
```

从变量参数列表中获取一个参数，然后修改ap使va_arg下一次的使用可以获取后面的参数。在va_arg第一次使用之前必须由va_start对ap进行初始化。

返回　假设参数的类型（在采用了默认的实际参数提升之后）与类型一致，返回参数的值。

相关函数　va_end函数、va_start函数

也可参见　vfprintf函数、vprintf函数、vsprintf函数　26.1节

---

va_end　结束可变实际参数列表的处理　<stdarg.h>

```
void va_end(va_list ap);
```

结束与ap相关的可变实际参数列表的处理。

相关函数　va_arg函数、va_start函数

也可参见　vfprintf函数、vprintf函数、vsprintf函数　26.1节

---

va_start　开始可变实际参数列表的处理　<stdarg.h>

```
void va_start(va_list ap, parmN);
```

必须在访问参数列表之前调用它。初始化ap以便稍后va_arg和va_end的使用。*parmN*是最后一个普通参数的名字（此参数后边跟着，...）。

相关函数　va_arg函数、va_end函数

也可参见　vfprintf函数、vprintf函数、vsprintf函数　26.1节

vfprintf　用可变实际参数列表格式化写文件　<stdio.h>

```
int vfprintf(FILE *stream, const char *format,
             va_list arg);
```

函数等价于用arg替换带有可变实际参数列表的fprintf函数。

返回　写入的字符数量。如果发生错误就返回负值。

相似函数　fprintf函数、printf函数、sprintf函数、vprintf函数、vsprintf函数

也可参见　va_arg函数、va_end函数、va_start函数　26.1节

vprintf　用可变实际参数列表格式化写　<stdio.h>

```
int vprintf(const char *format, va_list arg);
```

函数等价于用arg替换带有可变实际参数列表的printf函数。

返回　写入的字符数量。如果发生错误就返回负值。

相似函数　fprintf函数、printf函数、sprintf函数、vfprintf函数、vsprintf函数

634 也可参见　va_arg函数、va_end函数、va_start函数　26.1节

vsprintf　用可变实际参数列表格式化写字符串　<stdio.h>

```
int vsprintf(char *s, const char *format,
             va_list arg);
```

函数等价于用arg替换带有可变实际参数列表的sprintf函数。

返回　存储的字符数量，但不计空字符。

相似函数　fprintf函数、printf函数、sprintf函数、vfprintf函数、vprintf函数

也可参见　va_arg函数、va_end函数、va_start函数　26.1节

wcstombs　把宽字符串转换成多字节字符串　<stdlib.h>

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

把宽字符码序列转换成为对应的多字节字符。pwcs指向含有宽字符的数组。多字节字符存储在s指向的数组中。如果遇到存储的空字符或者要存储的多字节字符将超过n个字节的限制，则转换结束。

返回　存储的字节数，不包括空字符。如果遇到一个代码不对应有效多字节字符时，则返回(size_t) -1。

相关函数　mbstowcs函数

也可参见　mblen函数、mbtowc函数、setlocale函数、wctomb函数　25.2节

wctomb　把宽字符转换成多字节字符　<stdlib.h>

```
int wctomb(char *s, wchar_t wchar);
```

把代码为wchar的宽字符转换成为一个多字节字符。如果s不是空指针，则把结果存储到s指向的数组中。如果s是空指针，则初始化移位状态。

返回　如果s是空指针，则返回非零值或零值，这依赖于多字节字符是否是依赖状态编码的。如果wchar对应一个有效的多字节字符，则返回字符中字节的数量，如果不是这样，则返回-1。

相关函数　mblen函数、mbtowc函数

635 也可参见　mbstowcs函数、setlocale函数、wcstomb函数　25.2节

<match.h>函数的错误

定义域错误　参数超出了函数的定义域。如果出现定义域错误，函数的返回值是由实现定义的，并且函数会把EDOM存储到errno中。

取值范围错误　函数的返回值超出了double型值的取值范围。如果返回值的数太大以致于无法表示（上溢），则函数返回正的或负的HUGE_VAL，这要依赖于正确结果的符号。此外，函数会把ERANGE存储到errno中。如果返回值的数太小以致于无法表示（下溢），则函数返回零。

636 一些实现也可能会把ERANGE存储到errno中。

写 Makefile

概述

什么是 makefile? 或许很多 Winodws的程序员都不知道这个东西,因为那些 Windows的 IDE都为你做了这个工作,但我觉得要作一个好的和 professional的程序员,makefile还是要懂。这就好像现在有这么多的 HTML的编辑器,但如果你想成为一个专业人士,你还是要了解 HTML 的标识的含义。特别在 Unix 下的软件编译,你就不能不自己写 makefile了,会不会写 makefile,从一个侧面说明了一个人是否具备完成大型工程的能力。

因为,makefile关系到了整个工程的编译规则。一个工程中的源文件不计数,其按类型、功能、模块分别放在若干个目录中,makefile定义了一系列的规则来指定,哪些文件需要先编译,哪些文件需要后编译,哪些文件需要重新编译,甚至于进行更复杂的功能操作,因为 makefile就像一个 Shell脚本一样,其中也可以执行操作系统的命令。

makefile带来的好处就是——"自动化编译",一旦写好,只需要一个 make命令,整个工程完全自动编译,极大的提高了软件开发的效率。make是一个命令工具,是一个解释 makefile中指令的命令工具,一般来说,大多数的 IDE都有这个命令,比如: Delphi的 make, Visual C++的 nmake, Linux下 GNU的 make。可见,makefile都成为了一种在工程方面的编译方法。

现在讲述如何写 makefile的文章比较少,这是我想写这篇文章的原因。当然,不同产商的 make各不相同,也有不同的语法,但其本质都是在"文件依赖性"上做文章,这里,我仅对 GNU的 make进行讲述,我的环境是 RedHat Linux 8.0,make的版本是 3.80。必竟,这个 make是应用最为广泛的,也是用得最多的。而且其还是最遵循于 IEEE 1003.2-1992 标准的 (POSIX.2)。

在这篇文档中,将以 C/C++的源码作为我们基础,所以必然涉及一些关于 C/C++的编译的知识,相关于这方面的内容,还请各位查看相关的编译器的文档。这里所默认的编译器是 UNIX下的 GCC和 CC。

关于程序的编译和链接

在此,我想多说关于程序编译的一些规范和方法,一般来说,无论是 C、C++、还是 pas,首先要把源文件编译成中间代码文件,在 Windows下也就是 .obj 文件,UNIX下是 .o 文件,即 Object File,这个动作叫做编译 (compile)。然后再把大量的 Object File合成执行文件,这个动作叫作链接 (link)。

编译时,编译器需要的是语法的正确,函数与变量的声明的正确。对于后者,通常是你需要告诉编译器头文件的所在位置 (头文件中应该只是声明,而定义应该放在 C/C++文件中),只要所有的语法正确,编译器就可以编译出中间目标文件。一般来说,每个源文件都应该对应于一个中间目标文件 (O文件或是 OBJ文件)。

链接时,主要是链接函数和全局变量,所以,我们可以使用这些中间目标文件 (O文件或是 OBJ文件) 来链接我们的应用程序。链接器并不管函数所在的源文件,只管函数的中间目标文件 (Object File),在大多数时候,由于源文件太多,编译生成的中间目标文件太多,而在链接时需要明显地指出中间目标文件名,这对于编译很不方便,所以,我们要给中间目标文件打个包,在 Windows下这种包叫"库文件" (Library File),也就是 .lib 文件,在 UNIX下,是 Archive File,也就是 .a 文件。

总结一下,源文件首先会生成中间目标文件,再由中间目标文件生成执行文件。在编译时,编译器只检测程序语法,和函数、变量是否被声明。如果函数未被声明,编译器会给出一个警告,但可以生成 Object File。而在链接程序时,链接器会在所有的 Object File中找寻函数的实现,如果找不到,那到就会报链接错误码 (Linker Error),在 VC下,这种错误一般是:Link 2001错误,意思说是说,链接器未能找到函数的实现。你需要指定函数的 Object File.

好,言归正传,GNU的 make有许多的内容,闲言少叙,还是让我们开始吧。

Makefile 概要介绍

make命令执行时，需要一个 Makefile 文件，以告诉 make命令需要怎么样的去编译和链接程序。

首先，我们用一个示例来说明 Makefile的书写规则。以便给大家一个感兴认识。这个示例来源于 GNU的 make使用手册，在这个示例中，我们的工程有 8个 C文件，和 3个头文件，我们要写一个 Makefile来告诉 make命令如何编译和链接这几个文件。我们的规则是：

1) 如果这个工程没有编译过，那么我们的所有 C文件都要编译并被链接。

2) 如果这个工程的某几个 C文件被修改，那么我们只编译被修改的 C文件，并链接目标程序。

3) 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C文件，并链接目标程序。

只要我们的 Makefile写得够好，所有的这一切，我们只用一个 make命令就可以完成，make命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

一、Makefile 的规则

在讲述这个 Makefile之前，还是让我们先来粗略地看一看 Makefile的规则。

```
target ... : prerequisites ...
        command
        ...
        ...
```

target也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites就是，要生成那个 target所需要的文件或是目标。

command也就是 make需要执行的命令。（任意的 Shell命令）

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于 prerequisites中的文件，其生成规则定义在 command中。说白一点就是说，prerequisites中如果有一个以上的文件比 target文件要新的话，command所定义的命令就会被执行。这就是 Makefile的规则。也就是 Makefile中最核心的内容。

说到底，Makefile的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是 Makefile的主线和核心，但要写好一个 Makefile还不够，我会以后面一点一点地结合我的工作经验给你慢慢到来。内容还多着呢。:)

二、一个示例

正如前面所说的，如果一个工程有 3个头文件，和 8个 C文件，我们为了完成前面所述的那三个规则，我们的 Makefile应该是下面的这个样子的。

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
```

```
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

反斜杠（\）是换行符的意思。这样比较便于 Makefile的易读。我们可以把这个内容保存在文件为“ Makefile”或“ makefile”的文件中，然后在该目录下直接输入命令“ make”就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下“ make clean”就可以了。

在这个 makefile中，目标文件（target）包含：执行文件 edit和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab键作为开头。记住，make并不管命令是怎么工作的，他只管执行所定义的命令。make会比较 targets文件和 prerequisites文件的修改日期，如果 prerequisites文件的日期要比 targets文件的日期要新，或者 target不存在的话，那么，make就会执行后续定义的命令。

这里要说明一点的是， clean不是一个文件，它只不过是一个动作名字，有点像 C语言中的 lable一样，其冒号后什么也没有，那么，make就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make命令后明显得指出这个 lable的名字。这样的方法非常有用，我们可以在一个 makefile中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

三、make 是如何工作的

在默认的方式下，也就是我们只输入 make命令。那么，

1. make会在当前目录下找名字叫“ Makefile”或“ makefile”的文件。

2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“ edit”这个文件，并把这个文件作为最终的目标文件。

3. 如果 edit文件不存在，或是 edit所依赖的后面的 .o 文件的文件修改时间要比 edit这个文件新，那么，他就会执行后面所定义的命令来生成 edit这个文件。

4. 如果 edit所依赖的 .o文件也存在，那么 make会在当前文件中找目标为 .o文件的依赖性，如果找到则再根据那一个规则生成 .o文件。（这有点像一个堆栈的过程）

5. 当然，你的 C文件和 H文件是存在的啦，于是 make会生成 .o 文件，然后再用 .o 文件生命 make的终极任务，也就是执行文件 edit了。

这就是整个 make的依赖性，make会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make根本不理。make只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make执行。即命令——“ make clean”，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，我们的目标 file.o会被重编译（也就是在这个依性关系后面所定义的命令），于是 file.o的文件也是最新的啦，于是 file.o的文件修改时间要比 edit要新，所以 edit也会被重新链接了（详见 edit目标文件后定义的命令）。

而如果我们改变了“ command.h”，那么， kdb.o、 command.o和 files.o都会被重编译，并且， edit会被重链接。

四、makefile 中使用变量

在上面的例子中，先让我们看看 edit的规则：

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在 clean中）。当然，我们的 makefile并不复杂，所以在两个地方加也不累，但如果 makefile变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了 makefile的易维护，在 makefile中我们可以使用变量。makefile的变量也就是一个字符串，理解成 C语言中的宏可能会更好。

比如，我们声明一个变量，叫 objects, OBJECTS, objs, OBJS, obj, 或是 OBJ, 反正不管什么啦，只要能够表示 obj文件就行了。我们在 makefile一开始就这样定义：

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

于是，我们就可以很方便地在我们的 makefile中以“ $(objects)”的方式来使用这个变量了，于是我们的改良版 makefile就变成下面这个样子：

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
edit : $(objects)
    cc -o edit $(objects)
main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
command.o : command.c defs.h command.h
    cc -c command.c
display.o : display.c defs.h buffer.h
    cc -c display.c
insert.o : insert.c defs.h buffer.h
    cc -c insert.c
search.o : search.c defs.h buffer.h
    cc -c search.c
files.o : files.c defs.h buffer.h command.h
    cc -c files.c
utils.o : utils.c defs.h
    cc -c utils.c
clean :
    rm edit $(objects)
```

于是如果有新的 .o文件加入，我们只需简单地修改一下 objects 变量就可以了。

关于变量更多的话题，我会在后续给你一一道来。

五、让 make 自动推导

GNU的 make很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个 [.o]文件后都写上类似的命令，因为，我们的 make会自动识别，并自己推导命令。

只要 make看到一个 [.o]文件，它就会自动的把 [.c]文件加在依赖关系中，如果 make找到一个 whatever.o，那么 whatever.c，就会是 whatever.o的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，我们的 makefile再也不用写得这么复杂。我们的是新的 makefile又出炉了。

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
        cc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
.PHONY : clean
clean :
        rm edit $(objects)
```

这种方法，也就是 make的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean是个伪目标文件。

关于更为详细的“隐晦规则”和“伪目标文件”，我会在后续给你一一道来。

六、另类风格的 makefile

即然我们的 make可以自动推导命令，那么我看到那堆 [.o]和 [.h]的依赖就有点不爽，那么多的重复的 [.h]，能不能把其收拢起来，好吧，没有问题，这个对于 make来说很容易，谁叫它提供了自动推导命令和文件的功能呢？来看看最新风格的 makefile吧。

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
        cc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
.PHONY : clean
clean :
        rm edit $(objects)
```

这种风格，让我们的 makefile变得很简单，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的 .o文件，那就理不清楚了。

七、清空目标文件的规则

每个 Makefile中都应该写一个清空目标文件（ .o和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“ 修养 ”(呵呵，还记得我的《编程修养》吗)。一般的风格都是：

```
clean:
    rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

前面说过， .PHONY意思表示 clean是一个“ 伪目标 ”,。而在 rm命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然， clean的规则不要放在文件的开头，不然，这就会变成 make的默认目标，相信谁也不愿意这样。不成文的规矩是——“ clean从来都是放在文件的最后 ”。

Makefile 详细介绍

一、Makefile 里有什么？

Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1. 显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由 Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

2. 隐晦规则。由于我们的 make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make所支持的。

3. 变量的定义。在 Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C语言中的宏，当 Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。

4. 文件指示。其包括了三个部分，一个是在一个 Makefile中引用另一个 Makefile，就像 C语言中的 include一样；另一个是指根据某些情况指定 Makefile中的有效部分，就像 C语言中的预编译 #if一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。

5. 注释。Makefile中只有行注释，和 UNIX的 Shell脚本一样，其注释是用“ #”字符，这个就像 C/C++中的“ //”一样。如果你要在你的 Makefile中使用“ #”字符，可以用反斜框进行转义，如：“ \#”。

最后，还值得一提的是，在 Makefile中的命令，必须要以 [Tab]键开始。

二、Makefile 的文件名

默认的情况下，make命令会在当前目录下按顺序找寻文件名为“ GNUmakefile”、“ makefile”、“ Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“ Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“ GNUmakefile”，这个文件是 GNU的 make识别的。有另外一些 make只对全小写的“ makefile”文件名敏感，但是基本上来说，大多数的 make都支持“ makefile”和“ Makefile”这两种默认文件名。

当然，你可以使用别的文件名来书写 Makefile，比如：“ Make.Linux”，“ Make.Solaris”，“ Make.AIX”等，如果要指定特定的 Makefile，你可以使用 make的“ -f”和“ --file”参数，如：make -f Make.Linux或 make --file Make.AIX。

三、引用其它的 Makefile

在 Makefile使用 include关键字可以把别的 Makefile包含进来，这很像 C语言的 #include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

include <filename>

filename可以是当前操作系统 Shell的文件模式（可以保含路径和通配符）

在 include前面可以有一些空字符，但是绝不能是 [Tab]键开始。include和 <filename>可以用一个或多个空格隔开。举个例子，你有这样几个 Makefile：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量 $(bar)，其包含了 e.mk和 f.mk，那么，下面的语句：

include foo.make *.mk $(bar)

等价于：

include foo.make a.mk b.mk c.mk e.mk f.mk

make命令开始时，会把找寻 include所指出的其它 Makefile，并把其内容安置在当前的位置。就好像 C/C++的 #include指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果当前目录下没有找到，那么，make还会在下面的几个目录下找：

1. 如果 make执行时，有“ -I”或“ --include-dir”参数，那么 make就会在这个参数所指定的目录下去寻找。
2. 如果目录 <prefix>/include（一般是： /usr/local/bin或 /usr/include）存在的话，make也会去找。

如果有文件没有找到的话，make会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成 makefile的读取，make会再重试这些没有找到，或是不能读取的文件，如果还是不行，make才会出现一条致命信息。如果你想让 make不理那些无法读取的文件，而继续执行，你可以在 include前加一个减号“ -”。如：

-include <filename>

其表示，无论 include过程中出现什么错误，都不要报错继续执行。和其它版本 make兼容的相关命令是 sinclude，其作用和这一个是一样的。

四、环境变量 MAKEFILES

如果你的当前环境中定义了环境变量 MAKEFILES，那么，make会把这个变量中的值做一个类似于 include的动作。这个变量中的值是其它的 Makefile，用空格分隔。只是，它和 include不同的是，从这个环境变中引入的 Makefile的“目标”不会起作用，如果环境变量中定义的文件发现错误，make也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用 make时，所有的 Makefile都会受到它的影响，这绝不是你想看到的。在这里提这个事，只是为了告诉大家，也许有时候你的 Makefile出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

五、make 的工作方式

GNU的 make工作时的执行步骤入下：（想来其它的 make也是类似）

1. 读入所有的 Makefile。
2. 读入被 include的其它 Makefile。
3. 初始化文件中的变量。
4. 推导隐晦规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make会把其展开在

使用的位置。但 make并不会完全马上展开，make使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

当然，这个工作方式你不一定要清楚，但是知道这个方式你也会对 make更为熟悉。有了这个基础，后续部分也就容易看懂了。

书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在 Makefile中，规则的顺序是很重要的，因为， Makefile中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，所以一定要让 make知道你的最终目标是什么。一般来说，定义在 Makefile中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

一、规则举例

```
foo.o : foo.c defs.h       # foo模块
        cc -c -g foo.c
```

看到这个例子，各位应该不是很陌生了，前面也已说过， foo.o是我们的目标， foo.c和 defs.h是目标所依赖的源文件，而只有一个命令“ cc -c -g foo.c”（以 Tab键开头）。这个规则告诉我们两件事：

1. 文件的依赖关系， foo.o依赖于 foo.c和 defs.h的文件，如果 foo.c和 defs.h的文件日期要比 foo.o文件日期要新，或是 foo.o不存在，那么依赖关系发生。

2. 如果生成（或更新） foo.o文件。也就是那个 cc命令，其说明了，如何生成 foo.o这个文件。（当然 foo.c文件 include了 defs.h文件）

二、规则的语法

```
targets : prerequisites
  command
  ...
或是这样：
targets : prerequisites ; command
      command
      ...
```

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command 是命令行，如果其不与“ target:prerequisites”在一行，那么，必须以 [Tab 键]开头，如果和 prerequisites在一行，那么可以用分号做为分隔。（见上）

prerequisites也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘ \’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make两件事，文件的依赖关系和如何成成目标文件。

一般来说，make会以 UNIX的标准 Shell，也就是 /bin/sh来执行命令。

三、在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make支持三各通配符：“ *”，“ ?”和“ [...]”。这是和 Unix的 B-Shell是相同的。

波浪号（“ ~”）字符在文件名中也有比较特殊的用途。如果是“ ~/test”，这就表示当前用户的 $HOME目录下的 test目录。而“ ~hchen/test”则表示用户 hchen的宿主目录下的 test目录。（这些都是 Unix下的小知识了，make也支持）而在 Windows或是 MS-DOS下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“ HOME”而定。

通配符代替了你一系列的文件，如“ *.c”表示所以后缀为 c的文件。一个需要我们注意的是，如果我们的文件名中有通配符，如：“ *”，那么可以用转义字符“ \”，如“ *”来表示真实的“ *”字符，而不是任意长度的字符串。

好吧，还是先来看几个例子吧：

```
clean:
	rm -f *.o
```

上面这个例子我不不多说了，这是操作系统 Shell所支持的通配符。这是在命令中的通配符。

```
print: *.c
	lpr -p $?
	touch print
```

上面这个例子说明了通配符也可以在我们的规则中，目标 print依赖于所有的 [.c]文件。其中的“ $?”是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了，通符同样可以用在变量中。并不是说 [*.o]会展开，不！ objects的值就是“ *.o”。Makefile中的变量其实就是 C/C++中的宏。如果你要让通配符在变量中展开，也就是让 objects的值是所有 [.o]的文件名的集合，那么，你可以这样：

```
objects := $(wildcard *.o)
```

这种用法由关键字“ wildcard”指出，关于 Makefile的关键字，我们将在后面讨论。

四、文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当 make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉 make，让 make在自动去找。

Makefile文件中的特殊变量“ VPATH”就是完成这个功能的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

上面的的定义指定两个目录，“ src”和“ ../headers”，make会按照这个顺序进行搜索。目录由“ 冒号 ”分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用 make的“ vpath”关键字（注意，它是全小写的），这不是变量，这是一个 make的关键字，这和上面提到的那个 VPATH变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、 vpath <pattern> <directories>

为符合模式 <pattern>的文件指定搜索目录 <directories>。

2、 vpath <pattern>

清除符合模式 <pattern>的文件的搜索目录。

3、 vpath

清除所有已被设置好了的文件搜索目录。

vapth使用方法中的 <pattern>需要包含“ %”字符。“ %”的意思是匹配零或若干字符，例如，“ %.h”表示所有以“ .h”结尾的文件。 <pattern>指定了要搜索的文件集，而 <directories>则指定了 <pattern>的文件集的搜索的目录。

例如：

```
vpath %.h ../headers
```

该语句表示，要求 make在“ ../headers”目录下搜索所有以“ .h”结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用 vpath语句，以指定不同搜索策略。如果连续的 vpath语句中出现了相同的 <pattern>，或是被重复了的 <pattern>，那么，make会按照 vpath语句的先后顺序来执行搜索。如：

```
vpath %.c foo
vpath %  blish
vpath %.c bar
```

其表示“ .c”结尾的文件，先在“ foo”目录，然后是“ blish”，最后是“ bar”目录。

```
vpath %.c foo:bar
vpath %  blish
```

而上面的语句则表示“ .c”结尾的文件，先在“ foo”目录，然后是“ bar”目录，最后才是“ blish”目录。

五、伪目标

最早先的一个例子中，我们提到过一个“ clean”的目标，这是一个“ 伪目标 ”，

```
clean:
        rm *.o temp
```

正像我们前面例子中的“ clean”一样，即然我们生成了许多文件编译文件，我们也应该提供一个清除它们的“ 目标 ”以备完整地重编译而用。 （以“ make clean”来使用该目标）

因为，我们并不生成“ clean”这个文件。“ 伪目标 ”并不是一个文件，只是一个标签，由于“ 伪目标 ”不是文件，所以 make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“ 目标 ”才能让其生效。当然，“ 伪目标 ”的取名不能和文件名重名，不然其就失去了“ 伪目标 ”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“ .PHONY”来显示地指明一个目标是“ 伪目标 ”，向 make说明，不管是否有这个文件，这个目标就是“ 伪目标 ”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“ clean”文件，要运行“ clean”这个目标，只有“ make clean”这样。于是整个过程可以这样写：

```
 .PHONY: clean
clean:
        rm *.o temp
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“ 默认目标 ”，只要将其放在第一个。一个示例就是，如果你的 Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个 make完事，并且，所有的目标文件都写在一个 Makefile中，那么你可以使用“ 伪目标 ”这个特性：

```
all : prog1 prog2 prog3
.PHONY : all
prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o
prog2 : prog2.o
        cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile中的第一个目标会被作为其默认目标。我们声明了一个“ all”的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“ all”这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。“ .PHONY : all”声明了“ all”这个目标为“ 伪目标 ”。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例

子：

```
.PHONY: cleanall cleanobj cleandiff
cleanall : cleanobj cleandiff
        rm program
cleanobj :
        rm *.o
cleandiff :
        rm *.diff
```

“ make clean”将清除所有要被清除的文件。“ cleanobj”和“ cleandiff”这两个伪目标有点像“ 子程序 ”的意思。我们可以输入“ make cleanall”和“ make cleanobj”和“ make cleandiff”命令来达到清除不同种类文件的目的。

六、多目标

Makefile的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量“ $@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

上述规则等价于：

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

其中， -$(subst output,,$@)中的“ $”表示执行一个 Makefile的函数，函数名为 subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“ $@”表示目标的集合，就像一个数组，“ $@”依次取出目标，并执于命令。

七、静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

```
<targets ...>: <target-pattern>: <prereq-patterns ...>
        <commands>
        ...
```

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern是指明了 targets的模式，也就是的目标集模式。

prereq-parrterns是目标的依赖模式，它对 target-parrtern形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的 <target-parrtern>定义成“ %.o”，意思是我们的 <target>集合中都是以“ .o”结尾的，而如果我们的 <prereq-parrterns>定义成“ %.c”，意思是对 <target-parrtern>所形成的目标集进行二次定义，其计算方法是，取 <target-parrtern>模式中的“ %”（也就是去掉了 [.o]这个结尾），并为其加上 [.c]这个结尾，形成的新集合。

所以，我们的“ 目标模式 ”或是“ 依赖模式 ”中都应该有“ %”这个字符，如果你的文件名中有“ %”那么你可以使用反斜杠“ \”进行转义，来标明真实的“ %”字符。

看一个例子：

```
objects = foo.o bar.o
all: $(objects)
$(objects): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从 $object中获取，“ %.o”表明要所有以“ .o”结尾的目标，也就是“ foo.o bar.o”，也就是变量 $object集合的模式，而依赖模式“ %.c”则取模式“ %.o”的“ %”，也就是“ foo bar”，并为其加下“ .c”的后缀，于是，我们的依赖目标就是“ foo.c bar.c”。而命令中的“ $<”和“ $@”则是自动化变量，“ $<”表示所有的依赖目标集（也就是“ foo.c bar.c”），“ $@”表示目标集（也就是“ foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

```
foo.o : foo.c
	$(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
	$(CC) -c $(CFLAGS) bar.c -o bar.o
```

试想，如果我们的“ %.o”有几百个，那种我们只要用这种很简单的“ 静态模式规则 ”就可以写完一堆规则，实在是太有效率了。“ 静态模式规则 ”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
files = foo.elc bar.o lose.o
$(filter %.o,$(files)): %.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
	emacs -f batch-byte-compile $<
```

$(filter%.o,$(files)表示调用 Makefile的 filter函数，过滤“ $filter”集，只要其中模式为“ %.o”的内容。其的它内容，我就不用多说了吧。这个例字展示了 Makefile中更大的弹性。

八、自动生成依赖性

在 Makefile中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的 main.c中有一句“ #include "defs.h"”，那么我们的依赖关系应该是：

```
main.o : main.c defs.h
```

但是，如果是一个比较大型的工程，你必需清楚哪些 C文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改 Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用 C/C++编译的一个功能。大多数的 C/C++编译器都支持一个“ -M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

```
cc -M main.c
```

其输出是：

```
main.o : main.c defs.h
```

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用 GNU的 C/C++编译器，你得用“ -MM”参数，不然，“ -M”参数会把一些标准库的头文件也包含进来。

gcc -M main.c的输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
/usr/include/bits/sched.h /usr/include/libio.h \
```

```
/usr/include/_G_config.h /usr/include/wchar.h \
/usr/include/bits/wchar.h /usr/include/gconv.h \
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
/usr/include/bits/stdio_lim.h
```

gcc -MM main.c的输出则是：

```
main.o: main.c defs.h
```

那么，编译器的这个功能如何与我们的 Makefile联系在一起呢。因为这样一来，我们的 Makefile也要根据这些源文件重新生成，让 Makefile自已依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“ name.c”的文件都生成一个“ name.d”的 Makefile文件，[.d]文件中就存放对应 [.c]文件的依赖关系。

于是，我们可以写出 [.c]文件和 [.d]文件的依赖关系，并让 make自动更新或自成 [.d]文件，并把其包含在我们的主 Makefile中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生 [.d]文件：

```
%.d: %.c
        @set -e; rm -f $@; \
         $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
         sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
         rm -f $@.$$$$
```

这个规则的意思是，所有的 [.d]文件依赖于 [.c]文件，“ rm -f $@”的意思是删除所有的目标，也就是 [.d]文件，第二行的意思是，为每个依赖文件“ $<”，也就是 [.c]文件生成依赖文件，“ $@”表示模式“ %.d”文件，如果有一个 C文件是 name.c，那么“ %”就是“ name”，“ $$$$”意为一个随机编号，第二行生成的文件有可能是“ name.d.12345”，第三行使用 sed命令做了一个替换，关于 sed命令的用法请参看相关的使用文档。第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入 [.d]文件的依赖，即把依赖关系：

```
main.o : main.c defs.h
```

转成：

```
main.o main.d : main.c defs.h
```

于是，我们的 [.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个 [.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个 [.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主 Makefile中。我们可以使用 Makefile的“ include”命令，来引入别的 Makefile文件（前面讲过），例如：

```
sources = foo.c bar.c
include $(sources:.c=.d)
```

上述语句中的“ $(sources:.c=.d)”中的“ .c=.d”的意思是做一个替换，把变量 $(sources)所有 [.c]的字串都替换成 [.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为 include是按次来载入文件，最先载入的 [.d]文件中的目标会成为默认目标。

书写命令

每条规则中的命令和操作系统 Shell的命令行是一致的。make会一按顺序一条一条的执行命令，每条命令的开头必须以 [Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以 Tab键开头的，那么 make会认为其是一个空命令。

我们在 UNIX下可能会使用不同的 Shell，但是 make的命令默认是被“ /bin/sh”——UNIX的标准 Shell解释执行的。除非你特别指定一个其它的 Shell。Makefile中，“ #”是注释符，很像 C/C++中的“ //”，其后的本行字符都被注释。

一、显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

```
@echo 正在编译 XXX模块 ......
```

当make执行时，会输出“正在编译 XXX模块 ......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

```
echo 正在编译 XXX模块 ......
正在编译 XXX模块 ......
```

如果make执行时，带入make参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而make参数“-s”或“--slient”则是全面禁止命令的显示。

二、命令执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是cd命令，你希望第二条命令得在cd之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

```
exec:
        cd /home/hchen
        pwd
```

示例二：

```
exec:
        cd /home/hchen; pwd
```

当我们执行“make exec”时，第一个例子中的cd没有作用，pwd会打印出当前的Makefile目录，而第二个例子中，cd就起作用了，pwd会打印出“/home/hchen”。

make一般是使用环境变量SHELL中所定义的系统Shell来执行命令，默认情况下使用UNIX的标准Shell——/bin/sh来执行命令。但在MS-DOS下有点特殊，因为MS-DOS下没有SHELL环境变量，当然你也可以指定。如果你指定了UNIX风格的目录形式，首先，make会在SHELL所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在PATH环境变量中所定义的所有路径中寻找。MS-DOS中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

三、命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如mkdir命令，我们一定需要建立一个目录，如果目录不存在，那么mkdir就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用mkdir的意思就是一定要有这样的一个目录，于是我们就不希望mkdir出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为不管命令出不出错都认为是成功的。如：

```
clean:
```

```
-rm -f *.o
```

还有一个全局的办法是，给 make加上“ -i”或是“ --ignore-errors”参数，那么，Makefile中所有命令都会忽略错误。而如果一个规则是以“ .IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的 make的参数的是“ -k”或是“ --keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终目该规则的执行，但继续执行其它规则。

四、嵌套执行 make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的 Makefile，这有利于让我们的 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的 Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的 Makefile可以这样书写：

```
subsystem:
        cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
        $(MAKE) -C subdir
```

定义 $(MAKE)宏变量的意思是，也许我们的 make需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“ subdir”目录，然后执行 make命令。

我们把这个 Makefile叫做“总控 Makefile”，总控 Makefile的变量可以传递到下级的 Makefile中（如果你显示的声明），但是不会覆盖下层的 Makefile中所定义的变量，除非指定了“ -e”参数。

如果你要传递变量到下级 Makefile中，那么你可以使用这样的声明：

```
export <variable ...>
```

如果你不想让某些变量传递到下级 Makefile中，那么你可以这样声明：

```
unexport <variable ...>
```

如：

示例一：

```
export variable = value
```

其等价于：

```
variable = value
export variable
```

其等价于：

```
export variable := value
```

其等价于：

```
variable := value
export variable
```

示例二：

```
export variable += value
```

其等价于：

```
variable += value
export variable
```

如果你要传递所有的变量，那么，只要一个 export就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到下层 Makefile中，特别是 MAKEFILES变量，其中包含了 make的参数信息，如果我们执行“总控 Makefile”时有make参数或是在上层 Makefile中定义了这个变量，那么 MAKEFILES变量将会是这些参数，并会传递到下层 Makefile

中，这是一个系统级的环境变量。

但是 make命令中的有几个参数并不往下传递，它们是“ -C”,“ -f”,“ -h”“ -o”和“ -W”(有关 Makefile参数的细节将在后面说明)，如果你不想往下层传递参数，那么，你可以这样来：

```
subsystem:
        cd subdir && $(MAKE) MAKEFLAGS=
```

如果你定义了环境变量 MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有“ -t”,“ -n”和“ -q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在“嵌套执行”中比较有用的参数，“ -w”或是“ --print-directory”会在 make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级 make目录是“ /home/hchen/gnu/make”，如果我们使用“ make -w”来执行，那么当进入该目录时，我们会看到：

```
make: Entering directory `/home/hchen/gnu/make'.
```

而在完成下层 make后离开目录时，我们会看到：

```
make: Leaving directory `/home/hchen/gnu/make'
```

当你使用“ -C”参数来指定 make下层 Makefile时，“ -w”会被自动打开的。如果参数中有“ -s”(“ --slient”)或是“ --no-print-directory”，那么，“ -w”总是失效的。

五、定义命令包

如果 Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“ define”开始，以“ endef”结束，如：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

这里，“ run-yacc”是这个命令包的名字，其不要和 Makefile中的变量重名。在“ define”和“ endef”中的两行就是命令序列。这个命令包中的第一个命令是运行 Yacc程序，因为 Yacc程序总是生成“ y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

```
foo.c : foo.y
        $(run-yacc)
```

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“ run-yacc”中的“ $^”就是“ foo.y”,“ $@”就是“ foo.c”(有关这种以“ $”开头的特殊变量，我们会在后面介绍)，make在执行命令包时，命令包中的每个命令会被依次独立执行。

使用变量

在 Makefile中的定义的变量，就像是 C/C++语言中的宏一样，他代表了一个文本字串，在 Makefile中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++所不同的是，你可以在 Makefile中改变其值。在 Makefile中，变量可以使用在“目标”,“依赖目标”,“命令”或是 Makefile的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头)，但不应该含有“ :”、“ #”、“ =”或是空字符（空格、回车等)。变量是大小写敏感的，“ foo”、“ Foo”和“ FOO”是三个不同的变量名。传统的 Makefile的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如“ $<”、“ $@”等，这些是自动化变量，我会在后面介绍。

一、变量的基础

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“ $”符号，但最好用小括号“ ()”或是大括号“ {}”把变量给包括起来。如果你要使用真实的“ $”字符，那么你需要用“ $$”来表示。

变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)
$(objects) : defs.h
```

变量会在使用它的地方精确地展开，就像 C/C++中的宏一样，例如：

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

展开后得到：

```
prog.o : prog.c
        cc -c prog.c
```

当然，千万不要在你的 Makefile中这样干，这里只是举个例子来表明 Makefile中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。

另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

二、变量中的变量

在定义变量的值时，我们可以使用其它变量来构造变量的值，在 Makefile中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用“ =”号，在“ =”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。如：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
all:
        echo $(foo)
```

我们执行“ make all”将会打出变量 $(foo)的值是“ Huh?”($(foo)的值是 $(bar)， $(bar)的值是 $(ugh)， $(ugh)的值是“ Huh?”) 可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

当“ CFLAGS”在命令中被展开时，会是“ -Ifoo -Ibar -O”。但这种形式也有不好的地方，那就是递归定义，如：

```
CFLAGS = $(CFLAGS) -O
```

或：

```
A = $(B)
B = $(A)
```

这会让 make陷入无限的变量展开过程中去，当然，我们的 make是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的 make运行时非常慢，更糟糕的是，他会使用得两个 make的函数“ wildcard”和“ shell”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

为了避免上面的这种方法，我们可以使用 make中的另一种用变量来定义变量的方法。这种方法使用的是“ :=”操作符，如：

```
x := foo
```

```
y := $(x) bar
x := later
```

其等价于：

```
y := foo bar
x := later
```

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，y的值是“ bar”，而不是“ foo bar”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 make的函数、条件表达式和一个系统变量“ MAKELEVEL”的使用：

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

关于条件表达式和函数，我们在后面再说，对于系统变量“ MAKELEVEL”，其意思是，如果我们的 make有一个嵌套执行的动作（参见前面的“嵌套使用 make”），那么，这个变量会记录了我们的当前 Makefile的调用层数。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：

```
nullstring :=
space := $(nullstring) # end of the line
```

nullstring是一个 Empty变量，其中什么也没有，而我们的 space的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个 Empty变量来标明变量的值开始了，而后面采用“ #”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“ #”的使用，注释符“ #”的这种特性值得我们注意，如果我们这样定义一个变量：

```
dir := /foo/bar    # directory to put the frobs in
```

dir这个变量的值是“ /foo/bar”，后面还跟了 4个空格，如果我们这样使用这样变量来指定别的目录——“ $(dir)/file”那么就完蛋了。

还有一个比较有用的操作符是“ ?=”，先看示例：

```
FOO ?= bar
```

其含义是，如果 FOO没有被定义过，那么变量 FOO的值就是“ bar”，如果 FOO先前被定义过，那么这条语将什么也不做，其等价于：

```
ifeq ($(origin FOO), undefined)
  FOO = bar
endif
```

三、变量高级用法

这里介绍两种变量的高级使用方法，第一种是变量值的替换。

我们可以替换变量中的共有的部分，其格式是“ $(var:a=b)”或是“ ${var:a=b}”，其意思是，把变量“ var”中所有以“ a”字串“结尾”的“ a”替换成“ b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

这个示例中，我们先定义了一个“ $(foo)”变量，而第二行的意思是把“ $(foo)”中所有以“ .o”字串“结尾”全部替换成“ .c”，所以我们的“ $(bar)”的值就是“ a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

第二种高级用法是——“把变量的值再当成变量”。先看一个例子：

```
x = y
y = z
a := $($(x))
```

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

我们还可以使用更多的层次：

```
x = y
y = z
z = u
a := $($($(x)))
```

这里的$(a)的值是“u”，相关的推导留给读者自己去做吧。

让我们再复杂一点，使用上“在变量定义中使用变量”的第一个方式，来看一个例子：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(z)，也就是“Hello”。

再复杂一点，我们再加上函数：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

这个例子中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst函数把“variable1”中的所有“1”字串替换成“2”字串，于是，“variable1”变成“variable2”，再取其值，所以，最终，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不容易）

在这种方式中，或要可以使用多个变量来组成一个变量的名字，然后再取其值：

```
first_second = Hello
a = first
b = second
all = $($a_$b)
```

这里的“$a_$b”组成了“first_second”，于是，$(all)的值就是“Hello”。

再来看看结合第一种技术的例子：

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o
sources := $($(a1)_objects:.o=.c)
```

这个例子中，如果$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这种技术和“函数”与“条件语句”一同使用的例子：

```
ifdef do_sort
func := sort
else
func := strip
endif
```

```
bar := a d b g q c
foo := $($(func) $(bar))
```

这个示例中，如果定义了“ do_sort”，那么： foo := $(sort a d b g q c)，于是$(foo)的值就是“ a b c d g q”，而如果没有定义“ do_sort”，那么： foo := $(sort a d b g q c)，调用的就是 strip函数。

当然，“把变量的值再当成变量”这种技术，同样可以用在操作符的左边：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

这个例子中定义了三个变量：“ dir”，“ foo_sources”和“ foo_print”。

四、追加变量值

我们可以使用“ +=”操作符给变量追加值，如：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

于是，我们的$(objects)值变成：“ main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

使用“ +=”操作符，可以模拟为下面的这种例子：

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

所不同的是，用“ +=”更为简洁。

如果变量之前没有定义过，那么，“ +=”会自动变成“ =”，如果前面有变量定义，那么“ +=”会继承于前次操作的赋值符。如果前一次的是“ :=”，那么“ +=”会以“ :=”作为其赋值符，如：

```
variable := value
variable += more
```

等价于：

```
variable := value
variable := $(variable) more
```

但如果是这种情况：

```
variable = value
variable += more
```

由于前次的赋值符是“ =”，所以“ +=”也会以“ =”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以 make会自动为我们解决这个问题，我们不必担心这个问题。

五、override 指示符

如果有变量是通常 make的命令行参数设置的，那么 Makefile中对这个变量的赋值会被忽略。如果你想在 Makefile中设置这类参数的值，那么，你可以使用“ override”指示符。其语法是：

```
override <variable> = <value>
override <variable> := <value>
```

当然，你还可以追加：

```
override <variable> += <more text>
```

对于多行的变量定义，我们用 define指示符，在 define指示符前，也同样可以使用 ovveride指示符，如：

```
override define foo
bar
```

```
endef
```

六、多行变量

还有一种设置变量值的方法是使用 define关键字。使用 define关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过“ 命令包 ”的技术就是利用这个关键字）。

define指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以 endef关键字结束。其工作方式和“ =”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以 [Tab]键开头，所以如果你用 define定义的命令变量中没有以 [Tab]键开头，那么 make就不会把其认为是命令。

下面的这个示例展示了 define的用法：

```
define two-lines
echo foo
echo $(bar)
endef
```

七、环境变量

make运行时的系统环境变量可以在 make开始运行时被载入到 Makefile文件中，但是如果 Makefile中已定义了这个变量，或是这个变量由 make命令行带入，那么系统的环境变量的值将被覆盖。（如果 make指定了“ -e”参数，那么，系统环境变量将覆盖 Makefile中定义的变量）

因此，如果我们在环境变量中设置了“ CFLAGS”环境变量，那么我们就可以在所有的 Makefile中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果 Makefile中定义了 CFLAGS，那么则会使用 Makefile中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“ 全局变量 ”和“ 局部变量 ”的特性。

当 make嵌套调用时（参见前面的“ 嵌套调用 ”章节），上层 Makefile中定义的变量会以系统环境变量的方式传递到下层的 Makefile中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层 Makefile传递，则需要使用 exprot关键字来声明。（参见前面章节）

当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的 Makefile时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

八、目标变量

前面我们所讲的在 Makefile中定义的变量都是“ 全局变量 ”，在整个文件，我们都可以访问这些变量。当然，“ 自动化变量 ”除外，如“ $<”等这种类量的自动化变量就属于“ 规则型变量 ”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我样同样可以为某个目标设置局部变量，这种变量被称为“ Target-specific Variable”，它可以和“ 全局变量 ”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

```
<target ...> : <variable-assigment>
<target ...> : overide <variable-assigment>
```

<variable-assigment>可以是前面讲过的各种赋值表达式，如“ =”、“ :=”、“ +=”或是“ ? =”。第二个语法是针对于 make命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
	$(CC) $(CFLAGS) prog.o foo.o bar.o
prog.o : prog.c
	$(CC) $(CFLAGS) prog.c
foo.o : foo.c
	$(CC) $(CFLAGS) foo.c
bar.o : bar.c
	$(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的$(CFLAGS)的值是什么，在 prog目标，以及其所引发的所有规则中（prog.o foo.o bar.o 的规则），$(CFLAGS)的值都是“ -g”

九、模式变量

在 GNU的 make中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

```
%.o : CFLAGS = -O
```

同样，模式变量的语法和“目标变量”一样：

```
<pattern ...> : <variable-assigmment>
<pattern ...> : override <variable-assigmment>
```

override同样是针对于系统环境传入的变量，或是 make命令行指定的变量。

使用条件判断

使用条件判断，可以让 make根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

一、示例

下面的例子，判断$(CC)变量是否“ gcc”，如果是的话，则使用 GNU函数编译目标。

```
libs_for_gcc = -lgnu
normal_libs =
foo: $(objects)
ifeq ($(CC),gcc)
	$(CC) -o foo $(objects) $(libs_for_gcc)
else
	$(CC) -o foo $(objects) $(normal_libs)
endif
```

可见，在上面示例的这个规则中，目标“ foo”可以根据变量“ $(CC)”值来选取不同的函数库来编译程序。

我们可以从上面的示例中看到三个关键字：ifeq、else和 endif。ifeq的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else表示条件表达式为假的情况。endif表示一个条件语句的结束，任何一个条件表达式都应该以 endif结束。

当我们的变量$(CC)值是“ gcc”时，目标 foo的规则是：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

而当我们的变量$(CC)值不是“ gcc”时（比如“ cc”），目标 foo的规则是：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

当然，我们还可以把上面的那个例子写得更简洁一些：

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif
foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

二、语法

条件表达式的语法为：

```
<conditional-directive>
<text-if-true>
endif
```

以及：

```
<conditional-directive>
<text-if-true>
else
<text-if-false>
endif
```

其中<conditional-directive>表示条件关键字，如“ ifeq”。这个关键字有四个。

第一个是我们前面所见过的“ ifeq”

```
ifeq (<arg1>, <arg2>)
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"
```

比较参数“ arg1”和“ arg2”的值是否相同。当然，参数中我们还可以使用 make的函数。如：

```
ifeq ($(strip $(foo)),)
<text-if-empty>
endif
```

这个示例中使用了“ strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件关键字是“ ifneq”。语法是：

```
ifneq (<arg1>, <arg2>)
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"
```

其比较参数“ arg1”和“ arg2”的值是否相同，如果不同，则为真。和“ ifeq”类似。

第三个条件关键字是“ ifdef”。语法是：

```
ifdef <variable-name>
```

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然， <variable-name>同样可以是

一个函数的返回值。注意，ifdef只是测试一个变量是否有值，其并不会把变量扩展到当前位置。还是来看两个例子：

示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

第一个例子中，“ $(frobozz)”值是“ yes”，第二个则是“ no”。

第四个条件关键字是“ ifndef”。其语法是：

```
ifndef <variable-name>
```

这个我就不多说了，和“ ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“ #”同样也是安全的。“ else”和“ endif”也一样，只要不是以[Tab]键开始就行了。

特别注意的是，make是在读取 Makefile时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“ $@”等）放入条件表达式中，因为自动化变量是在运行时才有的。

而且，为了避免混乱，make不允许把整个条件语句分成两部分放在不同的文件中。

-

使用函数 ————

在 Makefile中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。make所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

一、函数的调用语法

函数调用，很像变量的使用，也是以“ $”来标识的，其语法如下：

```
$(<function> <arguments>)
```

或是

```
${<function> <arguments>}
```

这里，<function>就是函数名，make支持的函数不多。<arguments>是函数的参数，参数间以逗号“ ,”分隔，而函数名和参数之间以“ 空格 ”分隔。函数调用以“ $”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“ $(subst a,b,$(x))”这样的形式，而不是“ $(subst a,b,${x})”的形式。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)的值是“ a b c”，$(bar)的定义用，调用了函数“ subst”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)

的值是“ a,b,c”。

二、字符串处理函数

$(subst <from>,<to>,<text>)

名称：字符串替换函数——subst。

功能：把字串 <text>中的 <from>字符串替换成 <to>。

返回：函数返回被替换过后的字符串。

示例：

$(subst ee,EE,feet on the street),

把“ feet on the street”中的“ ee”替换成“ EE”，返回结果是“ fEEt on the strEEt”。

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数——patsubst。

功能：查找 <text>中的单词（单词以“ 空格 ”、“ Tab”或“ 回车 ”“ 换行 ”分隔）是否符合模式 <pattern>，如果匹配的话，则以 <replacement>替换。这里， <pattern>可以包括通配符“ %”，表示任意长度的字串。如果 <replacement>中也包含“ %”，那么， <replacement>中的这个“ %”将是 <pattern>中的那个“ %”所代表的字串。（可以用“ \”来转义，以“ \%”来表示真实含义的“ %”字符）

返回：函数返回被替换过后的字符串。

示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“ x.c.c bar.c”符合模式 [%.c]的单词替换成 [%.o]，返回结果是“ x.c.o bar.o”

备注：

这和我们前面“ 变量章节 ”说过的相关知识有点相似。如：

“ $(var:<pattern>=<replacement>)”

相当于

“ $(patsubst <pattern>,<replacement>,$(var))”，

而“ $(var: <suffix>=<replacement>)”

则相当于

“ $(patsubst %<suffix>,%<replacement>,$(var))”。

例如有： objects = foo.o bar.o baz.o，

那么，“ $(objects:.o=.c)”和“ $(patsubst %.o,%.c,$(objects))”是一样的。

$(strip <string>)

名称：去空格函数——strip。

功能：去掉 <string>字串中开头和结尾的空字符。

返回：返回被去掉空格的字符串值。

示例：

$(strip a b c)

把字串“ a b c ”去到开头和结尾的空格，结果是“ a b c”。

$(findstring <find>,<in>)

名称：查找字符串函数——findstring。

功能：在字串 <in>中查找 <find>字串。

返回：如果找到，那么返回 <find>，否则返回空字符串。

示例：

$(findstring a,a b c)

$(findstring a,b c)

第一个函数返回“ a”字符串，第二个返回“ ”字符串（空字符串）

$(filter <pattern...>,<text>)

名称：过滤函数——filter。

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>的字串。

示例：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

$(filter %.c %.s,$(sources))返回的值是“ foo.c bar.c baz.s”。

$(filter-out <pattern...>,<text>)

名称：反过滤函数——filter-out。

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

返回：返回不符合模式<pattern>的字串。

示例：

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

$(filter-out $(mains),$(objects)) 返回值是“ foo.o bar.o”。

$(sort <list>)

名称：排序函数——sort。

功能：给字符串<list>中的单词排序（升序）。

返回：返回排序后的字符串。

示例：$(sort foo bar lose)返回“ bar foo lose” 。

备注： sort函数会去掉<list>中相同的单词。

$(word <n>,<text>)

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。

示例：$(word 2, foo bar baz)返回值是“ bar”。

$(wordlist <s>,<e>,<text>)

名称：取单词串函数——wordlist。

功能：从字符串<text>中取从<s>开始到<e>的单词串。 <s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。

示例： $(wordlist 2, 3, foo bar baz)返回值是“ bar baz”。

$(words <text>)

名称：单词个数统计函数——words。

功能：统计<text>中字符串中的单词个数。

返回：返回<text>中的单词数。

示例：$(words, foo bar baz)返回值是“ 3”。

备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words <text>),<text>)。

$(firstword <text>)

名称：首单词函数——firstword。

功能：取字符串<text>中的第一个单词。

返回：返回字符串<text>的第一个单词。

示例：$(firstword foo bar)返回值是“ foo”。

备注：这个函数可以用word函数来实现：$(word 1,<text>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里，举一个现实中应用的例

子。我们知道，make使用“ VPATH”变量来指定“ 依赖文件 ”的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数 CFLAGS，如：

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

如果我们的“ $(VPATH)”值是“ src:../headers”，那么“ $(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“ -Isrc -I../headers”，这正是 cc或 gcc搜索头文件路径的参数。

三、文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“ /”）之前的部分。如果没有反斜杠，那么返回“ ./”。

返回：返回文件名序列<names>的目录部分。

示例： $(dir src/foo.c hacks)返回值是“ src/ ./”。

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“ /”）之后的部分。

返回：返回文件名序列<names>的非目录部分。

示例： $(notdir src/foo.c hacks)返回值是“ foo.c hacks”。

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列<names>中取出各个文件名的后缀。

返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。

示例： $(suffix src/foo.c src-1.0/bar.c hacks)返回值是“ .c .c”。

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列<names>中取出各个文件名的前缀部分。

返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。

示例： $(basename src/foo.c src-1.0/bar.c hacks)返回值是“ src/foo src-1.0/bar hacks”。

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀<suffix>加到<names>中的每个单词后面。

返回：返回加过后缀的文件名序列。

示例： $(addsuffix .c,foo bar)返回值是“ foo.c bar.c”。

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

示例： $(addprefix src/,foo bar)返回值是“ src/foo src/bar”。

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

返回：返回连接过后的字符串。

示例： $(join aaa bbb , 111 222 333)返回值是“ aaa111 bbb222 333”。

---

gunguymadman 回复于：2004-09-16 12:24:08

四、 foreach 函数

foreach函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile中的 foreach函数几乎是仿照于 Unix标准 Shell（/bin/sh）中的 for语句，或是 C-Shell（/bin/csh）中的 foreach语句而构建的。它的语法是：

```
$(foreach <var>,<list>,<text>)
```

这个函数的意思是，把参数 <list>中的单词逐一取出放到参数 <var>所指定的变量中，然后再执行 <text>所包含的表达式。每一次 <text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach函数的返回值。

所以，<var>最好是一个变量名，<list>可以是一个表达式，而 <text>中一般会使用 <var>这个参数来依次枚举 <list>中的单词。举个例子：

```
names := a b c d
files := $(foreach n,$(names),$(n).o)
```

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“ n”中，“ $(n).o”每次根据“ $(n)”计算出一个值，这些值以空格分隔，最后作为 foreach函数的返回，所以，$(files)的值是“ a.o b.o c.o d.o”。

注意，foreach中的 <var>参数是一个临时的局部变量，foreach函数执行完后，参数 <var>的变量将不在作用，其作用域只在 foreach函数当中。

五、 if 函数

if函数很像 GNU的 make所支持的条件语句——ifeq（参见前面所述的章节），if函数的语法是：

```
$(if <condition>,<then-part>)
```

或是

```
$(if <condition>,<then-part>,<else-part>)
```

可见，if 函数可以包含“ else”部分，或是不含。即 if 函数的参数可以是两个，也可以是三个。<condition>参数是 if的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则 <else-part>会被计算。

而 if 函数的返回值是，如果 <condition>为真（非空字符串），那个 <then-part>会是整个函数的返回值，如果 <condition>为假（空字符串），那么 <else-part>会是整个函数的返回值，此时如果 <else-part>没有被定义，那么，整个函数返回空字串。

所以，<then-part>和 <else-part>只会有一个被计算。

六、 call函数

call函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用 call函数来向这个表达式传递参数。其语法是：

```
$(call <expression>,<parm1>,<parm2>,<parm3>...)
```

当 make执行这个函数时， <expression>参数中的变量，如 $(1)， $(2)， $(3)等，会被参数 <parm1>， <parm2>， <parm3>依次取代。而 <expression>的返回值就是 call函数的返回值。例如：

```
reverse =  $(1) $(2)
foo = $(call reverse,a,b)
```

那么， foo的值就是“ a b”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

```
reverse =  $(2) $(1)
foo = $(call reverse,a,b)
```

此时的 foo的值就是“ b a”。

七、 origin函数

origin函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的？其语法是：

```
$(origin <variable>)
```

注意， <variable>是变量的名字，不应该是引用。所以你最好不要在 <variable>中使用“ $”字符。 Origin函数会以其返回值来告诉你这个变量的“ 出生情况 ”，下面，是 origin函数的返回值：

“ undefined”
如果 <variable>从来没有定义过， origin函数返回这个值“ undefined”。

“ default”
如果 <variable>是一个默认的定义，比如“ CC”这个变量，这种变量我们将在后面讲述。

“ environment”
如果 <variable>是一个环境变量，并且当 Makefile被执行时，“ -e”参数没有被打开。

“ file”
如果 <variable>这个变量被定义在 Makefile中。

“ command line”
如果 <variable>这个变量是被命令行定义的。

“ override”
如果 <variable>是被 override指示符重新定义的。

“ automatic”
如果 <variable>是一个命令运行中的自动化变量。关于自动化变量将在后面讲述。

这些信息对于我们编写 Makefile是非常有用的,例如,假设我们有一个 Makefile其包了一个定义文件 Make.def,在 Make.def中定义了一个变量“ bletch”,而我们的环境中也有一个环境变量“ bletch”,此时,我们想判断一下,如果变量来源于环境,那么我们就把之重定义了,如果来源于 Make.def或是命令行等非环境的,那么我们就不重新定义它。于是,在我们的 Makefile中,我们可以这样写:

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

当然,你也许会说,使用 override关键字不就可以重新定义环境中的变量了吗?为什么需要使用这样的步骤?是的,我们用 override是可以达到这样的效果,可是 override过于粗暴,它同时会把从命令行定义的变量也覆盖了,而我们只想重新定义环境传来的,而不想重新定义命令行传来的。

八、 shell函数

shell函数也不像其它的函数。顾名思义,它的参数应该就是操作系统 Shell的命令。它和反引号“ `”是相同的功能。这就是说, shell函数把执行操作系统命令后的输出作为函数返回。于是,我们可以用操作系统命令以及字符串处理命令 awk, sed等等命令来生成一个变量,如:

```
contents := $(shell cat foo)

files := $(shell echo *.c)
```

注意,这个函数会新生成一个 Shell程序来执行命令,所以你要注意其运行性能,如果你的 Makefile中有一些比较复杂的规则,并大量使用了这个函数,那么对于你的系统性能是有害的。特别是 Makefile的隐晦的规则可能会让你的 shell函数执行的次数比你想像的多得多。

九、控制 make的函数

make提供了一些函数来控制 make的运行。通常,你需要检测一些运行 Makefile时的运行时信息,并且根据这些信息来决定,你是让 make继续执行,还是停止。

```
$(error <text ...>)
```

产生一个致命的错误, <text ...>是错误信息。注意, error函数不会在一被使用就会产生错误信息,所以如果你把其定义在某个变量中,并在后续的脚本中使用这个变量,那么也是可以的。例如:

```
示例一:
ifdef ERROR_001
$(error error is $(ERROR_001))
endif

示例二:
ERR = $(error found an error!)
.PHONY: err
```

```
err: ; $(ERR)
```

示例一会在变量 ERROR_001定义了后执行时产生 error调用，而示例二则在目录 err被执行时才发生 error调用。

```
$(warning <text ...>)
```

这个函数很像 error函数，只是它并不会让 make退出，只是输出一段警告信息，而 make继续执行。

make 的运行 ————————

一般来说，最简单的就是直接在命令行下输入 make命令，make命令会找当前目录的 makefile来执行，一切都是自动的。但也有时你也许只想让 make重编译某些文件，而不是整个工程，而又有的时候你有几套编译规则，你想在不同的时候使用不同的编译规则，等等。本章节就是讲述如何使用 make命令的。

一、make的退出码

make命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果 make运行时出现任何错误，其返回 1。

2 —— 如果你使用了 make的“ -q”选项，并且 make使得一些目标不需要更新，那么返回 2。

Make的相关参数我们会在后续章节中讲述。

二、指定 Makefile

前面我们说过， GNU make 找寻默认的 Makefile 的规则是在当前目录下依次找三个文件——“ GNUmakefile”、“ makefile”和“ Makefile”。其按顺序找这三个文件，一旦找到，就开始读取这个文件并执行。

当前，我们也可以给 make命令指定一个特殊名字的 Makefile。要达到这个功能，我们要使用 make的“ -f”或是“ --file”参数（“ --makefile”参数也行）。例如，我们有个 makefile的名字是“ hchen.mk”，那么，我们可以这样来让 make来执行这个文件：

```
make – f hchen.mk
```

如果在 make的命令行是，你不只一次地使用了“ -f”参数，那么，所有指定的 makefile将会被连在一起传递给 make执行。

三、指定目标

一般来说，make的最终目标是 makefile中的第一个目标，而其它目标一般是由这个目标连带出来的。这是 make的默认行为。当然，一般来说，你的 makefile中的第一个目标是由许多个目标组成，你可以指示 make，让其完成你所指定的目标。要达到这一目的很简单，需在 make命令后直接跟目标的名字就可以完成（如前面提到的“ make clean”形式）

任何在 makefile中的目标都可以被指定成终极目标，但是除了以“ -”打头，或是包含了“ =”的目标，因为有这些字符的目标，会被解析成命令行参数或是变量。甚至没有被我们明确写出来的目标也可以成为 make的终极目标，也就是说，只要 make可以找到其隐含规则推导规则，那么这个隐含目标同样可以被指定成终极目标。

有一个 make的环境变量叫“ MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

```
sources = foo.c bar.c
ifneq ( $(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

基于上面的这个例子，只要我们输入的命令不是“ make clean”，那么 makefile会自动包含“ foo.d”和“ bar.d”这两个 makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

```
.PHONY: all
all: prog1 prog2 prog3 prog4
```

从这个例子中，我们可以看到，这个 makefile中有四个需要编译的程序——“ prog1”， “ prog2”， “ prog3”和“ prog4”，我们可以使用“ make all”命令来编译所有的目标（如果把 all置成第一个目标，那么只需执行“ make”），我们也可以使用“ make prog2”来单独编译目标“ prog2”。

即然 make 可以指定所有 makefile 中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的 makefile根据指定的不同的目标来完成不同的事。在 Unix世界中，软件发布时，特别是 GNU这种开源软件的发布时，其 makefile都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的 makefile中的目标。

“ all”

这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

“ clean”

这个伪目标功能是删除所有被 make创建的文件。

“ install”

这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。

“ print”

这个伪目标的功能是例出改变过的源文件。

“ tar”

这个伪目标功能是把源程序打包备份。也就是一个 tar文件。

“ dist”

这个伪目标功能是创建一个压缩文件，一般是把 tar文件压成 Z文件。或是 gz文件。

“ TAGS”

这个伪目标功能是更新所有的目标，以备完整地重编译使用。

“ check”和“ test”

这两个伪目标一般用来测试 makefile的流程。

当然一个项目的 makefile中也不一定要书写这样的目标，这些东西都是 GNU的东西，但是我想，GNU搞出这些东西一定有其可取之处（等你的 UNIX下的程序文件一多时你就会发现这些功能很有用了），这里只不过是说明了，如果你要书写这种功能，最好使用这种名字命名你的目标，这样规范一些，规范的好处就是——不用解释，大家都明白。而且如果你的 makefile中有这些功能，一是很实用，二是可以显得你的 makefile很专业（不是那种初学者的作品）。

四、检查规则

有时候，我们不想让我们的 makefile中的规则执行起来，我们只想检查一下我们的命令，或是执行的序列。于是我们可以使用 make命令的下述参数：

“ -n”

“ --just-print”

“ --dry-run”

“ --recon”

不执行参数，这些参数只是打印命令，不管目标是否更新，把规则和连带规则下的命令打印出来，但不执行，这些参数对于我们调试 makefile很有用处。

“ -t”

“ --touch”

这个参数的意思就是把目标文件的时间更新，但不更改目标文件。也就是说，make假装编译目标，但不是真正的编译目标，只是把目标变成已编译过的状态。

“ -q”

“ --question”

这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出，当然也不会执行编译，如果目标不存在，其会打印出一条出错信息。

“ -W <file>”

“ --what-if=<file>”

“ --assume-new=<file>”

“ --new-file=<file>”

这个参数需要指定一个文件。一般是是源文件（或依赖文件），Make会根据规则推导来运行依赖于这个文件

的命令，一般来说，可以和“ -n”参数一同使用，来查看这个依赖文件所发生的规则命令。

另外一个很有意思的用法是结合“ -p”和“ -v”来输出 makefile被执行时的信息（这个将在后面讲述）。

五、make的参数

下面列举了所有 GNU make 3.80版的参数定义。其它版本和产商的 make大同小异，不过其它产商的 make的具体参数还是请参考各自的产品文档。

“ -b”

“ -m”

这两个参数的作用是忽略和其它版本 make的兼容性。

“ -B”

“ --always-make”

认为所有的目标都需要更新（重编译）。

“ -C <dir>”

“ --directory=<dir>”

指定读取 makefile的目录。如果有多个“ -C”参数，make的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。如：“ make - C ~hchen/test - C prog”等价于“ make - C ~hchen/test/prog”。

“ — debug[=<options>]”

输出 make的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —— 也就是 all，输出所有的调试信息。（会非常的多）

b —— 也就是 basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —— 也就是 verbose，在 b选项的级别之上。输出的信息包括哪个 makefile被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —— 也就是 implicit，输出所以的隐含规则。

j —— 也就是 jobs，输出执行规则中命令的详细信息，如命令的 PID、返回码等。

m —— 也就是 makefile，输出 make读取 makefile，更新 makefile，执行 makefile的信息。

“ -d”

相当于“ --debug=a”。

“ -e”

“ --environment-overrides”

指明环境变量的值覆盖 makefile中定义的变量的值。

“ -f=<file>”

“ --file=<file>”

“ --makefile=<file>”

指定需要执行的 makefile。

“ -h”

“ --help”

显示帮助信息。

“ -i”

“ --ignore-errors”

在执行时忽略所有的错误。

“ -I <dir>”

“ --include-dir=<dir>”

指定一个被包含 makefile的搜索目标。可以使用多个“ -I”参数来指定多个目录。

“ -j [<jobsnum>]”

“ --jobs[=<jobsnum>]”

指同时运行命令的个数。如果没有这个参数，make运行命令时能运行多少就运行多少。如果有一个以上的“ -j”参数，那么仅最后一个“ -j”才是有效的。（注意这个参数在 MS-DOS中是无用的）

“ -k”

“ --keep-going”
出错也不停止运行。如果生成一个目标失败了，那么依赖于其上的目标就不会被执行了。
“ -l <load>”
“ --load-average[=<load]”
“ —max-load[=<load>]”
指定 make运行命令的负载。
“ -n”
“ --just-print”
“ --dry-run”
“ --recon”
仅输出执行过程中的命令序列，但并不执行。
“ -o <file>”
“ --old-file=<file>”
“ --assume-old=<file>”
不重新生成的指定的 <file>，即使这个目标的依赖文件新于它。
“ -p”
“ --print-data-base”
输出 makefile中的所有数据，包括所有的规则和变量。这个参数会让一个简单的 makefile都会输出一堆信息。如果你只是想输出信息而不想执行 makefile，你可以使用“ make -qp”命令。如果你想查看执行 makefile前的预设变量和规则，你可以使用“ make - p - f /dev/null”。这个参数输出的信息会包含着你的 makefile文件的文件名和行号，所以，用这个参数来调试你的 makefile会是很有用的，特别是当你的环境变量很复杂的时候。
“ -q”
“ --question”
不运行命令，也不输出。仅仅是检查所指定的目标是否需要更新。如果是 0则说明要更新，如果是 2则说明有错误发生。
“ -r”
“ --no-builtin-rules”
禁止 make使用任何隐含规则。
“ -R”
“ --no-builtin-variabes”
禁止 make使用任何作用于变量上的隐含规则。
“ -s”
“ --silent”
“ --quiet”
在命令运行时不输出命令的输出。
“ -S”
“ --no-keep-going”
“ --stop”
取消“ -k”选项的作用。因为有些时候，make的选项是从环境变量“ MAKEFLAGS”中继承下来的。所以你可以在命令行中使用这个参数来让环境变量中的“ -k”选项失效。
“ -t”
“ --touch”
相当于 UNIX的 touch命令，只是把目标的修改日期变成最新的，也就是阻止生成目标的命令运行。
“ -v”
“ --version”
输出 make程序的版本、版权等关于 make的信息。
“ -w”
“ --print-directory”
输出运行 makefile之前和之后的信息。这个参数对于跟踪嵌套式调用 make时很有用。

“ --no-print-directory”

禁止“ -w”选项。

“ -W <file>”

“ --what-if=<file>”

“ --new-file=<file>”

“ --assume-file=<file>”

假定目标<file>需要更新，如果和“ -n”选项使用，那么这个参数会输出该目标更新时的运行动作。如果没有“ -n”那么就像运行 UNIX的“ touch”命令一样，使得<file>的修改时间为当前时间。

“ --warn-undefined-variables”

只要 make发现有未定义的变量，那么就输出警告信息。

隐含规则 ————

在我们使用 Makefile时，有一些我们会经常使用，而且使用频率非常高的东西，比如，我们编译 C/C++的源程序为中间目标文件（Unix下是[.o]文件，Windows下是[.obj]文件）。本章讲述的就是一些在 Makefile中的“隐含的”，早先约定了的，不需要我们再写出来的规则。

“隐含规则”也就是一种惯例，make会按照这种“惯例”心照不喧地来运行，那怕我们的 Makefile中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make会自动推导出这种规则，并生成我们需要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“ CFLAGS”可以控制编译时的编译器参数。

我们还可以通过“模式规则”的方式写下自己的隐含规则。用“后缀规则”来定义隐含规则会有许多的限制。使用“模式规则”会更回得智能和清楚，但“后缀规则”可以用来保证我们 Makefile的兼容性。

我们了解了“隐含规则”，可以让其为我们更好的服务，也会让我们知道一些“约定俗成”了的东西，而不至于使得我们在运行 Makefile时出现一些我们觉得莫名其妙的东西。当然，任何事物都是矛盾的，水能载舟，亦可覆舟，所以，有时候“隐含规则”也会给我们造成不小的麻烦。只有了解了它，我们才能更好地使用它。

一、使用隐含规则

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make会试图去自动推导产生这个目标的规则和命令，如果 make可以自动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自动推导。当然，隐含规则是 make事先约定好的一些东西。例如，我们有下面的一个 Makefile:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

我们可以注意到，这个 Makefile中并没有写下如何生成 foo.o和 bar.o这两目标的规则和命令。因为 make的“隐含规则”功能会自动为我们自动去推导这两个目标的依赖目标和生成命令。

make会在自己的“隐含规则”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用 C的编译命令“ cc -c $(CFLAGS) [.c]”来生成[.o]的目标。也就是说，我们完全没有必要写下下面的两条规则：

```
foo.o : foo.c
        cc -c foo.c $(CFLAGS)
bar.o : bar.c
    cc -c bar.c $(CFLAGS)
```

因为，这已经是“约定”好了的事了，make和我们约定好了用 C编译器“ cc”生成[.o]文件的规则，这就是隐含规则。

当然，如果我们为[.o]文件书写了自己的规则，那么 make就不会自动推导并调用隐含规则，它会按照我们写好的规则忠实地执行。

还有，在 make的“隐含规则库”中，每一条隐含规则都在库中有其顺序，越靠前的则是越被经常使用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make也不会管。如下面这条规则（没有命令）：

```
foo.o : foo.p
```

依赖文件“ foo.p”（Pascal程序的源文件）有可能变得没有意义。如果目录下存在了“ foo.c”文件，那么我们

的隐含规则一样会生效，并会通过“ foo.c”调用 C的编译器生成 foo.o文件。因为，在隐含规则中， Pascal的规则出现在 C的规则之后，所以，make找到可以生成 foo.o的 C的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“ 依赖规则 ”，而不写命令。

二、隐含规则一览

这里我们将讲述所有预先设置（也就是 make内建）的隐含规则，如果我们不明确地写下规则，那么，make就会在这些规则中寻找所需要规则和命令。当然，我们也可以使用 make的参数“ -r”或“ --no-builtin-rules”选项来取消所有的预设置的隐含规则。

当然，即使是我们指定了“ -r”参数，某些隐含规则还是会生效，因为有许多的隐含规则都是使用了“ 后缀规则 ”来定义的，所以，只要隐含规则中有“ 后缀列表 ”（也就一系统定义在目标 .SUFFIXES的依赖目标），那么隐含规则就会生效。默认的后缀列表是： .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi , .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el。具体的细节，我们会在后面讲述。

还是先来看一看常用的隐含规则吧。

1、编译 C程序的隐含规则。

“ <n>.o”的目标的依赖目标会自动推导为“ <n>.c”，并且其生成命令是“ $(CC) -c $(CPPFLAGS) $(CFLAGS)”

2、编译 C++程序的隐含规则。

“ <n>.o”的目标的依赖目标会自动推导为“ <n>.cc”或是“ <n>.C”，并且其生成命令是“ $(CXX) -c $(CPPFLAGS) $(CFLAGS)”。（建议使用“ .cc”作为 C++源文件的后缀，而不是“ .C”）

3、编译 Pascal程序的隐含规则。

“ <n>.o”的目标的依赖目标会自动推导为“ <n>.p”，并且其生成命令是“ $(PC) -c $(PFLAGS)”。

4、编译 Fortran/Ratfor程序的隐含规则。

“ <n>.o”的目标的依赖目标会自动推导为“ <n>.r”或“ <n>.F”或“ <n>.f”，并且其生成命令是：

“ .f” “ $(FC) -c $(FFLAGS)”

“ .F” “ $(FC) -c $(FFLAGS) $(CPPFLAGS)”

“ .f” “ $(FC) -c $(FFLAGS) $(RFLAGS)”

5、预处理 Fortran/Ratfor程序的隐含规则。

“ <n>.f”的目标的依赖目标会自动推导为“ <n>.r”或“ <n>.F”。这个规则只是转换 Ratfor或有预处理的 Fortran程序到一个标准的 Fortran程序。其使用的命令是：

“ .F” “ $(FC) -F $(CPPFLAGS) $(FFLAGS)”

“ .r” “ $(FC) -F $(FFLAGS) $(RFLAGS)”

6、编译 Modula-2程序的隐含规则。

“ <n>.sym”的目标的依赖目标会自动推导为“ <n>.def”，并且其生成命令是：“ $(M2C) $(M2FLAGS) $(DEFFLAGS)”。“ <n.o>” 的目标的依赖目标会自动推导为“ <n>.mod”，并且其生成命令是：“ $(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。

“ <n>.o” 的目标的依赖目标会自动推导为“ <n>.s”，默认使用编译品“ as”，并且其生成命令是：“ $(AS) $(ASFLAGS)”。“ <n>.s” 的目标的依赖目标会自动推导为“ <n>.S”，默认使用 C预编译器“ cpp”，并且其生成命令是：“ $(AS) $(ASFLAGS)”。

8、链接 Object文件的隐含规则。

“ <n>”目标依赖于“ <n>.o”，通过运行 C的编译器来运行链接程序生成（一般是“ ld”），其生成命令是：“ $(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个 Object文件（由不同的源文件生成）的也有效。例如如下规则：

```
x : y.o z.o
```

并且“ x.c”、“ y.c”和“ z.c”都存在时，隐含规则将执行如下命令：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
```

```
rm -f y.o
rm -f z.o
```

如果没有一个源文件（如上例中的 x.c）和你的目标名字（如上例中的 x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9. Yacc C程序时的隐含规则。

“ <n>.c”的依赖文件被自动推导为“ n.y”（Yacc生成的文件），其生成命令是：“ $(YACC) $(YFALGS)”。（“ Yacc”是一个语法分析器，关于其细节请查看相关资料）

10. Lex C程序时的隐含规则。

“ <n>.c”的依赖文件被自动推导为“ n.l”（Lex生成的文件），其生成命令是：“ $(LEX) $(LFALGS)”。（关于“ Lex”的细节请查看相关资料）

11. Lex Ratfor程序时的隐含规则。

“ <n>.r”的依赖文件被自动推导为“ n.l”（Lex生成的文件），其生成命令是：“ $(LEX) $(LFALGS)”。

12. 从 C程序、Yacc文件或 Lex文件创建 Lint库的隐含规则。

“ <n>.ln” （lint 生成的文件）的依赖文件被自动推导为“ n.c”，其生成命令是：“ $(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“ <n>.y”和“ <n>.l”也是同样的规则。

三、隐含规则使用的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的 makefile中改变这些变量的值，或是在 make的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也可以利用 make的“ -R”或“ --no- builtin-variables”参数来取消你所定义的变量对隐含规则的作用。

例如，第一条隐含规则——编译 C程序的隐含规则的命令是“ $(CC) - c $(CFLAGS) $(CPPFLAGS)”。Make默认的编译命令是“ cc”，如果你把变量“ $(CC)”重定义成“ gcc”，把变量“ $(CFLAGS)”重定义成“ -g”，那么，隐含规则中的命令全部会以“ gcc - c -g $(CPPFLAGS)”的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如“ CC”；一种是参数相的关，如“ CFLAGS”。下面是所有隐含规则中会用到的变量：

1. 关于命令的变量。

AR
函数库打包程序。默认命令是“ ar”。

AS
汇编语言编译程序。默认命令是“ as”。

CC
C语言编译程序。默认命令是“ cc”。

CXX
C++语言编译程序。默认命令是“ g++”。

CO
从 RCS文件中扩展文件程序。默认命令是“ co”。

CPP
C程序的预处理器（输出是标准输出设备）。默认命令是“ $(CC) - E”。

FC
Fortran和 Ratfor 的编译器和预处理程序。默认命令是“ f77”。

GET
从 SCCS文件中扩展文件的程序。默认命令是“ get”。

LEX
Lex方法分析器程序（针对于 C或 Ratfor）。默认命令是“ lex”。

PC
Pascal语言编译程序。默认命令是“ pc”。

YACC
Yacc文法分析器（针对于 C程序）。默认命令是“ yacc”。

YACCR
Yacc文法分析器（针对于 Ratfor程序）。默认命令是“ yacc – r”。
MAKEINFO
转换 Texinfo源文件（.texi）到 Info文件程序。默认命令是“ makeinfo”。
TEX
从 TeX源文件创建 TeX DVI文件的程序。默认命令是“ tex”。
TEXI2DVI
从 Texinfo源文件创建军 TeX DVI 文件的程序。默认命令是“ texi2dvi”。
WEAVE
转换 Web到 TeX的程序。默认命令是“ weave”。
CWEAVE
转换 C Web 到 TeX的程序。默认命令是“ cweave”。
TANGLE
转换 Web到 Pascal语言的程序。默认命令是“ tangle”。
CTANGLE
转换 C Web 到 C。默认命令是“ ctangle”。
RM
删除文件命令。默认命令是“ rm – f”。
2、关于命令参数的变量
下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。
ARFLAGS
函数库打包程序 AR命令的参数。默认值是“ rv”。
ASFLAGS
汇编语言编译器参数。（当明显地调用“ .s”或“ .S”文件时）。
CFLAGS
C语言编译器参数。
CXXFLAGS
C++语言编译器参数。
COFLAGS
RCS命令参数。
CPPFLAGS
C预处理器参数。（ C和 Fortran 编译器也会用到）。
FFLAGS
Fortran语言编译器参数。
GFLAGS
SCCS “ get”程序参数。
LDFLAGS
链接器参数。（如：“ ld”）
LFLAGS
Lex文法分析器参数。
PFLAGS
Pascal语言编译器参数。
RFLAGS
Ratfor 程序的 Fortran 编译器参数。
YFLAGS
Yacc文法分析器参数。

四、隐含规则链
有些时候，一个目标可能被一系列的隐含规则所作用。例如，一个 [.o]的文件生成，可能会是先被 Yacc的 [.y]

文件先成 [.c]，然后再被 C的编译器生成。我们把这一系列的隐含规则叫做“ 隐含规则链 ”。

在上面的例子中，如果文件 [.c]存在，那么就直接调用 C的编译器的隐含规则，如果没有 [.c]文件，但有一个 [.y]文件，那么 Yacc的隐含规则会被调用，生成 [.c]文件，然后，再调用 C编译的隐含规则最终由 [.c]生成 [.o]文件，达到目标。

我们把这种 [.c]的文件（或是目标），叫做中间目标。不管怎么样，make会努力自动推导生成目标的一切方法，不管中间目标有多少，其都会执着地把所有的隐含规则和你书写的规则全部合起来分析，努力达到目标，所以，有些时候，可能会让你觉得奇怪，怎么我的目标会这样生成？怎么我的 makefile发疯了？

在默认情况下，对于中间目标，它和一般的目标有两个地方所不同：第一个不同是除非中间的目标不存在，才会引发中间规则。第二个不同的是，只要目标成功产生，那么，产生最终目标过程中，所产生的中间目标文件会被以“ rm -f”删除。

通常，一个被 makefile指定成目标或是依赖目标的文件不能被当作中介。然而，你可以明显地说明一个文件或是目标是中介目标，你可以使用伪目标“ .INTERMEDIATE”来强制声明。（如： .INTERMEDIATE ： mid）

你也可以阻止 make 自动删除中间目标，要做到这一点，你可以使用伪目标“ .SECONDARY”来强制声明（如： .SECONDARY ： sec）。你还可以把你的目标，以模式的方式来指定（如：%.o）成伪目标“ .PRECIOUS”的依赖目标，以保存被隐含规则所生成的中间文件。

在“ 隐含规则链 ”中，禁止同一个目标出现两次或两次以上，这样一来，就可防止在 make自动推导时出现无限递归的情况。

Make会优化一些特殊的隐含规则，而不生成中间文件。如，从文件“ foo.c”生成目标程序“ foo”，按道理，make会编译生成中间文件“ foo.o”，然后链接成“ foo”，但在实际情况下，这一动作可以被一条“ cc”的命令完成（cc -o foo foo.c），于是优化过的规则就不会生成中间文件。

五、定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有 '%'字符。'%'的意思是表示一个或多个任意字符。在依赖目标中同样可以使用 '%', 只是依赖目标中的 '%'的取值，取决于其目标。

有一点需要注意的是， '%'的展开发生在变量和函数的展开之后，变量和函数的展开发生在 make载入 Makefile时，而模式规则中的 '%'则发生在运行时。

1. 模式规则介绍

模式规则中，至少在规则的目标定义中要包含 '%', 否则，就是一般的规则。目标中的 '%'定义表示对文件名的匹配， '%'表示长度任意的非空字符串。例如： '%.c'表示以 ".c"结尾的文件名（文件名的长度至少为 3），而 "s.%.c'则表示以 "s."开头， ".c"结尾的文件名（文件名的长度至少为 5）。

如果 '%'定义在目标中，那么，目标中的 '%'的值决定了依赖目标中的 '%'的值，也就是说，目标中的模式的 '%'决定了依赖目标中 '%'的样子。例如有一个模式规则如下：

```
%.o : %.c ; <command .....>
```

其含义是，指出了怎么从所有的 [.c]文件生成相应的 [.o]文件的规则。如果要生成的目标是 "a.o b.o", 那么 '%c" 就是 "a.c b.c"。

一旦依赖目标中的 '%'模式被确定，那么，make会被要求去匹配当前目录下所有的文件名，一旦找到，make就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make就会产生所有的模式目标，此时，make关心的是依赖的文件名和生成目标的命令这两件事。

2. 模式规则示例

下面这个例子表示了 ,把所有的 [.c]文件都编译成 [.o]文件 .

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

其中，"$@'表示所有的目标的挨个值，"$<'表示了所有依赖目标的挨个值。这些奇怪的变量我们叫 '自动化变量 ", 后面会详细讲述。

下面的这个例子中有两个目标是模式的：

```
%.tab.c %.tab.h: %.y
```

```
bison -d $<
```

这条规则告诉 make把所有的 [.y]文件都以 "bison -d <n>.y"执行，然后生成 "<n>.tab.c"和 "<n>.tab.h"文件。(其中，"<n>"表示一个任意字符串)。如果我们的执行程序 "foo"依赖于文件 "parse.tab.o"和 "scan.o"，并且文件 "scan.o"依赖于文件 "parse.tab.h"，如果 "parse.y"文件被更新了，那么根据上述的规则， "bison -d parse.y"就会被执行一次，于是， "parse.tab.o"和 "scan.o"的依赖文件就齐了。(假设， "parse.tab.o"由 "parse.tab.c"生成，和 "scan.o"由 "scan.c"生成，而 "foo"由 "parse.tab.o"和 "scan.o"链接生成，而且 foo和其 [.o]文件的依赖关系也写好，那么，所有的目标都会得到满足)

3. 自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@

表示规则中的目标文件集。在模式规则中，如果有多个目标，那么， "$@"就是匹配于目标中模式定义的集合。

$%

仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是 "foo.a(bar.o)"，那么， "$%"就是 "bar.o"， "$@"就是 "foo.a"。如果目标不是函数库文件 (Unix下是 [.a]，Windows下是 [.lib])，那么，其值为空。

$<

依赖目标中的第一个目标名字。如果依赖目标是以模式 (即 "%") 定义的，那么 "$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+

这个变量很像 "$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$*

这个变量表示目标模式中 "%"及其之前的部分。如果目标是 "dir/a.foo.b"，并且目标的模式是 "a.%.b"，那么， "$*"的值就是 "dir/a.foo"。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么 "$*"也就不能被推导出，但是，如果目标文件的后缀是 make所识别的，那么 "$*"就是除了后缀的那一部分。例如：如果目标是 "foo.c"，因为 ".c"是 make所能识别的后缀名，所以， "$*"的值就是 "foo"。这个特性是 GNUmake的，很有可能不兼容于其它版本的 make，所以，你应该尽量避免使用 "$*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是 make所不能识别的，那么 "$*"就是空值。

当你希望只对更新过的依赖文件进行操作时， "$?"在显式规则中很有用，例如，假设有一个函数库文件叫 "lib"，其由其它几个 object文件更新。那么把 object文件打包的比较有效率的 Makefile规则是：

```
lib : foo.o bar.o lose.o win.o
        ar r lib $?
```

在上述所列出来的自动量变量中。四个变量 ($@ $< $% $*) 在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上 "D"或 "F"字样。这是 GNUmake中老版本的特性，在新版本中，我们使用函数 "dir"或 "notdir"就可以做到了。 "D"的含义就是 Directory，就是目录， "F"的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上 "D"或是 "F"的含义：

$(@D)

表示 "$@"的目录部分 (不以斜杠作为结尾)，如果 "$@"值是 "dir/foo.o"，那么 "$(@D)"就是 "dir"，而如果 "$@"

中没有包含斜杠的话，其值就是 "."（当前目录）。

$(@F)

表示 "$@"的文件部分，如果 "$@"值是 "dir/foo.o"，那么 "$(@F)"就是 "foo.o"，"$(@F)"相当于函数 "$(notdir $@)"。

"$(*D)"

"$(*F)"

和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(*D)"返回 "dir"，而 "$(*F)" 返回 "foo"

"$(%D)"

"$(%F)"

分别表示了函数包文件成员的目录部分和文件部分。这对于形同 "archive(member)"形式的目标中的 "member"中包含了不同的目录很有用。

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"

"$(+F)"

分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

"$(?D)"

"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于 "$<"，为了避免产生不必要的麻烦，我们最好给 $后面的那个特定字符都加上圆括号，比如，"$(<)"就要比 "$<"要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是 "显式规则 "和 "静态模式规则 "（参见前面 "书写规则 "一章）。其在隐含规则中并没有意义。

4. 模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的 "%"，或是没有前后缀，直接就是一个 "%"。因为 "%"代表一个或多个字符，所以在定义好了的模式中，我们把 "%"所匹配的内容叫做 "茎 "，例如 "%.c"所匹配的文件 "test.c"中 "test"就是 "茎 "。因为在目标和依赖目标中同时有 "%"时，依赖目标的 "茎 "会传给目标，当做目标中的 "茎 "。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行 "茎 "的传递时，我们需要知道这个步骤。例如有一个模式 "e%t"，文件 "src/eat"匹配于该模式，于是 "src/a"就是其 "茎 "，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式 "c%r"，那么，目标就是 "src/car"。（"茎 "被传递）

5. 重载内建隐含规则

你可以重载内建的隐含规则（或是定义一个全新的），例如你可以重新构造和内建隐含规则不同的命令，如：

```
%.o : %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)
```

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

```
%.o : %.s
```

同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

六、老式风格的 "后缀规则 "

后缀规则是一个比较老式的定义隐含规则的方法。后缀规则会被模式规则逐步地取代。因为模式规则更强更清晰。为了和老版本的 Makefile兼容，GNU make同样兼容于这些东西。后缀规则有两种方式："双后缀 "和 "单后缀 "。

双后缀规则定义了一对后缀：目标文件的后缀和依赖目标（源文件）的后缀。如".c.o"相当于"%o：%c"。单后缀规则只定义一个后缀，也就是源文件的后缀。如".c"相当于"%：%.c"。

后缀规则中所定义的后缀应该是 make所认识的，如果一个后缀是 make所认识的，那么这个规则就是单后缀规则，而如果两个连在一起的后缀都被 make所认识，那就是双后缀规则。例如：".c"和".o"都是 make所知道。因而，如果你定义了一个规则是".c.o"那么其就是双后缀规则，意义就是".c"是源文件的后缀，".o"是目标文件的后缀。如下示例：

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则不允许任何的依赖文件，如果有依赖文件的话，那就不是后缀规则，那些后缀统统被认为是文件名，如：

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

这个例子，就是说，文件".c.o"依赖于文件"foo.h"，而不是我们想要的这样：

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则中，如果没有命令，那是毫无意义的。因为他也不会移去内建的隐含规则。

而要让 make知道一些特定的后缀，我们可以使用伪目标".SUFFIXES"来定义或是删除，如：

```
.SUFFIXES: .hack .win
```

把后缀 .hack和 .win加入后缀列表中的末尾。

```
.SUFFIXES:                # 删除默认的后缀
.SUFFIXES: .c .o .h    # 定义自己的后缀
```

先清楚默认后缀，后定义自己的后缀列表。

make的参数"-r"或"-no-builtin-rules"也会使用得默认的后缀列表为空。而变量"SUFFIXE"被用来定义默认的后缀列表，你可以用".SUFFIXES"来改变后缀列表，但请不要改变变量"SUFFIXE"的值。

七、隐含规则搜索算法

比如我们有一个目标叫 T。下面是搜索目标 T的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在 Makefile被载入内存时，会被转换成模式规则。如果目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标 T，如果没有找到的话，那么进入第二次，第二次会把"member"当作 T来搜索。

1. 把 T的目录部分分离出来。叫 D，而剩余部分叫 N。（如：如果 T是"src/foo.o"，那么，D就是"src/"，N就是"foo.o"）
2. 创建所有匹配于 T或是 N的模式规则列表。
3. 如果在模式规则列表中有匹配所有文件的模式，如"%"，那么从列表中移除其它的模式。
4. 移除列表中没有命令的规则。
5. 对于第一个在列表中的模式规则：
   1) 推导其"茎"S，S应该是 T或是 N匹配于模式中"%"非空的部分。
   2) 计算依赖文件。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把 D加在第一个依赖文件的开头。
   3) 测试是否所有的依赖文件都存在或是理当存在。（如果有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）
   4) 如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。
6. 如果经过第 5步，没有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：
   1) 如果规则是终止规则，那就忽略它，继续下一条模式规则。
   2) 计算依赖文件。（同第 5步）
   3) 测试所有的依赖文件是否存在或是理当存在。
   4) 对于不存在的依赖文件，递归调用这个算法查找他是否可以被隐含规则找到。
   5) 如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。
7. 如果没有隐含规则可以使用，查看".DEFAULT"规则，如果有，采用，把".DEFAULT"的命令给 T使用。

一旦规则被找到，就会执行其相当的命令，而此时，我们的自动化变量的值才会生成。

使用 make更新函数库文件 ——————————————

函数库文件也就是对 Object文件（程序编译的中间文件）的打包文件。在 Unix下，一般是由命令 "ar"来完成打包工作。

一、函数库文件的成员

一个函数库文件由多个文件组成。你可以以如下格式指定函数库文件及其组成：

```
archive(member)
```

这个不是一个命令，而一个目标和依赖的定义。一般来说，这种用法基本上就是为了 "ar"命令来服务的。如：

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

如果要指定多个 member，那就以空格分开，如：

```
foolib(hack.o kludge.o)
```

其等价于：

```
foolib(hack.o) foolib(kludge.o)
```

你还可以使用 Shell的文件通配符来定义，如：

```
foolib(*.o)
```

二、函数库成员的隐含规则

当 make搜索一个目标的隐含规则时，一个特殊的特性是，如果这个目标是 "a(m)"形式的，其会把目标变成 "(m)"。于是，如果我们的成员是 "%.o"的模式定义，并且如果我们使用 "make foo.a(bar.o)"的形式调用 Makefile时，隐含规则会去找 "bar.o"的规则，如果没有定义 bar.o的规则，那么内建隐含规则生效，make会去找 bar.c文件来生成 bar.o，如果找得到的话，make执行的命令大致如下：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

还有一个变量要注意的是 "$%"，这是专属函数库文件的自动化变量，有关其说明请参见 "自动化变量 "一节。

三、函数库文件的后缀规则

你可以使用 "后缀规则 "和 "隐含规则 "来生成函数库打包文件，如：

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

其等效于：

```
(%.o) : %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

四、注意事项

在进行函数库打包文件生成时，请小心使用 make的并行机制（ "-j"参数）。如果多个 ar命令在同一时间运行在同一个函数库打包文件上，就很有可以损坏这个函数库文件。所以，在 make未来的版本中，应该提供一种机制来避免并行操作发生在函数打包文件上。

但就目前而言，你还是应该不要尽量不要使用 "-j"参数。