1. 创建出来的文件权限与mkdir一样

#include <sys/stat.h>

int main(void) {

mkdir("/home/battery/Desktop/newdir", S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH );

}

windows是#include <direct.h>

2. popen的使用

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

const int PIPE\_BUF\_SIZE =8\*512;

int main(void) {

FILE \*rfile0 =popen("/home/battery/Desktop/echo.sh", "r");

FILE \*wfile0 =popen("cat -", "w");

if( !access("/tmp/webcamDir", F\_OK) )

mkdir("/tmp/webcamDir", S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH );

FILE \*wfile1 =popen("cat - > ~/Desktop/testOutput", "w");

char buf[PIPE\_BUF\_SIZE]; int nRealSize;

while(1) {

nRealSize=fread(buf, PIPE\_BUF\_SIZE, 1, rfile0);

fwrite(buf, nRealSize, 1, wfile0);

fwrite(buf, nRealSize, 1, wfile1);

fflush(wfile0); fflush(wfile1);

}

pclose(rfile0);

pclose(wfile0);

pclose(wfile1);

return 0;

}

#echo.sh

#!/bin/bash //这不是非要在第一行

a=1

while [ true ]

do

echo -n $a

done

3. 一个理解缓冲区的程序

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

const int PIPE\_BUF\_SIZE =8\*512;

int main(void) {

FILE \*rfile0 =popen("~/Desktop/echo.sh", "r");

char buf[PIPE\_BUF\_SIZE]; int nRealSize;

while(1) {

nRealSize=fread(buf, PIPE\_BUF\_SIZE, 1, rfile0);

printf("%d", nRealSize);

fflush(stdout); //没有这个就是隔一段时间出来一片

}

pclose(rfile0);

return 0;

}

#echo.sh

#!/bin/bash

a=1

while [ true ]

do

echo -n $a

done

4.树莓派里面的gcc居然对文件名有要求：

pi@raspberrypi:~/test $ gcc back.test

back.test: file not recognized: File format not recognized

collect2: error: ld returned 1 exit status

5.NULL在stdio.h中定义，不过C里面的定义不是很好。

#define NULL (void \*)0

6.后台进程中将stdin，stdout，stderr重定向到/dev/null的问题：

有人认为对于后台守护进程做此类重定向操作浪费资源，建议直接关闭0、1、2号句

柄拉倒，这是非常不正确的。假设它们确实被关闭了，则一些普通数据文件句柄将等

于0、1、2。以2号句柄为例，某些库函数失败后会向2号句柄输出错误信息，这将破

坏原有数据。所以都是重定向到/dev/null。还有一些没看到但是最推荐的代码是：

int fd = open( "/dev/null", O\_RDWR );

/\*

\* handle failure of open() somehow

\*/

dup2( fd, 0 );

dup2( fd, 1 );

dup2( fd, 2 );

if ( fd > 2 )

{

close( fd );

｝

1. 检测文件是否存在：使用open函数O\_EXCL | O\_CREAT 尝试打开

文件打开的几种有意思的状态：  
O\_DSYNC 等待物理I/O结束后再write。在不影响读取新写入的数据的前提下，不等待文件属性更新。

O\_RSYNC read等待所有写入同一区域的写操作完成后再进行

O\_SYNC 等待物理I/O结束后再write，包括更新文件属性的I/O

8.创建文件（包括临时）先设置UMASK 设置权限的“补码”，和chmod恰好相反。再使用tmpfile()或者mktemp()。

9.将linux 下由Clion编写的项目代码文件直接移植到VS中，很比较大的几率会出现编码错误。因为linux下虽然默认使用UTF-8格式，但是是无签名的即无BOM。创建VS空项目之后，将几个含有中文的文件，点击左上角“文件”-》“高级文件保存选项”，改为UTF-8带签名保存即可。

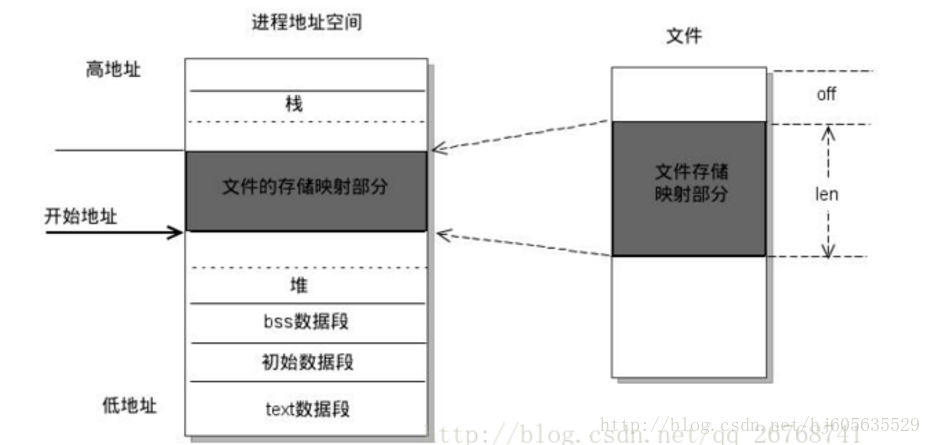
10.  <sys/mman.h>

Linux的mmap系统调用（libc封装了同名函数）可以分配一段匿名的虚拟内存区域，也可以映射一个文件到内存。

相关读写代码：<https://blog.csdn.net/hj605635529/article/details/73163513>

void\* mmap(void\* start,size\_t length,int prot,int flags,int fd,off\_t offset);

mmap() 必须以PAGE\_SIZE为单位进行映射，而内存也只能以页为单位进行[映射](https://baike.baidu.com/item/%E6%98%A0%E5%B0%84" \t "/home/vega/Documents\\x/_blank)，若要映射非PAGE\_SIZE整数倍的地址范围，要先进行内存对齐，强行以PAGE\_SIZE的倍数大小进行映射。



使用这种方式，相比较read/write方式，会不需要经过内核的缓冲区，于是进行读写操作都算是直接进行。缺点是，只能在映射范围内进行更改，不能像write一样允许文件的不断增长。很神奇直接改内存，也会改动文件对应的部分，因为是直接将内存上的值写到文件中，所以如果想用文本查看器查看需要写入的是char类型。

如果需要使用mmap实现进程间信息共享，要现在磁盘上建立一个文件。

 int munmap( void \* addr, size\_t len )  解除映射关系，解除之后，对原来映射地址的访问将导致段错误发生。

 int msync ( void \* addr , size\_t len, int flags)  一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致。

C语言GCC没有开启对C99的支持就用long double，这个倒问题不太大，但printf里L修饰符不可用。

fseek应该用在二进制模式下，在文本模式下会出现异常结果（有待深入了解，问题打包在文件夹）。win下面换行实际存了2字节，不过在C的文本模式下被视为一个\n，而fseek是对内存上地址的加减，文件内部指针却不受此影响，然后就势必会造成C的定位计算错误。

C语言中不能直接返回一个数组，可以使用结构体包装或者返回一个指针类型？

int (\*func(const int &i))[3] { //其实函数应该写成一个函数指针的形式

}

PS:下面这种类型转换，C++编译会出现报错：‘int (\*)[4]’ to ‘int\*\*’