**Q1: GCC编译过程是怎样的，如何用源代码生成可执行文件？**

**A1:**

通常所说的编译器包括一个预处理器，一个编译器和一个链接器。对于GNU Binutils/GCC 系列工具，有预处理器cpp（C Pre-Processor），编译器gcc（GNU C Compiler）和g++（GNU C++ Compiler），汇编器as（The GNU assembler）和链接器ld（The GNU Linker）。

这里有一个很经典的C程序，传说中的Hello World

#include <stdio.h>

int main() {

printf("Hello, world!\n");

}

·编译成 hello.exe gentoo@yuantoo tmp $ gcc hello.c -o hello.exe

·运行 gentoo@yuantoo tmp $ ./hello.exe

好，现在看看它到底都做了什么工作？

我用gcc hello.c -o hello.exe把hello.c编译成hello.exe。这里其实还有几个步骤，但是Gcc自动的完成了它们。这包括预处理、编译和链接。

第一步，预处理

gentoo@yuantoo tmp $ gcc -E hello.c

结果是，gcc在屏幕上飞快的打印了无数的看不见的信息。可以看到最后几行是这样的：

extern char \*ctermid (char \*\_\_s) ;

# 807 "/usr/include/stdio.h" 3 4

extern void flockfile (FILE \*\_\_stream) ;

extern int ftrylockfile (FILE \*\_\_stream) ;

extern void funlockfile (FILE \*\_\_stream) ;

# 831 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

int main() {

printf("Hello, world!\n");

}

也就是说，我们的hello.c的内容是在最后，而前面的那些东西都是从<stdio.h>以及stdio.h包含的文件中插入进来的信息。预处理器主要的工作就是处理所有源码中以#开头的行，将#include 指令替换成指令指出的文件的内容，对#define 定义的符号进行了文本替换，以及根据符号选择需要进入结果文件的内容。我们短短四行字的代码文件，经过 gcc 的预处理，得到了一个 913 行的文件。

预处理之后的工作是汇编--这也是真正编译工作的第一步骤。

gentoo@yuantoo tmp $ as hello.s -o hello.o

得到了hello.o，就是包含hello.c代码的对象代码。得到的hello还不能运行，我们需要把它和C语言运行库链接起来。它不仅包含了程序的入口，还有 printf 等标准 C 库函数的实现。

gentoo@yuantoo tmp $ ld -o a.out -dynamic-linker

/lib/ld-linux.so.2 /usr/lib/crt?.o hello.o -lc

这里-o hello.exe表示输出为hello.exe文件。

-dynamic-linker /lib/ld-linux.so.2表示将程序动态链接到/lib/ld-linux.so.2这个shared object。这个shared object作为操作系统中的一个特殊的库，它负责引入其他的so。同时，我们要在我们的程序中包含/usr/lib下的几个crt?.o对象文件：它们中包含了C程序的所需要的运行时环境。最后一个参数-lc表示将程序链接到标准C库上（名为libc.so.5，在/usr/lib 目录下。ld 会根据配置文件自动搜索/usr/lib目录；如果库文件在其他目录中，则需要用-L参数指出。）执行完毕，我们得到了一个具有执行权限的hello.exe 文件。

就这样，一个简单的C程序文件，经过0、编写，1、预处理，2、编译，3、链接，终于生成了一个可执行文件。一般而言，gcc编译器可以替我们完成整个过程，只要简简单单一个 gcc -o hello.exe hello.c 命令，三个步骤就可以统统完成。

从上面可以看出，编写代码和编译代码完全是分离的两个过程，可以用完全不同的工具替换每个步骤（譬如用 notepad 或者 EditPlus 作编辑器编写代码，用Cygwin 或者 Mingw 做编译器，等等）。VC 和 TC 都是所谓的IDE（集成开发环境），它包含了编辑器、编译器和调试器，通常还包含了项目管理和文档生成以及其他一系列辅助工具，可以大大简化项目开发周期。它们既不是编译-器，也不是编辑器；它们包含了这些所有东西。

**Q2: 使用链接器进行链接的时候，是将被link的.so文件中的二进制内容全部link到最终的exe文件中，还是只link被调用到的子过程？对于Windows平台上链接.obj，lib，又是怎样？一样的吗？**

**A3:**

先说说unix/linux下的情况。在静态连接时，可执行文件会找到相应用到的.o文件代码连接进去。不是只连接用到的函数。另外，.a文件不大一样。因-为.a是一堆.o打包而成，所以连接的时候只会把.a中用到的.o连接进去。动态连接就完全不同，只记录了需要的动态库名字，一点点代码都不会连接进去。

而且对于动态库来说,里面的符号表是没有用的(除非为了调试),可以通过 strip命令减少动态库的大小.而静态库里面的符号表则不能去除.因为其实静态库只是通过ar命令把.o对象文件打个包而已。对于hpux来说，所有静态库里面的静态全局对象不会自动构造的。如果想要让静态全局对象自动构造的话，则需要把这些.o创建成动态库。

2，windows下也类似，简单的说，静态连接就是把用到的库和你的程序完全"打包"在一起，生成的程序可以独立运行，对静态库不再有依赖，缺点是生成的文件-较大，当库更新时需要重新编译连接。

动态连接只是在自己的可执行程序里面记录了动态库中的符号信息，动态库和可执行程序是独立的，可执行程序脱离了动态库就无法运行，好处是可执行程序体积小，动态-库可以供多个程序使用，如果动态库的接口没有改变，当其升级后，可执行程序无需改变。

3、一般来说，当link一些目标文件和库文件的时候，link程序对目标文件和库文件的处理是不同的，它会把你要求的目标文件全部链接进去，但库文件中只被目标文件引用的符号所在的模块。所以把常用的目标文件加入到一个库中是比较好的。

对于库文件中的选择性链接是基于模块的，一般来说加入到库中时的一个目标文件就是一个模块。如果你的一个模块中有三个函数，其中一个被调用了，这三个函数的代码-都会出现在最终的可执行文件中。因此，当我们制作库文件时，最好让目标文件足够小。

**Q3: 用 gcc 编译成静态库得到的.a文件和用Microsoft cl编译得到的lib文件是否相同？**

**A3:**

.a 文件的来历是：用 gcc 编译C/C++源码得到对象文件（扩展名.o）之后，用ar(1)命令打包得到的。可以用ar命令对.a文件里包含的各个对象文件进行查询、添加、删除等操作。

Microsoft cl 编译的lib通常被认为是一张符号表，链接器通过读取这个符号表，查询各个函数的实现的位置。lib可以与对象文件搭配，起到静态链接的效果，也可以作为 dll 的引入表，自动完成动态链接时候的繁琐步骤。

因此，lib和.a文件实际上是有差别的。

**Q4：GCC编译选项说明？**

**A4：**

-c 只激活预处理,编译,和汇编,也就是他只把程序做成obj文件

-o 制定目标名称,缺省的时候,gcc 编译出来的文件是a.out

-D 允许从编译程序命令行定义宏符号

一共有两种情况:一种是用-DMACRO,相当于在程序中使用#define MACRO,另一种是用-DMACRO=A,相当于程序中的#define MACRO A.

eg：如对下面这段代码:

编译时可加上-DDEBUG参数,执行程序则打印出编译信息

#ifdefine DEBUG

printf("debug message\n");

#endif

-Umacro 　　相当于C语言中的#undef macro

-undef 　　取消对任何非标准宏的定义

-g

指明编译程序在编译的输出中应产生调试信息.这个调试信息使源代码和变量名引用在调试程序中或者当程序异常退出后在分析core文件时可被使用.

-Wall 以最高级别使用GNU编译程序,专门用于显示警告用!!

-I dir 　　在你是用#include"file"的时候,gcc/g++会先在当前目录查找你所制定的头文件,如果没有找到,他回到缺省的头文件目录找。

-i 　就是取消前一个参数的功能,所以一般在-I dir之后使用。

-idirafter dir 　　在-I的目录里面查找失败,到这个目录里面查找

-iprefix prefix -iwithprefix dir

一般一起使用,当-I的目录查找失败,会到prefix+dir 下查找

-nostdin C++ 规定不在g++指定的标准路经中搜索,但仍在其他路径中搜索.

-S 只激活预处理和编译，就是指把文件编译成为汇编代码。生成.s的汇编代码，你可以用文本编辑器查看。

-E 只激活预处理,这个不生成文件,你需要把它重定向到一个输出文件里面。

-C 在预处理的时候,不删除注释信息,一般和-E使用,有时候分析程序，用这个很方便的。

-M 生成文件关联的信息。包含目标文件所依赖的所有源代码你可以用gcc -M hello.c 来测试一下，很简单。

-MM 和上面的那个一样，但是它将忽略由#include造成的依赖关系

-Ldir 指定连接库的搜索目录，制定编译的时候，搜索库的路径。比如你自己的库，可以用它制定目录，不然编译器将只在标准库的目录找。这个dir就是目录的名称。

-llibrary 指定连接库的名字

**Q4：extern “C”和extern的作用分别是什么？**

**A4：**

由于C++有重载的特性，C++会根据函数名参数的个数更改汇编以后的函数名，这就导致了C++和C语言汇编生成函数名的规则是不同的（C++相对复杂一些）。为了兼容C++使用C语言函数库，需要在CPP文件中说明使用的是C函数库的函数。extern 和前者没有什么关系，只是外部变量的声明。

**Q5：C语言中常用缓冲区溢出有哪些？**

**A5：**

C中大多数缓冲区溢出问题可以直接追溯到标准 C 库。最有害的罪魁祸首是不进行自变量检查的、有问题的字符串操作（strcpy、strcat、sprintf 和 gets）。一般来讲，象“避免使用 strcpy()”和“永远不使用 gets()”这样严格的规则接近于这个要求。

今天，编写的程序仍然利用这些调用，因为从来没有人教开发人员避免使用它们。某些人从各处获得某个提示，但即使是优秀的开发人员也会被这弄糟。他们也许在危险函数的自变量上使用自己总结编写的检查，或者错误地推论出使用潜在危险的函数在某些特殊情况下是“安全”的。

第一位公共敌人是 gets()。gets() 根本不执行边界检查。为一个替代方法，可以使用方法 fgets()，它可以做与 gets() 所做的同样的事情，但它接受用来限制读入字符数目的大小参数。

|  |
| --- |
| void main()  {  char buf[1024];  gets(buf);  } |

而使用以下代码：

|  |  |  |
| --- | --- | --- |
| #define BUFSIZE 1024  void main()  {  char buf[BUFSIZE];  fgets(buf, BUFSIZE, stdin);  }  大多数都是由于没有限制输入大小导致的缓冲区溢出，下面是一个小总结。 | | |
| **函数** | | **严重性** | **解决方案** | |
| gets | | 最危险 | 使用 fgets（buf, size, stdin）。这几乎总是一个大问题！ | |
| strcpy | | 很危险 | 改为使用 strncpy。 | |
| strcat | | 很危险 | 改为使用 strncat。 | |
| sprintf | | 很危险 | 改为使用 snprintf，或者使用精度说明符。 | |
| scanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| sscanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| fscanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| vfscanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| vsprintf | | 很危险 | 改为使用 vsnprintf，或者使用精度说明符。 | |
| vscanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| vsscanf | | 很危险 | 使用精度说明符，或自己进行解析。 | |
| streadd | | 很危险 | 确保分配的目的地参数大小是源参数大小的四倍。 | |
| strecpy | | 很危险 | 确保分配的目的地参数大小是源参数大小的四倍。 | |
| strtrns | | 危险 | 手工检查来查看目的地大小是否至少与源字符串相等。 | |
| realpath | | 很危险（或稍小，取决于实现） | 分配缓冲区大小为 MAXPATHLEN。同样，手工检查参数以确保输入参数不超过 MAXPATHLEN。 | |
| syslog | | 很危险（或稍小，取决于实现） | 在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。 | |
| getopt | | 很危险（或稍小，取决于实现） | 在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。 | |
| getopt\_long | | 很危险（或稍小，取决于实现） | 在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。 | |
| getpass | | 很危险（或稍小，取决于实现） | 在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。 | |
| getchar | | 中等危险 | 如果在循环中使用该函数，确保检查缓冲区边界。 | |
| fgetc | | 中等危险 | 如果在循环中使用该函数，确保检查缓冲区边界。 | |
| getc | | 中等危险 | 如果在循环中使用该函数，确保检查缓冲区边界。 | |
| read | | 中等危险 | 如果在循环中使用该函数，确保检查缓冲区边界。 | |
| bcopy | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| fgets | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| memcpy | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| snprintf | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| strccpy | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| strcadd | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| strncpy | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |
| vsnprintf | | 低危险 | 确保缓冲区大小与它所说的一样大。 | |

**Q6：C语言中栈和堆比较？**

**A6：**

C语言程序的内存分配方式

1. **内存分配方式**

内存分配方式有三种：  
　　[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中，效率很高，但是分配的内存容量有限。  
　　[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，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。  
**2.程序的内存空间**  
　　一个程序将操作系统分配给其运行的内存块分为4个区域，如下图所示。  
　　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 ，存放为运行函数而分配的局部变量、函数参数、返回数据、返回地址等。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 。分配方式类似于链表。  
　　3、全局区（静态区）（static）—存放全局变量、静态数据、常量。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（类成员函数和全局函数）的二进制代码。

下面给出例子程序，  
int a = 0; //全局初始化区  
char \*p1; //全局未初始化区  
int main()

{  
 int b; //栈  
　　char s[] = "abc"; //栈  
　　char \*p2; //栈  
　　char \*p3 = "123456"; //123456在常量区，p3在栈上。  
　　static int c =0;//全局（静态）初始化区  
　　p1 = new char[10];  
　　p2 = new char[20];  
　　//分配得来得和字节的区域就在堆区。  
　　strcpy(p1, "123456"); //123456放在常量区，编译器可能会将它与p3所指向的 "123456"优化成一个地方。  
}  
**3．堆与栈的比较**  
　　3.1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，C++中是new运算符。  
　　如p1 = (char \*)malloc(10); p1 = new char[10];  
　　如p2 = (char \*)malloc(10); p2 = new char[20];  
　　但是注意p1、p2本身是在栈中的。  
　　3.2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。  
　　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。  
　　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。  
　　3.3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。  
　　3.4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VirtualAlloc分配内存，他不是在堆，也不是栈，而是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。  
　　3.5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。堆中的具体内容有程序员安排。  
　　3.6存取效率的比较  
　　char s1[] = "a";  
　　char \*s2 = "b";  
　　a是在运行时刻赋值的；而b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：  
　　int　main(){  
　　char a = 1;  
　　char c[] = "1234567890";  
　　char \*p ="1234567890";  
　　a = c[1];  
　　a = p[1];  
　　return 0;  
　　}  
　　对应的汇编代码  
　　10: a = c[1];  
　　00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]  
　　0040106A 88 4D FC mov byte ptr [ebp-4],cl  
　　11: a = p[1];  
　　0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]  
　　00401070 8A 42 01 mov al,byte ptr [edx+1]  
　　00401073 88 45 FC mov byte ptr [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，再根据edx读取字符，显然慢了。

3.7小结  
　　堆和栈的主要区别由以下几点：  
　　1、管理方式不同；  
　　2、空间大小不同；  
　　3、能否产生碎片不同；  
　　4、生长方向不同；  
　　5、分配方式不同；  
　　6、分配效率不同；  
　　管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。  
　　空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M。当然，这个值可以修改。  
　　碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构。  
　　生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。  
　　分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由malloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。  
　　分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。  
　　从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址， EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。  
　　虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。  
　　无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果。  
**4.new/delete与malloc/free比较**  
　　从C++角度上说，使用new分配堆空间可以调用类的构造函数，而malloc()函数仅仅是一个函数调用，它不会调用构造函数，它所接受的参数是一个unsigned long类型。同样，delete在释放堆空间之前会调用析构函数，而free函数则不会。  
class Time{  
　　public:  
　　Time(int,int,int,string);  
　　~Time(){  
　　cout<<"call Time’s destructor by:"<<name<<endl;  
　　}  
　　private:  
　　int hour;  
　　int min;  
　　int sec;  
　　string name;  
　　};  
　　Time::Time(int h,int m,int s,string n){  
　　hour=h;  
　　min=m;  
　　sec=s;  
　　name=n;  
　　cout<<"call Time’s constructor by:"<<name<<endl;  
}  
int main(){  
　　Time \*t1;  
　　t1=(Time\*)malloc(sizeof(Time));  
　　free(t1);  
　　Time \*t2;  
　　t2=new Time(0,0,0,"t2");  
　　delete t2;  
　　system("PAUSE");  
　　return EXIT\_SUCCESS;  
}  
　　结果：  
　　call Time’s constructor by:t2  
　　call Time’s destructor by:t2  
　　从结果可以看出，使用new/delete可以调用对象的构造函数与析构函数，并且示例中调用的是一个非默认构造函数。但在堆上分配对象数组时，只能调用默认构造函数，不能调用其他任何构造函数。

**Q7：FILE指针和文件描述符的关系如何？**

**A7：**

系统IO调用有 creat(),open(),read(),write(),close(),ioctl(),接受一个文件描述符，是一个整数，用于索引开放文件的每个进程表(per-process table -of -open -file)

为了确保程序的可移植性应该使用标准IO库调用，如fopen(0,fclose(),fputc(),fseek()等，它们绝大多数中的名字中带有一个"f"。这些调用都接受一个类型为FILE结构的指针(有时称为流指针)的参数。FILE指针指向一个流结构，结构的内容根据编译器的不同而有所不同，在UNIX中通常是Open File的每个进程表的一个条目。在典型情况下，它包含了流缓冲区、所有用于提示缓冲区有多少字节是实际的文件数据的变量以及提示流状态的标志(如ERROR和EOF)等

所以，文件描述符就是Open File中的每个进程表的一个偏移量(如"3")。它用于UNIX系统中，用于标识文件。

**Q8：各个平台下各种数据类型所占内存大小是怎样的？**

**A8：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 \ 平台 | Window32 | Window64 | Linux32 | Linux64 |
| Int | 4 | 4 | 4 | 4 |
| Short | 2 | 2 | 2 | 2 |
| Unsigned | 4 | 4 | 4 | 4 |
| Long | 4 | 4 | 4 | 8 |
| Float | 4 | 4 | 4 | 4 |
| Double | 8 | 8 | 8 | 8 |
| Void \* | 4 | 8 | 4 | 8 |
| Char | 1 | 1 | 1 | 1 |
| Size\_t | 4 | 8 | 4 | 8 |
| Long long | 8 | 8 | 8 | 8 |
| Long double | 8 | 8 | 12 | 16 |