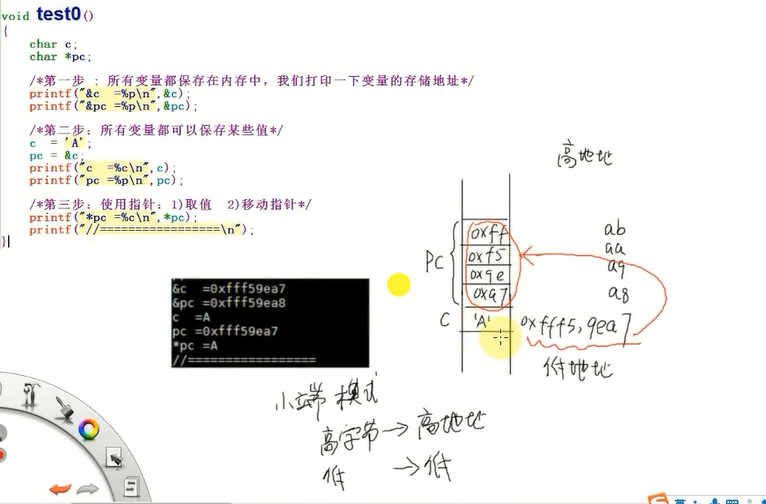
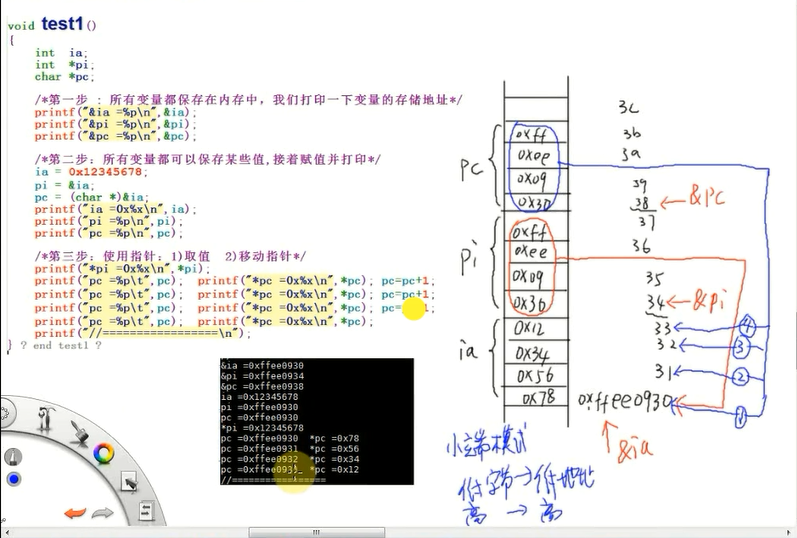
1. C语言指针补充（指向char和int的指针 指向char数组和int数组的指针）





%p:以16进制输出指针的值，附加0x，例如&pc=%p，输出的是指针所在的地址，如果输出数据不足8位，左边补零。

在linux环境下，gcc -o pointer\_test pointer\_test.c，默认为64位，(char \*)和(int \*)占8个字节，而gcc -m32-o pointer\_test pointer\_test.c采用32位模式，两者占4个字节，若采用此命令出现以下问题，则需要重新下载该命令



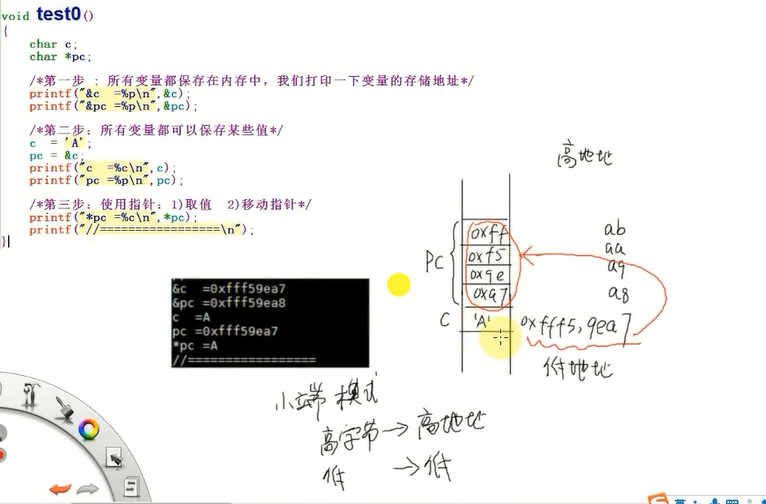
sudo apt-get update

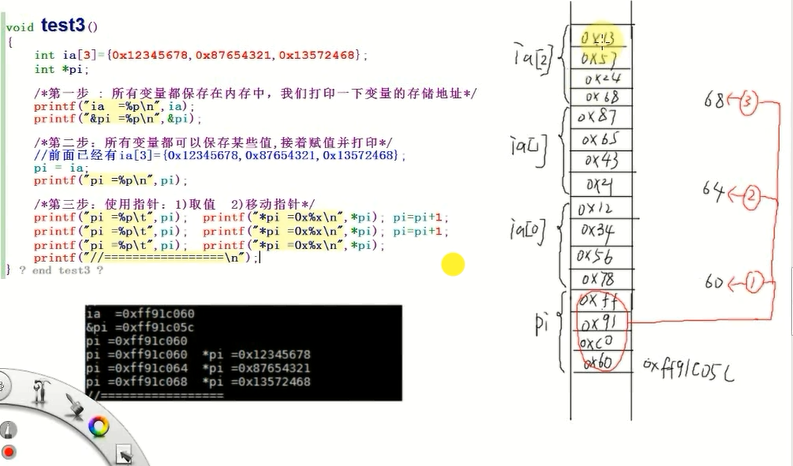
sudo apt-get purge libc6-dev

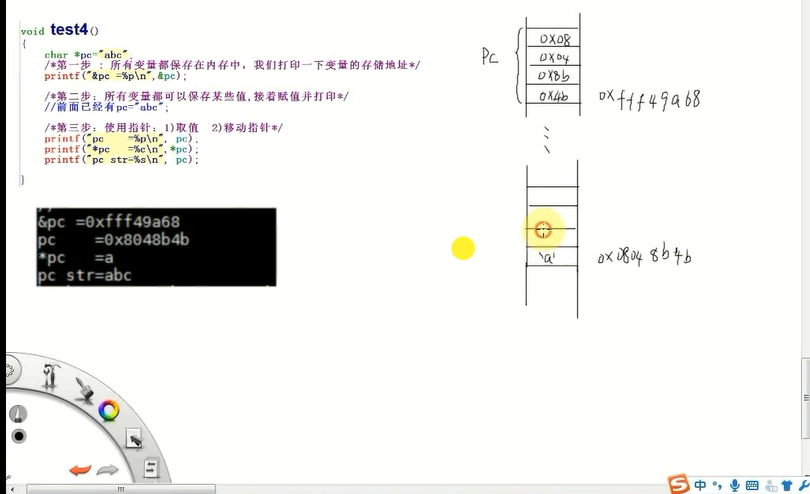
sudo apt-get install libc6-dev

sudo apt-get install libc6-dev-i386

指向char数组和int数组的指针







1. gcc和arm-linux-gcc
2. gcc

-v 查看gcc编译器版本号 显示gcc执行时的详细过程

-o<file> 指定输出文件名为file

-E 只进行预处理，生成.i文件，不进行编译、汇编、连接

-S 只对.i进行编译，不进行汇编、连接，生成.s文件

-c 编译和汇编 不连接

生成可执行文件的两种方式（预处理 编译 汇编 连接）

（1） gcc hello.c 输出一个a.out，然后./a.out来执行该应用程序。

gcc -o hello hello.c 输出hello，然后./hello来执行该应用程序。

（2） gcc -E -o hello.i hello.c 预处理🡪hello.i

gcc -S -o hello.s hello.i 编译 🡪hello.s 汇编语言

gcc -c -o hello.o hello.s 汇编 🡪hello.o（OBJ文件/ELF目标文件）机器码

gcc -o hello hello.o 连接 🡪hello （可执行程序）

（3） gcc -c -o hello.o hello.c

gcc -o hello hello.o

这是因为gcc会对.c文件默认进行预处理操作，-c再来指明了编译、汇编，从而得到.o文件，再通过gcc -o hello hello.o将.o文件进行链接，得到可执行应用程序，而链接就是将汇编生成的OBJ文件、系统库的OBJ文件、库文件链接起来，最终生成可以在特定平台运行的可执行程序。

1. gcc连接深入讲解
2. gcc -v -nostdlib -o hello hello.o会提示因为没有链接系统标准启动文件和标准库文件，而链接失败。这个-nostdlib选项常用于裸机/bootloader、linux内核等程序，因为它们不需要启动文件、标准库文件。

系统标准启动文件（crt1.o、crti.o、crtbegin.o、crtend.o、crtn.o）和标准库文件。一般应用程序才需要系统标准启动文件和标准库文件。裸机/bootloader、linux内核等程序不需要启动文件、标准库文件。

1. 动态连接与静态连接（默认为动态连接）

动态链接使用动态链接库进行链接，生成的程序在执行的时候需要加载所需的动态库才能运行。动态链接生成的程序体积较小，但是必须依赖所需的动态库，否则无法执行。

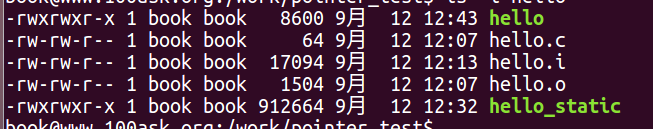
静态链接使用静态库进行链接，生成的程序包含程序运行所需要的全部库，可以直接运行，不过静态链接生成的程序体积较大。

gcc -c -o hello.o hello.c

gcc -o hello\_shared hello.o

gcc -static -o hello\_static hello.o

二者大小对比如下：

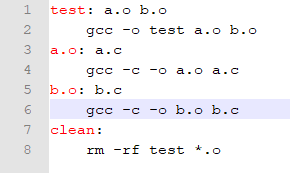


1. Makefile
2. 核心🡪规则

目标文件：依赖文件1、依赖文件2……

[TAB]命令

当目标文件不存在时或者某依赖文件比目标文件新时，则执行命令

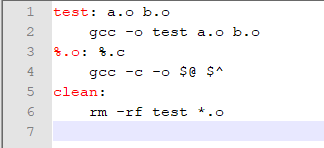


1. Makefile语法
2. 通配符 %.o %.c …….

$@ : 表示一个规则中的目标.当规则中有多个目标时，$@ 所指的是其中任何造成规则的命令运行的目标

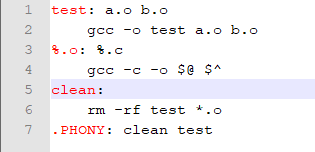
$^ : 表示规则中的所有依赖项

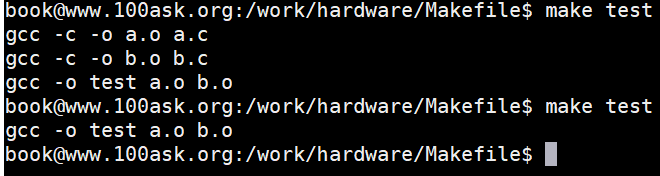
$< : 表示规则中的第一个依赖项



1. 假想目标: .PHONY

make 目标名 决定执行哪一个目标，若不写目标人，则默认执行第一个目标，如果当前目录有和执行目标同名的文件或者目录，则make会显示目标文件已更新（tip: 目标 is up to data），不会执行目标，而将目标定义为假想目标之后便不会出现此问题





1. 变量（#表示注释）

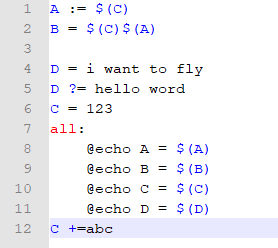
:= # 即时变量 A := xxx # A的值即刻确定，在定义时即确定

= # 延时变量 B = xxx # B的值使用到时才确定，其值为最后一次的值

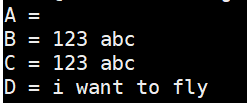
?= # 延时变量, 如果是第1次定义才起效, 如果在前面该变量已定义则 忽略这句

+= # 附加, 它是即时变量还是延时变量取决于前面的定义 相当于字符串连接，连接处以空格隔开

代码:



运行结果：



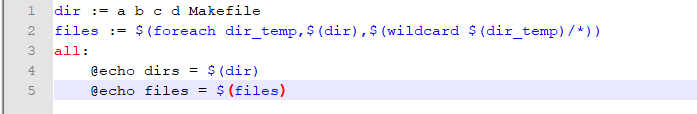
注意：echo为打印字符串命令

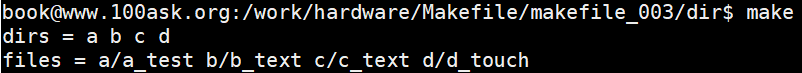
echo "It is a test" > myfile 输出字符到指定文件

1. 函数  
   a. $(foreach var,list,text) 🡪C语言中的for语句

遍历list中的值，每前进一次，将当前值赋给var，然后text引用var进行扩展（可以使用其他函数进行操作，也可以执行其他命令）,将每次扩展得到的字符串以空格隔开并相连

Eg:遍历当前目录下的a b c d目录所有包含信息 并打印出来





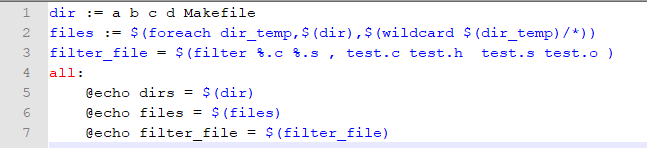
b. $(wildcard pattern)

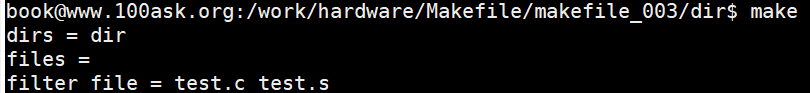
# pattern定义了文件名的格式,

# wildcard取出其中存在的文件 输出值为当前目录下的绝对路径

c. $(filter pattern...,text)

# 在text中取出符合patten格式的值





d.$(filter-out pattern...,text)

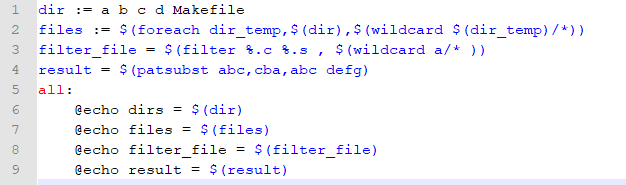
# 在text中取出不符合patten格式的值

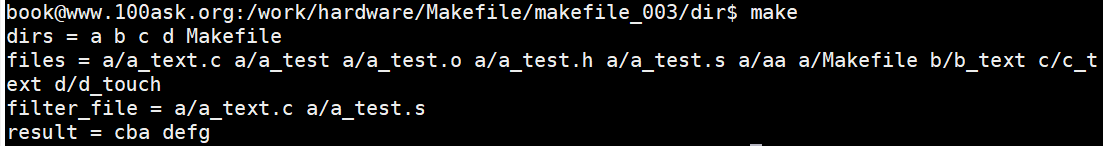
输出结果与上图相反

e. $(patsubst pattern,replacement,$(var))

# 从列表中取出每一个值 如果符合pattern则替换为replacement

#匹配模式为完全匹配





（5） Makefile实例

a. 改进: 支持头文件依赖

命令参考博客：<http://blog.csdn.net/qq1452008/article/details/50855810>

gcc -M c.c // 打印出依赖 标准库以及自定义头文件

gcc -MM c.c //打印出依赖 不包括标准库

gcc -M -MF c.d c.c // 把依赖写入文件c.d

gcc -c -o c.o c.c -MD -MF c.d // 编译c.o, 把依赖写入文件c.d

b. 添加CFLAGS 🡪 CFLAGS是用来定义编译器的编译选项

c. $\*:

$\* 表示目标模式中 '%' 及其之前的部分.如果目标是 'dir/a.foo.b'，并且目标的模式为 'a.%.b'，那么 '$\*' 的值就是 'dir/a.foo'. 如果目标中没有模式的定义，那么 '$\*' 就不能被推导出.但是，如果目标文件是 make 所识别的，那么 '$\*' 就是除了后缀的那一部分，

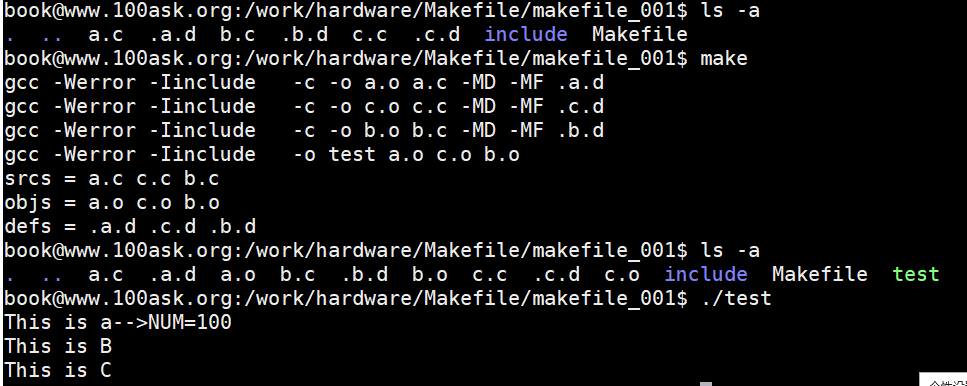
例如:

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 GNU make 的.

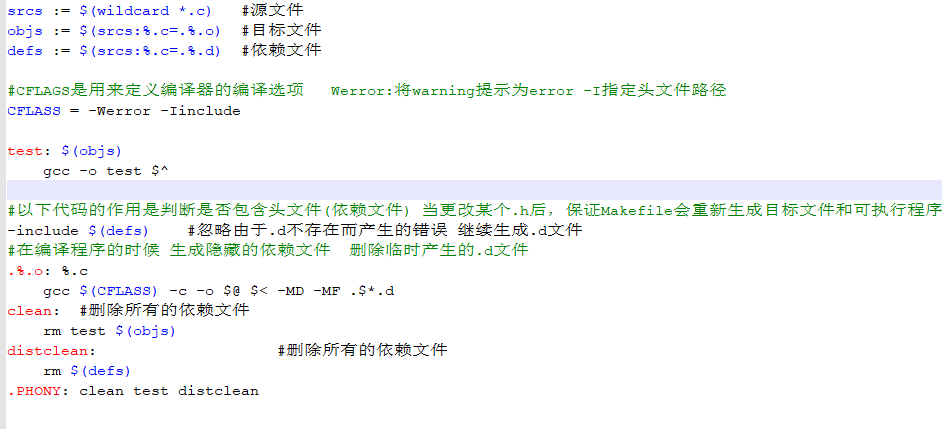
d. $(sources:.c=.d)是一个变量替换语法，把sources变量中每一项的.c替换成.d，





以上代码存在一个BUG，当.\*.d不存在时，编译中会提示这些文件不存在，导致无法编译。

以下为更改后的代码



-include：忽略因.d文件不存在导致的错误🡪 .a.d: No such file or directory

Include: 在运行到 include tmp.d 的时候，有两个动作：

检查 makefile 所在目录下是否有 tmp.d 文件；

查找 makefile 文件中是否有以 tmp.d 为目标的规则。

若有一个都会执行通过，若一个都没有就报错并中止执行。

而 -则是为了忽略错误，继续生成.d文件