面试题总结

助教：朱海蕊

祝您找到心仪的工作

目录

第一部分：基本概念及其它问答题

0、进程线程的区别？

进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他?线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正?相反。同时，线程适合于在 SMP 机器上运行，而进程则可以跨机器迁移

1、关键字 static 的作用是什么？

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？

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(构造函数)

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、static 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";//栈
char *s2="bbbbbbbbbbbbbbbbb";//常量空间
                                    //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 void main() {
    char a=1;
    char c[]="1234567890";
    char *p="1234567890";
    a = c[1];
    a = p[1];
    return;
}
```

对应的汇编代码

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。(反汇编查看汇编的代码)

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). 下面的求平方的函数有什么错误：

```
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。Const 只是对程序而言的。

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。（数据结构中的内容）

25、结构与联合有何区别？

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

	含参数的宏	函数
处理过程		
编译时	不分配内存	分配内存
程序运行时	没有参数类型问题	定义实参、形参类型
处理时间		
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)如果栈 B 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

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;
```

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

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？

答案是：typedef 更好，typedef 会做安全性检查。宏定义只是简单的字符串代换(原地扩展)，而 typedef 则不是原地扩展，它的新名字具有一定的封装性，以致于新命名的标识符具有更易定义变量的功能。请看如下例子：

typedef (int*) pINT;

以及下面这行:

#define pINT2 int*

效果相同？实则不同！实践中见差别：

pINT a,b;的效果同 int *a; int *b;表示定义了两个整型指针变量。

而 pINT2 a,b;的效果同 int *a, b;表示定义了一个整型指针变量 a 和整型变量 b。

39、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 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。

44、sizeof 和 strlen 的区别

1.sizeof 是一个操作符，strlen 是库函数。
2.sizeof 的参数可以是数据的类型，也可以是变量，而 strlen 只能以结尾为‘\0‘的字符串作参数。
3.编译器在编译时就计算出了 sizeof 的结果。而 strlen 函数必须在运行时才能计算出来。并且 sizeof
4.计算的是数据类型占内存的大小，而 strlen 计算的是字符串实际的长度。
5.数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针了。
注意：有些是操作符看起来像是函数，而有些函数名看起来又像操作符，这类容易混淆的名称一定要加以区分，否则遇到数组名这类特殊数据类型作参数时就很容易出错。最容易混淆为函数的操作符就是 sizeof。

45、C 中的 malloc 和 C++中的 new 有什么区别

malloc 和 new 有以下不同：
1.new、delete 是操作符，可以重载，只能在 C++中使用。
2.malloc、free 是函数，可以覆盖，C、C++中都可以使用。
3.new 可以调用对象的构造函数，对应的 delete 调用相应的析构函数。
4.malloc 仅仅分配内存，free 仅仅回收内存，并不执行构造和析构函数
5.new、delete 返回的是某种数据类型指针，malloc、free 返回的是 void 指针。

46、简述 C、C++程序编译的内存分配情况

C、C++中内存分配方式可以分为三种：

（1）从静态存储区域分配：

内存在程序编译时就已经分配好，这块内存在程序的整个运行期间都存在。速度快、不容易出错，因为有系统会善后。例如全局变量，static 变量等。

（2）在栈上分配：

在执行函数时，函数内局部变量的存储单元都在栈上创建，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：

即动态内存分配。程序在运行的时候用 malloc 或 new 申请任意大小的内存，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。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，另外频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

一个 C、C++程序编译时内存分为 5 大存储区：堆区、栈区、全局区、文字常量区、程序代码区。

47、简述 strcpy、sprintf 与 memcpy 的区别

三者主要有以下不同之处：

1.操作对象不同，strcpy 的两个操作对象均为字符串，sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串，memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

2.执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。

3.实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字符串的转化，memcpy 主要是内存块间的拷贝。

48、链表和数组有什么区别

数组和链表有以下几点不同：

（1）存储形式：数组是一块连续的空间，声明时就要确定长度。链表是一块可不连续的动态空间，长度可变，每个结点要保存相邻结点指针。

（2）数据查找：数组的线性查找速度快，查找操作直接使用偏移地址。链表需要按顺序检索结点，效率低。

（3）数据插入或删除：链表可以快速插入和删除结点，而数组则可能需要大量数据移动。

（4）越界问题：链表不存在越界问题，数组有越界问题。

说明：在选择数组或链表数据结构时，一定要根据实际需要进行选择。数组便于查询，链表便于插入删除。数组节省空间但是长度固定，链表虽然变长但是占了更多的存储空间。

49、谈谈你对编程规范的理解或认识

编程规范可总结为：程序的可行性，可读性、可移植性以及可测试性。
说明：这是编程规范的总纲目，面试者不一定要去背诵上面给出的那几个例子，应该去理解这几个例子说明的问题，想一想，自己如何解决可行性、可读性、可移植性以及可测试性这几个问题，结合以上几个例子和自己平时的编程习惯来回答这个问题。

50、&&和&、||和|有什么区别

（1）&和|对操作数进行求值运算，&&和||只是判断逻辑关系。
（2）&&和||在在判断左侧操作数就能确定结果的情况下就不再对右侧操作数求值。
注意：在编程的时候有些时候将&&或||替换成&或|没有出错，但是其逻辑是错误的，可能会导致不可预想的后果（比如当两个操作数一个是 1 另一个是 2 时）

51、Linux 系统调用及用户编程接口（API）、系统命令的区别

系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。
例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

为什么用户程序不能直接访问系统内核提供的服务呢？

- 这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。

- 用户进程在通常情况下不允许访问内核数据，也无法直接调用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

- 当用户空间的进程需要获得一定的系统服务时，应用程序调用系统调用，这时操作系统就根据系统调用号（每个系统调用被赋予一个系统调用号）使用户进程进入内核空间的具体位置调用相应的内核代码。

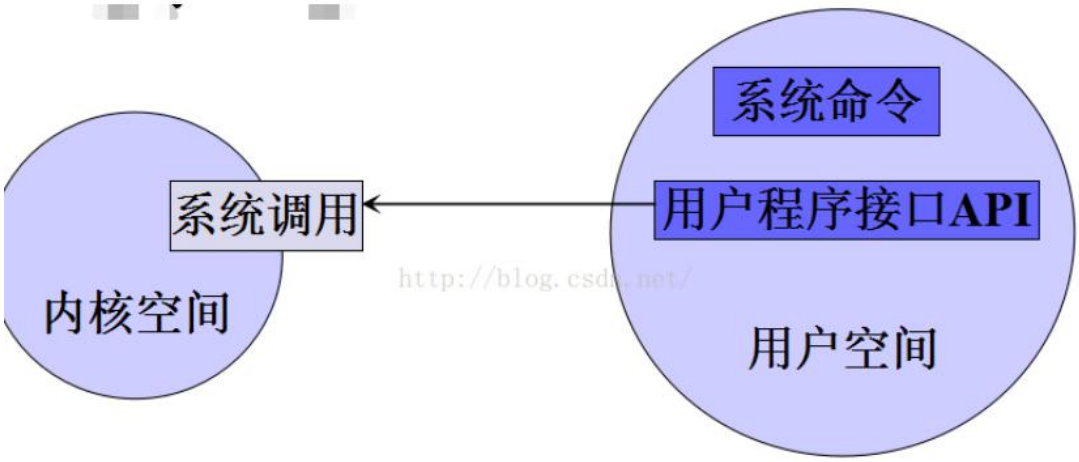
- 进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。
  系统调用按照功能逻辑大致可分为

- 进程控制
- 进程间通信
- 文件系统控制
- 系统控制
- 存储管理

- 网络管理
- socket 控制
- 用户管理等

系统调用并不是直接与程序员进行交互的，它仅仅是一个通过软中断机制向内核提交请求，以获取内核服务的接口。在实际使用中程序员调用的通常是：**用户编程接口——API** ，API 遵循了 UNIX 中最流行的应用编程界面标—POSIX 编程标准。

系统命令相对 API 更高了一层，它实际上是一个可执行程序，它的**内部引用了一个或多个用户编程接口（API）来实现相应的功能。**



系统调用、API与系统命令之间的关系

52、Linux 中的交叉编译

1.在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码，我们就称这种编译器支持交叉编译。这个编译过程就叫交叉编译。
2.简单地说，就是在一个平台上生成另一个平台上的可执行代码。
3.这里需要注意的是所谓平台，实际上包含两个概念：体系结构（Architecture）、操作系统（Operating System）。
4.同一个体系结构可以运行不同的操作系统；
5.同样，同一个操作系统也可以在不同的体系结构上运行。

53、Linux 中文件及文件描述符概述

在 Linux 中，一切皆文件！
所有对设备和文件的操作都是使用文件描述符来进行的。
文件描述符是一个非负的整数，它是一个索引值，并指向在内核中每个进程所打开的文件的记录表；

当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；
当需要读写文件时，需要把文件描述符作为参数传递给相应的函数。
通常，一个进程启动时，都会打开 3 个文件：
标准输入：对应文件描述符为 0（STDIN_FILENO）
标准输出：对应文件描述符为 1（STDOUT_FILENO）
标准出错处理：对应文件描述符为 2（STDERR_FILENO）

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)
{
    FUNC1 p=&inc;
    int temp =p(arg1);
    fun(&temp,&arg1, arg2);
    printf("%d\n",*arg2);
}
int 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”

```
#include"string.h"
main()
{
    char*src="hello,world";
    char* dest=NULL;
    int len=strlen(src);
    dest=(char*)malloc(len);
    char* d=dest;
    char* s=src[len];
    while(len--!=0)
    d++=s--;
```

```
    printf("%s",dest);
    return 0;
}
```

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];
     tr[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 效率问题

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（记忆方法：从中间分开，const 和谁在一起，谁就是常量）

```
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;  //指向常量的指针，指向的常量值不可以改
```

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;
```

//虽然 d 指向的内容是只读的，但是改变 d，d 指向的内容随之改变　*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,

```
(1)          a = a + b;
             b = a - b;
             a = a - b;
(2)    a = a^b;
       b = a^b;
       a = a^b;
(3)    b ^= 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));//找到 0 为止
}
```

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、补齐与对齐

```
struct A  {
        char t:4;//1
        char k:4;//1
        unsigned short i:8;//2
        unsigned long m;//4
};
```

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、对齐与补齐

```
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。字符串没有结尾。
}
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，strlen 不含\0
{
  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(src);
    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"
      Int 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); }
```

46、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 *。

```
{
    {
        {
            //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;

                    }
                }
            }
    }
```

- [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

答：
```
#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。

答：
```
#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）　//快速排序

```
  #include<iostream>
usingnamespace std;
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;
    Return low;
 }
Void QSort (int*L,intlow,int high) {
    if (low < high) {
         intpl = Partition (L,low,high);
        QSort (L,low,pl - 1);
        QSort (L,pl + 1,high);
    }
}
Int main () {
    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;
    Int aa = 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 {
I   nt 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; }
  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、请简述以下两个 for 循环的优缺点 ？

```
{{{
for (i=0; i<N; i++)
{
if (condition)
    DoSomething();
else
    DoOtherthing();
}   if (condition)
{
for (i=0; i<N; i++)
    DoSomething();
}
else
{
    for (i=0; i<N; i++)
    DoOtherthing();
}

}}}
```

- 答：
- 优点：程序简洁
- 缺点：多执行了 N-1 次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。　　优点：循环的效率高,程序不简洁

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; }
```

15、实现 strcmp

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

    return 1;
        Else if (*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)

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){
    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:

```
void* memcpy( void *dst, const void *src, unsigned int len )
271 {
272    register char *d;
273    register char *s;
274
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、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;

}
```

25、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; }
```

26、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"

```
#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);
    }
```

27、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 );
}
```

28、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;
}
```

29、有双向循环链表结点定义为：(提高题)

```
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;
        }
    }
}
```

30、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" （提高题）

```
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;
                        }
                    }
            }
 }
```

31、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、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？
 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;
}
```

33、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; }
```

34、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。 （提高题）

数组实现：

```
#include <stdio.h>
#include <malloc.h>
int Josephu(int n, int m) {
I   nt 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; }
```

35、strcpy 函数：

```
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)) //[/1]
Printf ( "Invalid argument(s)") ; //[2]
char * strDestCopy=strDest;  //[/3]
while ((*strDest++=*strSrc++)!='\0'); //[/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。

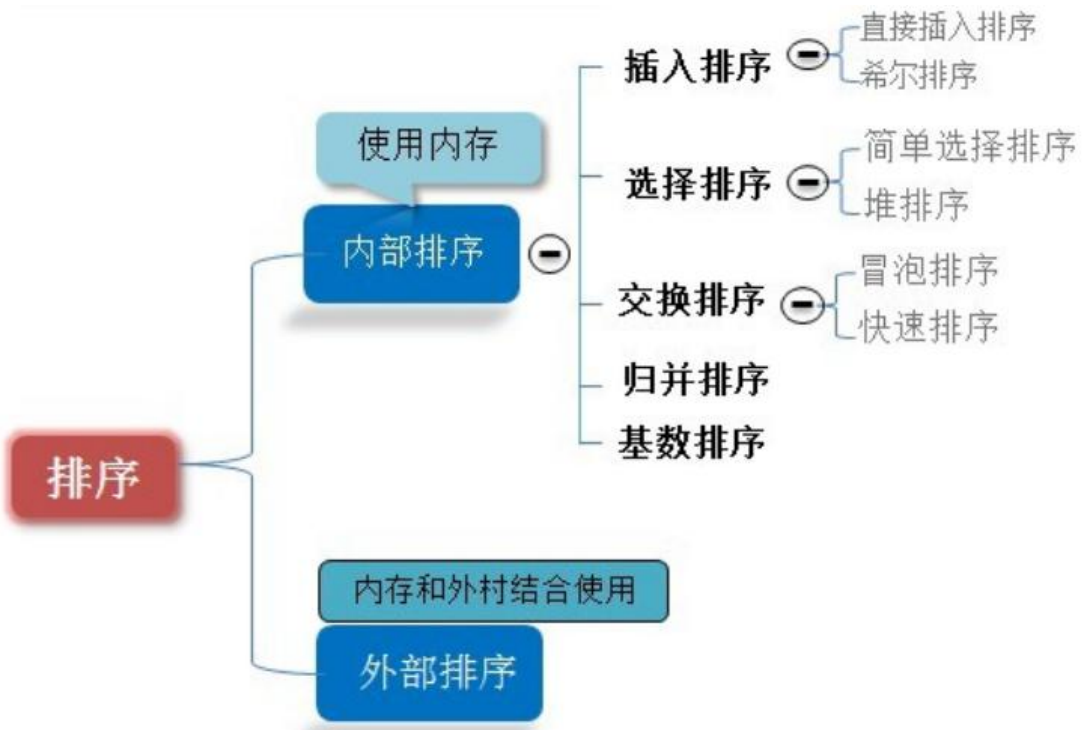
二、位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名•位域名 位域允许用各种格式输出。

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; }

第四部分 八大排序算法

概述：排序有内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。

我们这里说说八大排序就是内部排序



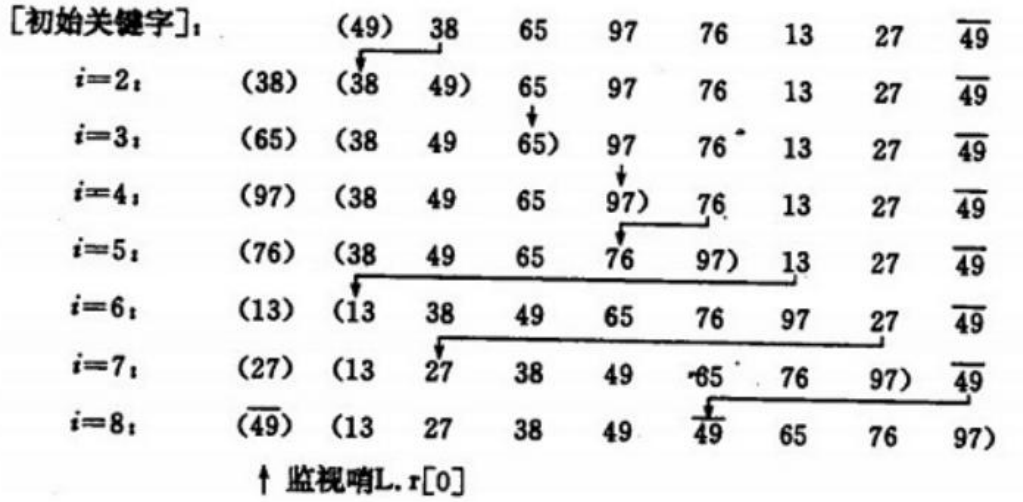
1.插入排序—直接插入排序

基本思想:

将一个记录插入到已排序好的有序表中，从而得到一个新，记录数增 1 的有序表。即：先将序列的第 1 个记录看成是一个有序的子序列，然后从第 2 个记录逐个进行插入，直至整个序列有序为止。

要点：设立哨兵，作为临时存储和判断数组边界之用。

直接插入排序示例：



如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，**所以插入排序是稳定的。**

算法的实现：

```
1.  void print(int a[], int n ,int i){
2.      cout<<i <<":";
3.      for(int j= 0; j<8; j++){
4.          cout<<a[j] <<" ";
5.      }
6.      cout<<endl;
7.  }
8.
9.
10. void InsertSort(int a[], int n)
11. {
12.     for(int i= 1; i<n; i++){
13.         if(a[i] < a[i-1]){              //若第 i 个元素大于 i-1 元素，直接插入。小于的话，移
    动有序表后插入
14.             int j= i-1;
15.             int x = a[i];       //复制为哨兵，即存储待排序元素
16.             a[i] = a[i-1];          //先后移一个元素
17.             while(x < a[j]){  //查找在有序表的插入位置
18.                 a[j+1] = a[j];
19.                 j--;        //元素后移
20.             }
21.             a[j+1] = x;     //插入到正确位置
22.         }
23.         print(a,n,i);           //打印每趟排序的结果
24.     }
25.
26. }
27.
28. int main(){
29.     int a[8] = {3,1,5,7,2,4,9,6};
30.     InsertSort(a,8);
31.     print(a,8,8);
32. }
```

效率：

时间复杂度：O（n^2）.

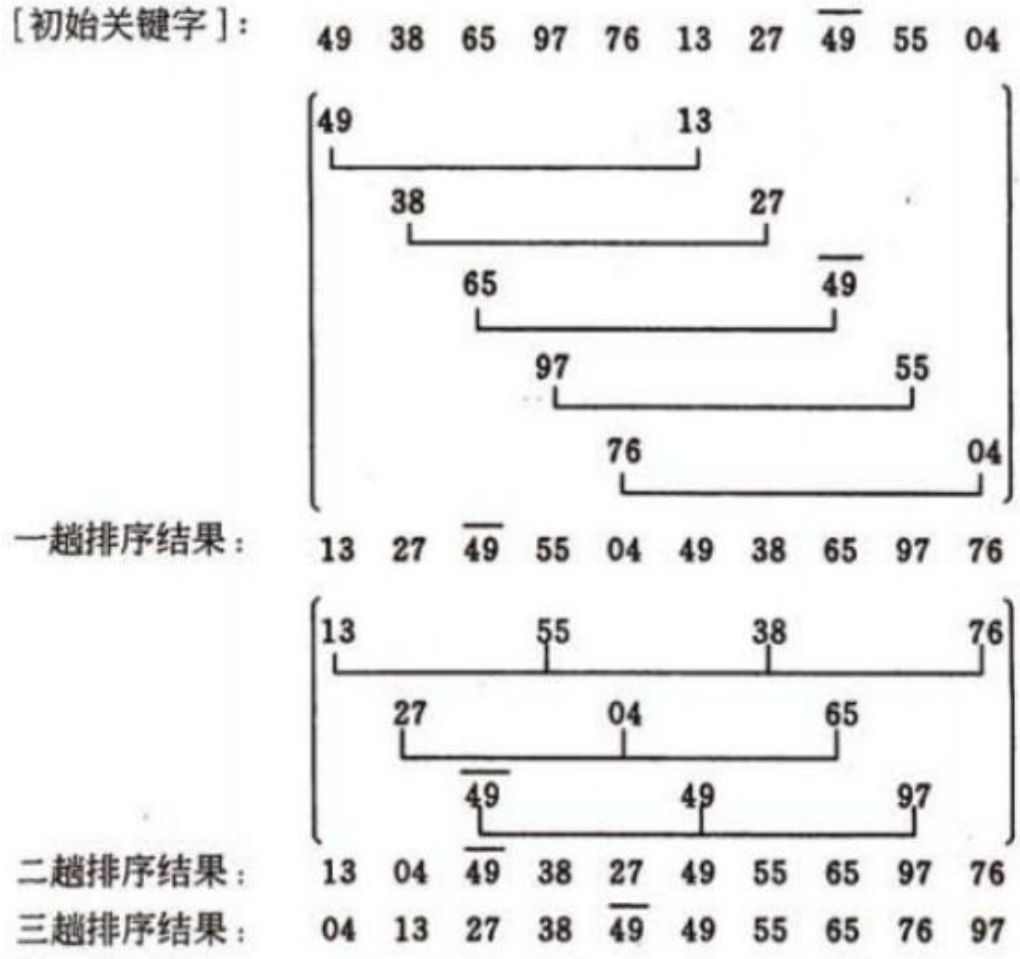
2. 插入排序—希尔排序

基本思想：

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

操作方法：

1,、选择一个增量序列 t1，t2，…，tk，其中 ti>tj，tk=1；
2、按增量序列个数 k，对序列进行 k 趟排序；
3、每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。
希尔排序的示例：



算法实现：

我们简单处理增量序列：增量序列 d = {n/2 ,n/4, n/8 .....1} n 为要排序数的个数
即：先将要排序的一组记录按某个增量 d（n/2,n 为要排序数的个数）分成若干组子序列，每组中记录的下标相差 d.对每组中全部元素进行直接插入排序，然后再用一个较小的增量（d/2）对它进行分组，在每组中再进行直接插入排序。继续不断缩小增量直至为 1，最后使用直接插入排序完成排序。

```
1.  void print(int a[], int n ,int i){
2.      cout<<i <<":";
3.      for(int j= 0; j<8; j++){
```

```
4.          cout<<a[j] <<" ";
5.      }
6.      cout<<endl;
7.  }
8.  /**
9.   * 直接插入排序的一般形式
10. *
11. * @param int dk 缩小增量，如果是直接插入排序，dk=1
12. *
13. */
14.
15. void ShellInsertSort(int a[], int n, int dk)
16. {
17.     for(int i= dk; i<n; ++i){
18.         if(a[i] < a[i-dk]){         //若第 i 个元素大于 i-1 元素，直接插入。小于的话，移动
    有序表后插入
19.             int j = i-dk;
20.             int x = a[i];          //复制为哨兵，即存储待排序元素
21.             a[i] = a[i-dk];        //首先后移一个元素
22.             while(x < a[j]){     //查找在有序表的插入位置
23.                 a[j+dk] = a[j];
24.                 j -= dk;           //元素后移
25.             }
26.             a[j+dk] = x;           //插入到正确位置
27.         }
28.         print(a, n,i );
29.     }
30.
31. }
32.
33. /**
34.  * 先按增量 d（n/2,n 为要排序数的个数进行希尔排序
35.  *
36.  */
37. void shellSort(int a[], int n){
38.
39.     int dk = n/2;
40.     while( dk >= 1  ){
41.         ShellInsertSort(a, n, dk);
42.         dk = dk/2;
43.     }
44. }
45. int main(){
46.     int a[8] = {3,1,5,7,2,4,9,6};
```

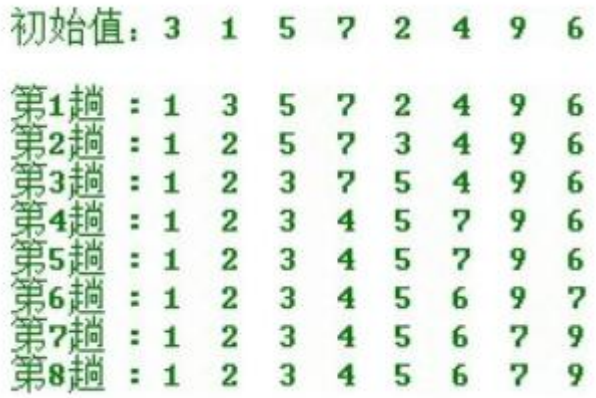
```
47.    //ShellInsertSort(a,8,1); //直接插入排序
48.    shellSort(a,8);          //希尔插入排序
49.    print(a,8,8);
50. }
```

希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于增量因子序列 d 的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的增量因子序列的方法。增量因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：增量因子中除 1 外没有公因子，且最后一个增量因子必须为 1。希尔排序方法是一个不稳定的排序方法

3. 选择排序—简单选择排序

基本思想：

在要排序的一组数中，选出最小（或者最大）的一个数与第 1 个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第 2 个位置的数交换，依次类推，直到第 n-1 个元素（倒数第二个数）和第 n 个元素（最后一个数）比较为止。
简单选择排序的示例：



操作方法：

第一趟，从 n 个记录中找出关键码最小的记录与第一个记录交换；
第二趟，从第二个记录开始的 n-1 个记录中再选出关键码最小的记录与第二个记录交换；
以此类推.....
第 i 趟，则从第 i 个记录开始的 n-i+1 个记录中选出关键码最小的记录与第 i 个记录交换，直到整个序列按关键码有序。

算法实现：

```
1.  void print(int a[], int n ,int i){
2.      cout<<"第"<<i+1 <<"趟 : ";
```

```
3.      for(int j= 0; j<8; j++){
4.          cout<<a[j] <<"  ";
5.      }
6.      cout<<endl;
7.  }
8.  /**
9.   * 数组的最小值
10.  *
11.  * @return int 数组的键值
12.  */
13. int SelectMinKey(int a[], int n, int i)
14. {
15.     int k = i;
16.     for(int j=i+1 ;j< n; ++j) {
17.         if(a[k] > a[j]) k = j;
18.     }
19.     return k;
20. }
21.
22. /**
23.  * 选择排序
24.  *
25.  */
26. void selectSort(int a[], int n){
27.     int key, tmp;
28.     for(int i = 0; i< n; ++i) {
29.         key = SelectMinKey(a, n,i);          //选择最小的元素
30.         if(key != i){
31.             tmp = a[i];  a[i] = a[key]; a[key] = tmp; //最小元素与第 i 位置元素互换
32.         }
33.         print(a,  n , i);
34.     }
35. }
36. int main(){
37.     int a[8] = {3,1,5,7,2,4,9,6};
38.     cout<<"初始值：";
39.     for(int j= 0; j<8; j++){
40.         cout<<a[j] <<"  ";
41.     }
42.     cout<<endl<<endl;
43.     selectSort(a, 8);
44.     print(a,8,8);
45. }
```

简单选择排序的改进——二元选择排序

简单选择排序，每趟循环只能确定一个元素排序后的定位。我们可以考虑改进为每趟循环确定两个元素（当前趟最大和最小记录）的位置,从而减少排序所需的循环次数。改进后对 n 个数据进行排序，最多只需进行[n/2]趟循环即可。具体实现如下：

```
1.  void SelectSort(int r[],int n) {
2.     int i ,j , min ,max, tmp;
3.     for (i=1 ;i <= n/2;i++) {
4.        // 做不超过 n/2 趟选择排序
5.        min = i; max = i ; //分别记录最大和最小关键字记录位置
6.        for (j= i+1; j<= n-i; j++) {
7.           if (r[j] > r[max]) {
8.              max = j ; continue ;
9.           }
10.          if (r[j]< r[min]) {
11.             min = j ;
12.          }
13.    }
14.    //该交换操作还可分情况讨论以提高效率
15.    tmp = r[i-1]; r[i-1] = r[min]; r[min] = tmp;
16.    tmp = r[n-i]; r[n-i] = r[max]; r[max] = tmp;
17.
18.  }
19. }
```

4. 选择排序—堆排序

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

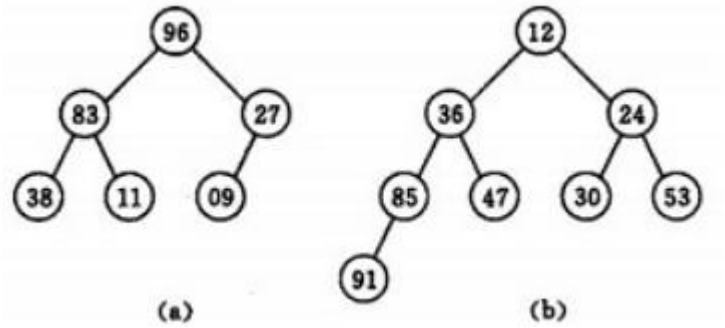
基本思想：

堆的定义如下：具有 n 个元素的序列（k1,k2,...,kn),当且仅当满足

时称之为堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最小项（小顶堆）。若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点（堆顶元素）的值是最小(或最大)的。如：

（a）大顶堆序列：（96, 83,27,38,11,09)

(b) 小顶堆序列：（12，36，24，85，47，30，53，91）



初始时把要排序的 n 个数的序列看作是一棵**顺序存储的二叉树（一维数组存储二叉树）**，调整它们的存储序，使之成为一个堆，将堆顶元素输出，得到 n 个元素中最小(或最大)的元素，这时堆的根节点的数最小（或者最大）。然后对前面(n-1)个元素重新调整使之成为堆，输出堆顶元素，得到 n 个元素中次小(或次大)的元素。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有 n 个节点的有序序列。称这个过程为**堆排序**。

因此，实现堆排序需解决两个问题：

1. 如何将 n 个待排序的数建成堆；
2. 输出堆顶元素后，怎样调整剩余 n-1 个元素，使其成为一个新堆。

首先讨论第二个问题：输出堆顶元素后，对剩余 n-1 元素重新建成堆的调整过程。

调整小顶堆的方法：

1）设有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。将堆底元素送入堆顶（（最后一个元素与堆顶进行交换），堆被破坏，其原因仅是根结点不满足堆的性质。

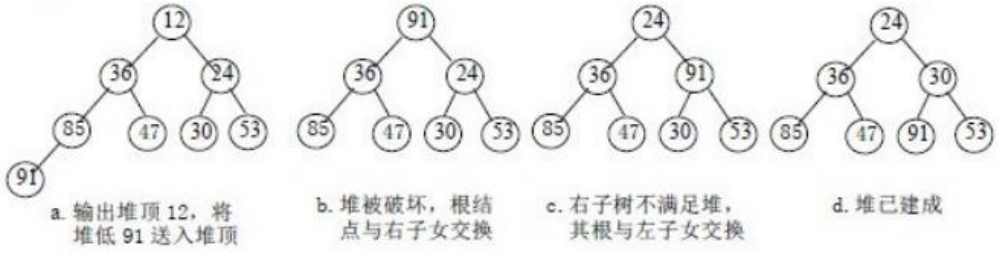
2）将根结点与左、右子树中较小元素的进行交换。

3）若与左子树交换：如果左子树堆被破坏，即左子树的根结点不满足堆的性质，则重复方法 （2）.

4）若与右子树交换，如果右子树堆被破坏，即右子树的根结点不满足堆的性质。则重复方法 （2）.

5）继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。

称这个自根结点到叶子结点的调整过程为筛选。如图：



再讨论对 n 个元素初始建堆的过程。

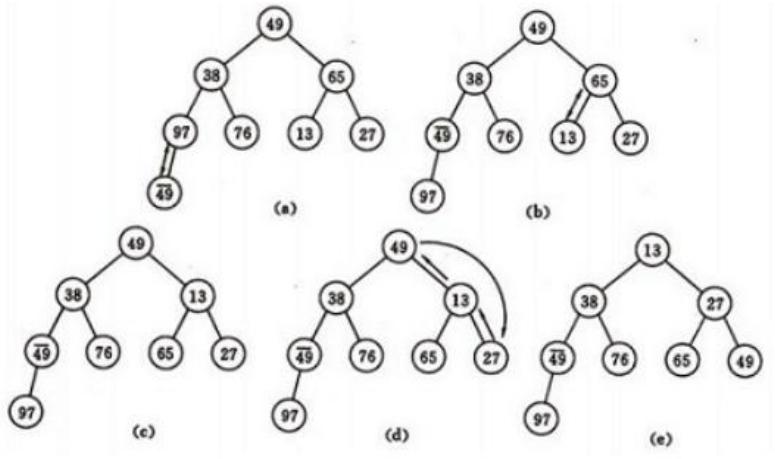
建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。

1）n 个结点的完全二叉树，则最后一个结点是第$\lfloor n/2 \rfloor$个结点的子树。

2）筛选从第$\lfloor n/2 \rfloor$个结点为根的子树开始，该子树成为堆。

3）之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

如图建堆初始过程：无序序列：（49，38，65，97，76，13，27，49）



（a）无序序列；（b）97 被筛选之后的状态；（c）65 被筛选之后的状态；
（d）38 被筛选之后的状态；（e）49 被筛选之后建成的堆

算法的实现：

从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

```
1.  void print(int a[], int n){
2.      for(int j= 0; j<n; j++){
3.          cout<<a[j] <<"  ";
4.      }
5.      cout<<endl;
6.  }
7.
8.
9.
10. /**
11.  * 已知 H[s…m]除了 H[s] 外均满足堆的定义
12.  * 调整 H[s],使其成为大顶堆.即将对第 s 个结点为根的子树筛选,
13.  *
14.  * @param H 是待调整的堆数组
15.  * @param s 是待调整的数组元素的位置
16.  * @param length 是数组的长度
```

```
 *
 */
void HeapAdjust(int H[],int s, int length)
{
    int tmp  = H[s];
     int child = 2*s+1; //左孩子结点的位置。(i+1 为当前调整结点的右孩子结点的位置)
    while (child < length) {
         if(child+1 <length && H[child]<H[child+1]) { // 如果右孩子大于左孩子(找到比当前待调整结点大的孩子结点)
            ++child ;
        }
        if(H[s]<H[child]) {  // 如果较大的子结点大于父结点
            H[s] = H[child]; // 那么把较大的子结点往上移动，替换它的父结点
            s = child;      // 重新设置 s ,即待调整的下一个结点的位置
            child = 2*s+1;
        } else {           // 如果当前待调整结点大于它的左右孩子，则不需要调整，直接退出
             break;
        }
        H[s] = tmp;        // 当前待调整的结点放到比其大的孩子结点位置上
    }
    print(H,length);
}


/**
 * 初始堆进行调整
 * 将 H[0..length-1]建成堆
 * 调整完之后第一个元素是序列的最小的元素
 */
void BuildingHeap(int H[], int length)
{
    //最后一个有孩子的节点的位置 i=  (length -1) / 2
    for (int i = (length -1) / 2 ; i >= 0; --i)
        HeapAdjust(H,i,length);
}
/**
 * 堆排序算法
 */
void HeapSort(int H[],int length)
{
    //初始堆
    BuildingHeap(H, length);
```

```
58.    //从最后一个元素开始对序列进行调整
59.    for (int i = length - 1; i > 0; --i)
60.    {
61.        //交换堆顶元素 H[0]和堆中最后一个元素
62.        int temp = H[i]; H[i] = H[0]; H[0] = temp;
63.        //每次交换堆顶元素和堆中最后一个元素之后，都要对堆进行调整
64.        HeapAdjust(H,0,i);
65.    }
66. }
67.
68. int main(){
69.    int H[10] = {3,1,5,7,2,4,9,6,10,8};
70.    cout<<"初始值：";
71.    print(H,10);
72.    HeapSort(H,10);
73.    //selectSort(a, 8);
74.    cout<<"结果：";
75.    print(H,10);
76.
77. }
```

分析：

设树深度为 $k=\lfloor \log_2 n \rfloor+1$。。从根到叶的筛选，元素比较次数至多 2(k-1)次，交换记录至多 k 次。所以，在建好堆后，排序过程中的筛选次数不超过下式：

$$2(\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-2) \rfloor+\cdots+\log_2 2)<2n(\lfloor \log_2 n \rfloor)$$

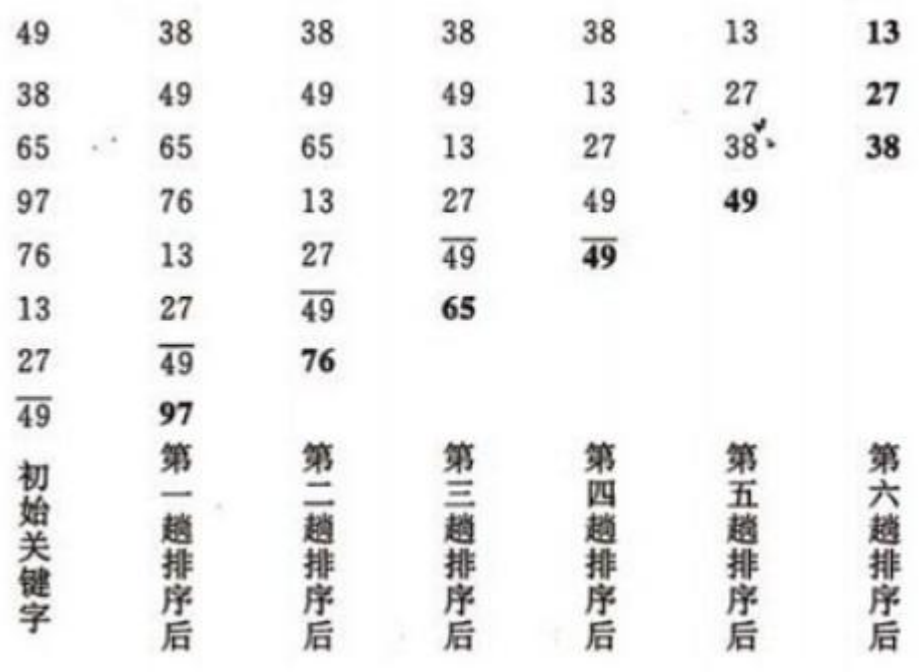
而建堆时的比较次数不超过 4n 次，因此堆排序最坏情况下，时间复杂度也为：O(nlogn)。

5. 交换排序—冒泡排序

基本思想：

在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

冒泡排序的示例：



算法的实现：

```
1.  void bubbleSort(int a[], int n){
2.      for(int i =0 ; i< n-1; ++i) {
3.          for(int j = 0; j < n-i-1; ++j) {
4.              if(a[j] > a[j+1])
5.              {
6.                  int tmp = a[j] ; a[j] = a[j+1] ;  a[j+1] = tmp;
7.              }
8.          }
9.      }
10. }
```

冒泡排序算法的改进

对冒泡排序常见的改进方法是加入一标志性变量 exchange，用于标志某一趟排序过程中是否有数据交换，如果进行某一趟排序时并没有进行数据交换，则说明数据已经按要求排列好，可立即结束排序，避免不必要的比较过程。本文再提供以下两种改进算法：

1．设置一标志性变量 pos,用于记录每趟排序中最后一次进行交换的位置。由于 pos 位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到 pos 位置即可。

改进后算法如下:

```
1.  void Bubble_1 ( int r[], int n) {
```

```
2.      int i= n -1;  //初始时,最后位置保持不变
3.      while ( i> 0) {
4.          int pos= 0; //每趟开始时,无记录交换
5.          for (int j= 0; j< i; j++)
6.              if (r[j]> r[j+1]) {
7.                  pos= j; //记录交换的位置
8.                  int tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;
9.              }
10.         i= pos; //为下一趟排序作准备
11.     }
12. }
```

2．传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者) , 从而使排序趟数几乎减少了一半。
改进后的算法实现为:

```
1.  void Bubble_2 ( int r[], int n){
2.      int low = 0;
3.      int high= n -1; //设置变量的初始值
4.      int tmp,j;
5.      while (low < high) {
6.          for (j= low; j< high; ++j) //正向冒泡,找到最大者
7.              if (r[j]> r[j+1]) {
8.                  tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;
9.              }
10.         --high;                 //修改 high 值, 前移一位
11.         for ( j=high; j>low; --j) //反向冒泡,找到最小者
12.             if (r[j]<r[j-1]) {
13.                 tmp = r[j]; r[j]=r[j-1];r[j-1]=tmp;
14.             }
15.         ++low;                  //修改 low 值,后移一位
16.     }
17. }
```

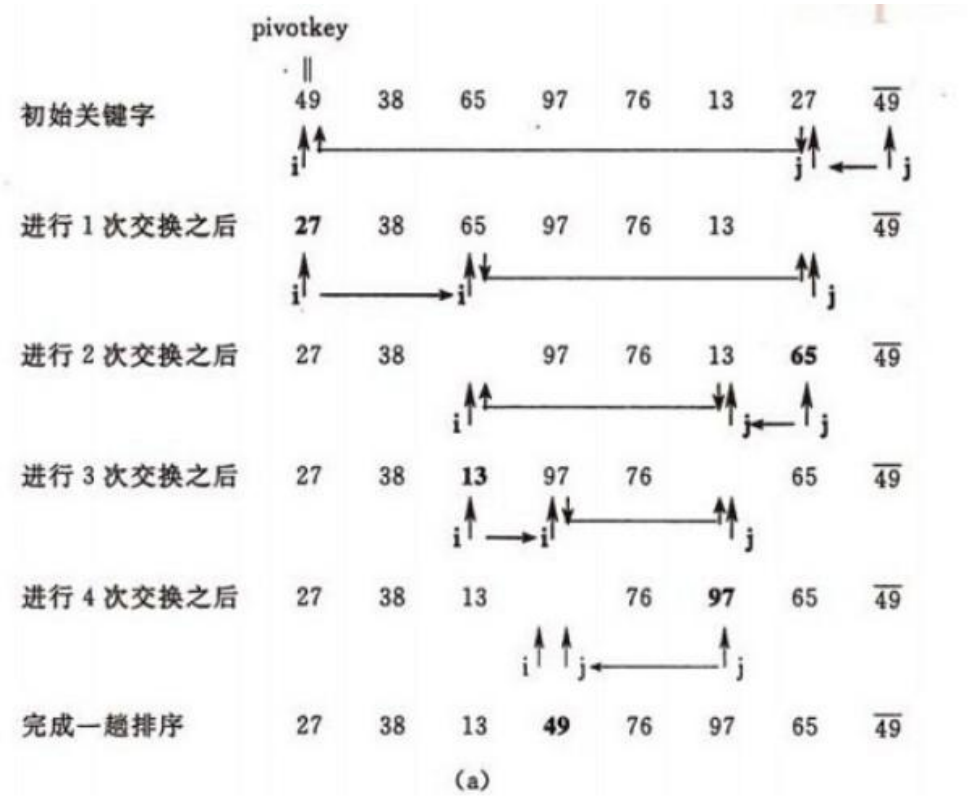
6. 交换排序—快速排序

基本思想：

1）选择一个基准元素,通常选择第一个元素或者最后一个元素,
2）通过一趟排序讲待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的 元素值比基准值大。
3）此时基准元素在其排好序后的正确位置
4）然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

快速排序的示例：

（a）一趟排序的过程：



（b）排序的全过程

算法的实现：

递归实现：

```
1. void print(int a[], int n){
2.     for(int j= 0; j<n; j++){
3.         cout<<a[j] <<"  ";
4.     }
5.     cout<<endl;
```

```
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int partition(int a[], int low, int high)
{
    int privotKey = a[low];                         //基准元素
    while(low < high){                              //从表的两端交替地向中间扫描
        while(low < high  && a[high] >= privotKey) --high;  //从 high 所指位置向前搜索，至多到 low+1 位置。将比基准元素小的交换到低端
        swap(&a[low], &a[high]);
        while(low < high  && a[low] <= privotKey ) ++low;
        swap(&a[low], &a[high]);
    }
    print(a,10);
    return low;
}


void quickSort(int a[], int low, int high){
    if(low < high){
        int privotLoc = partition(a,  low,  high);  //将表一分为二
        quickSort(a,  low,  privotLoc -1);        //递归对低子表递归排序
        quickSort(a,   privotLoc + 1, high);       //递归对高子表递归排序
    }
}

int main(){
    int a[10] = {3,1,5,7,2,4,9,6,10,8};
    cout<<"初始值：";
    print(a,10);
    quickSort(a,0,9);
    cout<<"结果：";
    print(a,10);

}
```

分析：

快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始

序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

快速排序的改进

在本改进算法中,只对长度大于 k 的子序列递归调用快速排序,让原序列基本有序，然后再对整个基本有序序列用插入排序算法排序。实践证明，改进后的算法时间复杂度有所降低，且当 k 取值为 8 左右时,改进算法的性能最佳。算法思想如下：

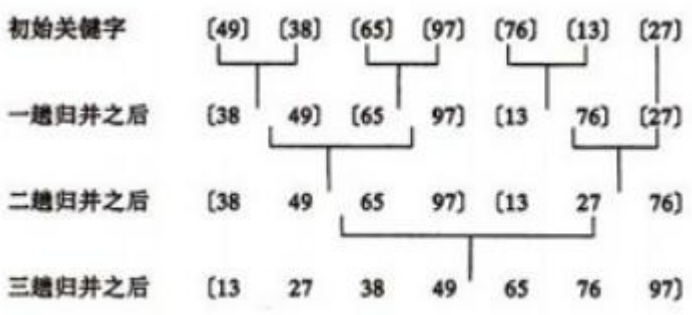
```
1.  void print(int a[], int n){
2.      for(int j= 0; j<n; j++){
3.          cout<<a[j] <<"  ";
4.      }
5.      cout<<endl;
6.  }
7.
8.  void swap(int *a, int *b)
9.  {
10.     int tmp = *a;
11.     *a = *b;
12.     *b = tmp;
13. }
14.
15. int partition(int a[], int low, int high)
16. {
17.     int privotKey = a[low];                 //基准元素
18.     while(low < high){                  //从表的两端交替地向中间扫描
19.         while(low < high  && a[high] >= privotKey) --high; //从 high 所指位置向前搜索，
    至多到 low+1 位置。将比基准元素小的交换到低端
20.         swap(&a[low], &a[high]);
21.         while(low < high  && a[low] <= privotKey ) ++low;
22.         swap(&a[low], &a[high]);
23.     }
24.     print(a,10);
25.     return low;
26. }
27.
28.
29. void qsort_improve(int r[ ],int low,int high, int k){
30.     if( high -low > k ) { //长度大于 k 时递归, k 为指定的数
31.         int pivot = partition(r, low, high); // 调用的 Partition 算法保持不变
32.         qsort_improve(r, low, pivot - 1,k);
33.         qsort_improve(r, pivot + 1, high,k);
34.     }
```

```
}
void quickSort(int r[], int n, int k){
    qsort_improve(r,0,n,k);//先调用改进算法 Qsort 使之基本有序

    //再用插入排序对基本有序序列排序
    for(int i=1; i<=n;i ++){
        int tmp = r[i];
        int j=i-1;
        while(tmp < r[j]){
            r[j+1]=r[j]; j=j-1;
        }
        r[j+1] = tmp;
    }

}



int main(){
    int a[10] = {3,1,5,7,2,4,9,6,10,8};
    cout<<"初始值：";
    print(a,10);
    quickSort(a,9,4);
    cout<<"结果：";
    print(a,10);

}
```

7. 归并排序（Merge Sort）

基本思想：

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。归并排序示例：



合并方法：

设 r[i…n]由两个有序子表 r[i…m]和 r[m+1…n]组成，两个子表长度分别为 n-i +1、n-m。

1. j=m+1；k=i；i=i; //置两个子表的起始下标及辅助数组的起始下标
2. 若 i>m 或 j>n，转⑷ //其中一个子表已合并完，比较选取结束
3. //选取 r[i]和 r[j]较小的存入辅助数组 rf
   如果 r[i]<r[j]，rf[k]=r[i]； i++； k++； 转⑵
   否则，rf[k]=r[j]； j++； k++； 转⑵
4. //将尚未处理完的子表中元素存入 rf
   如果 i<=m，将 r[i…m]存入 rf[k…n] //前一子表非空
   如果 j<=n， 将 r[j…n] 存入 rf[k…n] //后一子表非空
5. 合并结束。

```
1.  //将 r[i…m]和 r[m +1 …n]归并到辅助数组 rf[i…n]
2.  void Merge(ElemType *r,ElemType *rf, int i, int m, int n)
3.  {
4.     int j,k;
5.     for(j=m+1,k=i; i<=m && j <=n ; ++k){
6.        if(r[j] < r[i]) rf[k] = r[j++];
7.        else rf[k] = r[i++];
8.     }
9.     while(i <= m)  rf[k++] = r[i++];
10.    while(j <= n)  rf[k++] = r[j++];
11. }
```

归并的迭代算法

1 个元素的表总是有序的。所以对 n 个元素的待排序列，每个元素可看成 1 个有序子表。对子表两两合并生成 n/2 个子表，所得子表除最后一个子表长度可能为 1 外，其余子表长度均为 2。再进行两两合并，直到生成 n 个元素按关键码有序的表。

```
1.  void print(int a[], int n){
2.     for(int j= 0; j<n; j++){
3.        cout<<a[j] <<"  ";
```

```
4.      }
5.      cout<<endl;
6.  }
7.
8.  //将 r[i…m]和 r[m +1 …n]归并到辅助数组 rf[i…n]
9.  void Merge(ElemType *r,ElemType *rf, int i, int m, int n)
10. {
11.     int j,k;
12.     for(j=m+1,k=i; i<=m && j <=n ; ++k){
13.         if(r[j] < r[i]) rf[k] = r[j++];
14.         else rf[k] = r[i++];
15.     }
16.     while(i <= m)  rf[k++] = r[i++];
17.     while(j <= n)  rf[k++] = r[j++];
18.     print(rf,n+1);
19. }
20.
21. void MergeSort(ElemType *r, ElemType *rf, int lenght)
22. {
23.     int len = 1;
24.     ElemType *q = r ;
25.     ElemType *tmp ;
26.     while(len < lenght) {
27.         int s = len;
28.         len = 2 * s ;
29.         int i = 0;
30.         while(i+ len <lenght){
31.             Merge(q, rf,  i, i+ s-1, i+ len-1 ); //对等长的两个子表合并
32.             i = i+ len;
33.         }
34.         if(i + s < lenght){
35.             Merge(q, rf,  i, i+ s -1, lenght -1); //对不等长的两个子表合并
36.         }
37.         tmp = q; q = rf; rf = tmp; //交换 q,rf，以保证下一趟归并时，仍从 q 归并到 rf
38.     }
39. }
40.
41.
42. int main(){
43.     int a[10] = {3,1,5,7,2,4,9,6,10,8};
44.     int b[10];
45.     MergeSort(a, b, 10);
46.     print(b,10);
47.     cout<<"结果： ";
```

```
48.    print(a,10);
49.
50. }
```

两路归并的递归算法

```
1.  void MSort(ElemType *r, ElemType *rf,int s, int t)
2.  {
3.     ElemType *rf2;
4.     if(s==t) r[s] = rf[s];
5.     else
6.     {
7.        int m=(s+t)/2;        /*平分*p 表*/
8.        MSort(r, rf2, s, m);      /*递归地将 p[s…m]归并为有序的 p2[s…m]*/
9.        MSort(r, rf2, m+1, t);    /*递归地将 p[m+1…t]归并为有序的 p2[m+1…t]*/
10.       Merge(rf2, rf, s, m+1,t);  /*将 p2[s…m]和 p2[m+1…t]归并到 p1[s…t]*/
11.    }
12. }
13. void MergeSort_recursive(ElemType *r, ElemType *rf, int n)
14. {  /*对顺序表*p 作归并排序*/
15.    MSort(r, rf,0, n-1);
16. }
```

8. 桶排序/基数排序(Radix Sort)

说基数排序之前，我们先说桶排序：

基本思想：是将阵列分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递回方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。当要被排序的阵列内的数值是均匀分配的时候，桶排序使用线性时间（Θ（n））。但桶排序并不是 比较排序，他不受到 O(n log n) 下限的影响。

简单来说，就是把数据分组，放在一个个的桶中，然后对每个桶里面的在进行排序。

例如要对大小为[1..1000]范围内的 n 个整数 A[1..n]排序

首先，可以把桶设为大小为 10 的范围，具体而言，设集合 B[1]存储[1..10]的整数，集合 B[2]存储 (10..20]的整数，……集合 B[i]存储((i-1)*10, i*10]的整数，i = 1,2,..100。总共有 100 个桶。

然后，对 A[1..n]从头到尾扫描一遍，把每个 A[i]放入对应的桶 B[j]中。 再对这 100 个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任 何排序法都可以。

最后，依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这 样就得到所有数字排好序的一个序列了。

假设有 n 个数字，有 m 个桶，如果数字是平均分布的，则每个桶里面平均有 n/m 个数字。如果

对每个桶中的数字采用快速排序，那么整个算法的复杂度是

O(n + m * n/m*log(n/m)) = O(n + nlogn - nlogm)

从上式看出，当 m 接近 n 的时候，桶排序复杂度接近 O(n)

当然，以上复杂度的计算是基于输入的 n 个数字是平均分布这个假设的。这个假设是很强的 ，实际应用中效果并没有这么好。如果所有的数字都落在同一个桶中，那就退化成一般的排序了。

前面说的几大排序算法 ，大部分时间复杂度都是 O（n2），也有部分排序算法时间复杂度是 O(nlogn)。而桶式排序却能实现 O（n）的时间复杂度。但桶排序的缺点是：

1）首先是空间复杂度比较高，需要的额外开销大。排序有两个数组的空间开销，一个存放待排序数组，一个就是所谓的桶，比如待排序值是从 0 到 m-1，那就需要 m 个桶，这个桶数组就要至少 m 个空间。

2）其次待排序的元素都要在一定的范围内等等。

桶式排序是一种分配排序。分配排序的特定是不需要进行关键码的比较，但前提是要知道待排序列的一些具体情况。

分配排序的基本思想：说白了就是进行多次的桶式排序。

基数排序过程无须比较关键字，而是通过“分配”和“收集”过程来实现排序。它们的时间复杂度可达到线性阶：O(n)。

实例:

扑克牌中 52 张牌，可按花色和面值分成两个字段，其大小关系为：

花色： **梅花< 方块< 红心< 黑心**

面值： **2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A**

若对扑克牌按花色、面值进行升序排序，得到如下序列：

即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。

为得到排序结果，我们讨论两种排序方法。

方法 1：先对花色排序，将其分为 4 个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后，将 4 个组连接起来即可。

方法 2：先按 13 个面值给出 13 个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成 13 堆。再按花色给出 4 个编号组(梅花、方块、红心、黑心)，将 2 号组中牌取出分别放入对应花色组，再将 3 号组中牌取出分别放入对应花色组，……，这样，4 个花色组中均按面值有序，然后，将 4 个花色组依次连接起来即可。

设 n 个元素的待排序列包含 d 个关键码{k1，k2，…，kd}，则称序列对关键码{k1，k2，…，kd}有序是指：对于序列中任两个记录 r[i]和 r[j](1≤i≤j≤n)都满足下列有序关系：

$$(k_i^1, k_i^2, \quad , k_i^d) < (k_j^1, k_j^2, \quad , k_j^d)$$

其中 k1 称为最主位关键码，kd 称为最次位关键码

两种多关键码排序方法：

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：

最高位优先(Most Significant Digit first)法，简称 MSD 法：

1）先按 k1 排序分组，将序列分成若干子序列，同一组序列的记录中，关键码 k1 相等。

2）再对各组按 k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 kd 对各子组排序后。

3）再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是 MSD 法。

最低位优先(Least Significant Digit first)法，简称 LSD 法：

1) 先从 kd 开始排序，再对 kd-1 进行排序，依次重复，直到按 k1 排序分组分成最小的子序列后。

2) 最后将各个子序列连接起来，便可得到一个有序的序列，扑克牌按花色、面值排序中介绍的方法二即是 LSD 法。

基于 LSD 方法的链式基数排序的基本思想

“多关键字排序”的思想实现“单关键字排序”。对数字型或字符型的单关键字，可以看作由多个数位或多个字符构成的多关键字，此时可以采用“分配-收集”的方法进行排序，这一过程称作基数排序法，其中每个数字或字符可能的取值个数称为基数。比如，扑克牌的花色基数为 4，面值基数为 13。在整理扑克牌时，既可以先按花色整理，也可以先按面值整理。按花色整理时，先按红、黑、方、花的顺序分成 4 摞（分配），再按此顺序再叠放在一起（收集），然后按面值的顺序分成 13 摞（分配），再按此顺序叠放在一起（收集），如此进行二次分配和收集即可将扑克牌排列有序。

基数排序:

是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是

高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

算法实现：

```
1.  Void RadixSort(Node L[],length,maxradix)
2.  {
3.    int m,n,k,lsp;
4.    k=1;m=1;
5.    int temp[10][length-1];
6.    Empty(temp); //清空临时空间
7.    while(k<maxradix) //遍历所有关键字
8.    {
9.      for(int i=0;i<length;i++) //分配过程
10.   {
11.     if(L[i]<m)
12.       Temp[0][n]=L[i];
13.     else
14.       Lsp=(L[i]/m)%10; //确定关键字
15.     Temp[lsp][n]=L[i];
16.     n++;
17.   }
18.   CollectElement(L,Temp); //收集
19.   n=0;
20.   m=m*10;
21.  k++;
22. }
23. }
```

9、总结

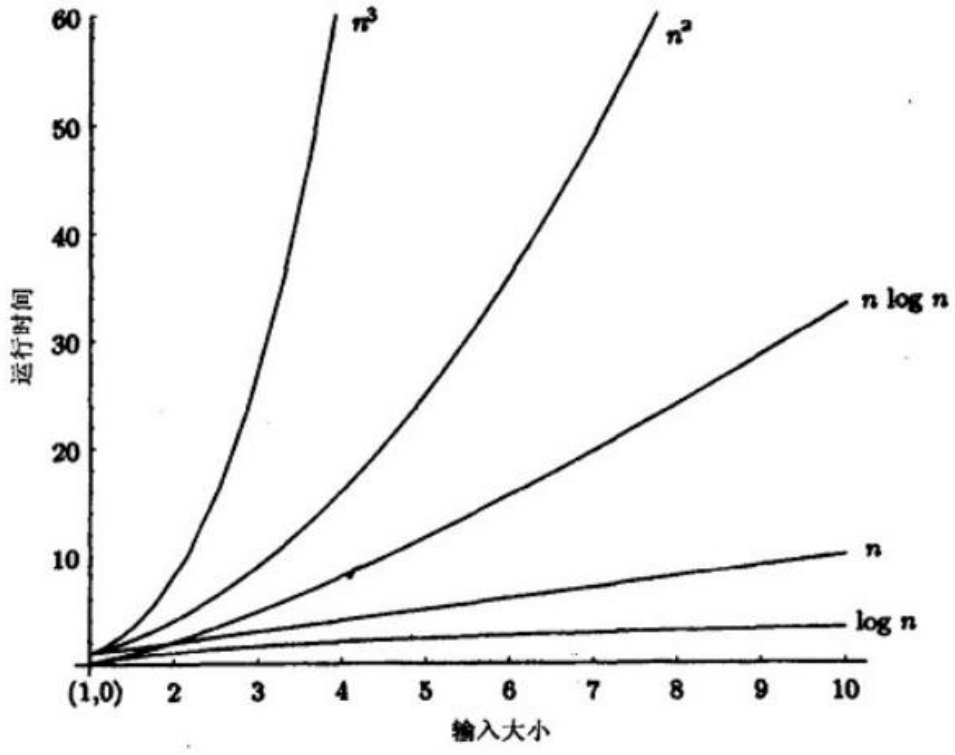
各种排序的稳定性，时间复杂度和空间复杂度总结：

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

所以对 n 较大的排序记录。一般的选择都是时间复杂度为 O(nlog2n)的排序方法。

我们比较时间复杂度函数的情况：



时间复杂度函数 0(n)的增长情况

表 1.1　不同大小输入的运行时间

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16 384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32 768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65 536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	$10^{19\,709}$ cent
131 072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	$10^{39\,438}$ cent
262 144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	$10^{78\,894}$ cent
524 288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	$10^{157\,808}$ cent
048 576	0.02 μ	1048.60 μ	20 972 μ	18.3 min	37 years	$10^{315\,634}$ cent

注：nsec 为纳秒，μ 为毫秒，sec 为秒，min 为分钟，hrs 为小时，days 为天，mnths 为月，years 为年，cent 为世纪。

所以对 n 较大的排序记录。一般的选择都是时间复杂度为 O(nlog2n)的排序方法。

时间复杂度来说：

(1)平方阶(O(n^2))排序

各类简单排序:直接插入、直接选择和冒泡排序；

(2)线性对数阶(O(nlog2n))排序

快速排序、堆排序和归并排序；

(3)O($n^{1+§}$))排序,§是介于 0 和 1 之间的常数。

希尔排序

(4)线性阶(O(n))排序

基数排序，此外还有桶、箱排序。

说明：

当原表有序或基本有序时，直接插入排序和冒泡排序将大大减少比较次数和移动记录的次数，时间复杂度可降至 O（n）；

而快速排序则相反，当原表基本有序时，将蜕化为冒泡排序，时间复杂度提高为 O（n2）；

原表是否有序，对简单选择排序、堆排序、归并排序和基数排序的时间复杂度影响不大。

稳定性：

排序算法的稳定性:若待排序的序列中，存在多个具有相同关键字的记录，经过排序， 这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对 次序发生了改变，则称该算法是不稳定的。

稳定性的好处：排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。另外，如果排序算法稳定，可以避免多余的比较；
稳定的排序算法：冒泡排序、插入排序、归并排序和基数排序
不是稳定的排序算法：选择排序、快速排序、希尔排序、堆排序

选择排序算法准则：

每种排序算法都各有优缺点。因此，在实用时需根据不同情况适当选用，甚至可以将多种方法结合起来使用。
选择排序算法的依据
影响排序的因素有很多，平均时间复杂度低的算法并不一定就是最优的。相反，有时平均时间复杂度高的算法可能更适合某些特殊情况。同时，选择算法时还得考虑它的可读性，以利于软件的维护。一般而言，需要考虑的因素有以下四点：
1．待排序的记录数目 n 的大小；
2．记录本身数据量的大小，也就是记录中除关键字外的其他信息量的大小；
3．关键字的结构及其分布情况；
4．对排序稳定性的要求。

设待排序元素的个数为 n.

1）当 n 较大，则应采用时间复杂度为 O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。
快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
堆排序 ： 如果内存空间允许且要求稳定性的，
归并排序：它有一定数量的数据移动，所以我们可能过与插入排序组合，先获得一定长度的序列，然后再合并，在效率上将有所提高。
2） 当 n 较大，内存空间允许，且要求稳定性 =》归并排序
3）当 n 较小，可采用直接插入或直接选择排序。
直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。
直接选择排序 ：元素分布有序，如果不要求稳定性，选择直接选择排序
5）一般不使用或不直接使用传统的冒泡排序。
6）基数排序
它是一种稳定的排序算法，但有一定的局限性：
1、关键字可分解。
2、记录的关键字位数较少，如果密集更好
3、如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。
当 n 较大，则应采用时间复杂度为 O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。
快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机

分布时，快速排序的平均时间最短

第五部分 嵌入式相关

1、Linux 下进程通信的八种方法

Linux 下进程通信的八种方法：管道(pipe)，命名管道(FIFO)，内存映射(mapped memeory)，消息队列(message queue)，共享内存(shared memory)，信号量(semaphore)，信号(signal)，套接字(Socket)

(1) 管道（pipe）：管道允许一个进程和另一个与它有共同祖先的进程之间进行通信；

(2) 命名管道（FIFO）：类似于管道，但是它可以用于任何两个进程之间的通信，命名管道在文件系统中有对应的文件名。命名管道通过命令 mkfifo 或系统调用 mkfifo 来创建；

(3) 信号（signal）：信号是比较复杂的通信方式，用于通知接收进程有某种事情发生，除了用于进程间通信外，进程还可以发送信号给进程本身；Linux 除了支持 UNIX 早期信号语义函数 signal 外，还支持语义符合 POSIX.1 标准的信号函数 sigaction(实际上，该函数是基于 BSD 的，BSD 即能实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数的功能);

(4) 内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它；

(5) 消息队列（message queue）：消息队列是消息的连接表，包括 POSIX 消息对和 System V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能成该无格式字节流以及缓冲区大小受限等缺点；

(6) 信号量（semaphore）：信号量主要作为进程间以及同进程不同线程之间的同步手段；

(7) 共享内存 （shared memory）：它使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。这是针对其他通信机制运行效率较低而设计的。它往往与其他通信机制，如信号量结合使用，以达到进程间的同步及互斥；

(8) 套接字（Socket）：它是更为通用的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 UNIX 系统的 BSD 分支开发出来的，但现在一般可以移植到其他类 UNIX 系统上：Linux 和 System V 的变种都支持套接字。

2、CISC 体系与 RISC 体系分别指什么？

RISC(精简指令集计算机)和 CISC(复杂指令集计算机)是当前 CPU 的两种架构。它们的区别在于不同的 CPU 设计理念和方法.

CISC 架构，它的设计目的是要用最少的机器语言指令来完成所需的计算任务。这种架构会增加 CPU 结构的复杂性和对 CPU 工艺的要求，但对于编译器的开发十分有利, 今天只有 Intel 及其兼容 CPU 还在使用 CISC 架构。精简指令集，RISC 微处理器不仅精简了指令系统，采用超标量和超流水线结构；它们的指令数目只有几十条，却大大增强了并行处理能力。性能特点一：由于指令集简化后，流水线以及常用指令均可用硬件执行； 性能特点二：采用大量的寄存器，使大部分指令操作都在寄存器之间进行，提高了处理速度； 性能特点三：

采用缓存—主机—外存三级存储结构，使取数与存数指令分开执行，使处理器可以完成尽可能多的工作，且不因从存储器存取信息而放慢处理速度。

3、什么是中断？中断发生时 CPU 做什么工作？

中断是 CPU 响应外设需求的一种模式，在外设需要 CPU 时，会向中断控制器发送中断请求，这时 CPU 要保护现场，即把正在运行的程序保存起来，一般是把状态压入堆栈，然后读中断号，启动相应的中断服务程序，服务完成后，载入保护现场，即把堆栈的数据弹出，继续运行之前的程序。

中断：是指当主机接到外界硬件(如 I/O 设备)发来的信号时，马上停止原来的工作，转去处理这一事件，在处理完了以后，主机又回到原来的工作继续工作。中断是机器 BIOS 中最重要的概念，当系统外部设备有请求时，比如鼠标，串口数据同步请求等，都会产生中断，这在一个单 CPU 的环境下，系统会保存当前机器状态，响应这个请求，在请求完成后，就恢复设备状态；

4、中断与异常有何区别？

异常：在产生时必须考虑与处理器的时钟同步，实践上，异常也称为同步中断。在处理器执行到由于编程失误而导致的错误指令时，或者在执行期间出现特殊情况(如缺页)，必须靠内核处理的时候，处理器就会产生一个异常。

中断应该是指外部硬件产生的一个电信号，从 cpu 的中断引脚进入，打断 cpu 当前的运行；所谓异常，是指软件运行中发生了一些必须作出处理的事件，cpu 自动产生一个陷入来打断当前运行，转入异常处理流程。

5、简述 I2C 传输方式？

I2C 协议：是单片机与其他芯片进行通讯的协议：

1：只要求两条总线线路，一条是串行时钟线，一条是串行数据线；

2：通过软件设定地址

3：是一个多主机总线，如果两个或更多主机同时初始化数据传送可通过冲突检测和仲裁防止数据破坏；

4：I2C 总线传输的是数据的总高位

I2C(Inter－Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。从硬件上来说，只有两条线，十分简单，所以很多设备都使用这种协议。通过控制两条线的时序来传递数据。总线上可以挂接很多设备，那么主设备怎么找到对应的从设备呢？每一个从设备都有一个自己的地址，访问时遵守 i2c 总线协议，以 eeprom 为例：高 4 位固定，低三位 A0,A1,A2（由设备在板子上的连接确定的。）,最后一个读写位这个地址由从设备和开发板连线决定的。

因为 s5pv210 处理器集成了 i2c 的控制器，所以我们只需要给几个寄存器赋值就可以使用了，但是有的 cpu 没有集成 i2c 控制器，那么怎么办？

1 外接一个控制器

2 用两个 gpio 模拟时序输出，根据图 可以看出在 scl 为低时改变 sda 的值。这样才有效，发送 1 要在 scl 为低电平的时候进行。

I2C 总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此 I2C 总线

占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本。

6、中断和轮询哪个效率高？怎样决定是采用中断方式还是采用轮询方式去实现驱动？

中断是 CPU 处于被中状态下来接受设备的信号，而轮询是 CPU 主动去查询该设备是否有请求。凡事都是两面性，所以，看效率不能简单的说那个效率高。如果是请求设备是一个频繁请求 cpu 的设备，或者有大量数据请求的网络设备，那么轮询的效率是比中断高。如果是一般设备，并且该设备请求 cpu 的频率比较底，则用中断效率要高一些。

7、异步通信与同步通信的理解？

同步通信中，接收端和发送端的收发时钟是严格同步的，也就是说通信过程中采用同一个公共时钟信号进行同步。由于收发时钟是严格同步，在接收端就不 会因接收时钟和发送时钟不一致所造成的时钟误差积累。正是这种收发时钟的严格同步，才允许在同步通信中，可以用很高的传输速率一次传送几十、几百甚至更多 字节的数据。

异步通信中，接收端和发送端的收发时钟是不需要严格同步的，也就是说在通信过程中收发时钟是异步的，是有小的不同(允许在 5%以内)。正是这 种收发时钟的异步，在接收端就会因接收时钟和发送时钟不一致所造成的时钟误差积累。这就使得异步通信的传输速率低，每次只能传送几位数据。

8、临界区与临界资源的理解？

在多进程系统中，有些资源一次只能让一个进程使用，如打印机，串口等，这样的资源被称为临界资源，在进程中对临界资源实施操作的那段程序就被称为临界区。

9、U-boot 的启动过程？

1、硬件初始化：前 4K 是在 cpu 内部内存中运行的，即 SRAM

1）主要作用设置异常向量表，进入管理模式，屏蔽 IRQ,FIQ 中断，关闭看门狗，关闭中断，关 MMU，cache，时钟，内存，

2）加载 U-Boot 第二阶段代码到 RAM 空间 （选择从哪里启动，是 nor flash 还是 nand flash）

3）软件初始化：

2、扩展内容：

LOGO 显示

3、引导系统

加载操作系统内核，并且给内核传递参数，并且正确挂接完根文件系统以后，uboot 才算结束。

10、简述 ARM-linux 启动过程？

对于 ARM 处理器来说上电或复位后执行的第一条指令，该地址为 0x00000000。对于一般的嵌入式系统，通常把 Flash 等非易失性存储器映射到这个地址处，而 bootloader 就位于该存储器的最前端，所以系统上电或复位后执行的第一段程序便是 bootloader，在 bootloader 中主要完成：

1)设置和初始化 RAM，为调用 Linux 内核做好准备。初始化 RAM 的任务包括设置 CPU 的控制寄存器参数，以便能正常使用 RAM 以及检测 RAM 大小等。

2) 初始化串口，串口在 Linux 的启动过程中有着非常重要的作用，它是 Linux 内核和用户交互的方式之 一。Linux 在启动过程中可以将信息通过串口输出，这样便可清楚的了解 Linux 的启动过程。

3) 检测处理器类型

4) 设置 Linux 启动参数 Bootloader 在执行过程中必须设置和初始化 Linux 的内核启动参数

5) 调用 Linux 内核映像（Linux 内核有两种映像：一种是非压缩内核，叫 Image，另一种是它的压缩版本，叫 zImage。根据内核映像的不同，Linux 内核的启动在开始阶段也有所不同。但为了能使用 zImage，必须在它的开头加上解压缩的代码，将 zImage 解压缩之后才能执行，zImage 的入口程序即为 arch/arm/boot/compressed/head.S。它依次完成以下工作：开启 MMU 和 Cache，调用 decompress_kernel()解压内核，最后通过调用 call_kernel()进入非压缩内核 Image 的启动。Linux 非压缩内核的入口位于文件 /arch/arm/kernel/head-armv.S 中的 text 段。该段的基地址就是压缩内核解压后的跳转地址。该程序通过查找处理器内核类型和处理器类型调用相应的初始化函数，再建立页表，最后跳转到 start_kernel()函数开始内核的初始化工作。）

6）完成剩余的与硬件平台相关的初始化工作，在进行一系列与内核相关的初始化后，调用第一个用户进程－init 进程并等待用户进程的执行，这样整个 Linux 内核便启动完毕。

11、谈谈对中断的顶半部底半部机制的理解？

在 Linux 内核中，为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点，Linux 将中断处理程序分为两个部分：上半部（top half）和下半部（bottom half）。中断处理程序的上半部在接收到一个中断时就立即执行，但只做比较紧急的工作，这些工作都是在所有中断被禁止的情况下完成的，所以要快，否则其它的中断就得 不到及时的处理。那些耗时又不紧急的工作被推迟到下半部去。中断处理程序的下半部分（如果有的话）几乎做了中断处理程序所有的事情。它们最大的不同是上半 部分不可中断，而下半部分可中断。在理想的情况下，最好是中断处理程序上半部分将所有工作都交给下半部分执行，这样的话在中断处理程序上半部分中完成的工 作就很少，也就能尽可能快地返回。但是，中断处理程序上半部分一定要完成一些工作，例如，通过操作硬件对中断的到达进行确认，还有一些从硬件拷贝数据等对 时间比较敏感的工作。剩下的其他工作都可由下半部分执行。

对于上半部分和下半部分之间的划分没有严格的规则，靠驱动程序开发人员自己的编程习惯来划分，不过还是有一些习惯供参考：

如果该任务对时间比较敏感，将其放在上半部中执行。

如果该任务和硬件相关，一般放在上半部中执行。

如果该任务要保证不被其他中断打断，放在上半部中执行（因为这是系统关中断）。

其他不太紧急的任务， 一般考虑在下半部执行。

下半部分并不需要指明一个确切时间，只要把这些任务推迟一点，让它们在系统不太忙并且中断恢复后执行就可以了。通常下半部分在中断处理程序一返回就会马上运行。内核中实现下半部的手段不断演化，目前已经从最原始的 BH（bottom half）衍生出 BH（在 2.5 中去除）、软中断（softirq 在 2.3 引入）、tasklet（在 2.3 引入）、工作队列（work queue 在 2.5 引入）。

尽管上半部和下半部的结合能够改善系统的响应能力，但是，Linux 设备驱动中的中断处理并不一定要分成两个半部。如果中断要处理的工作本身就很少，则完全可以直接在上半部全部完成。

12、对 Linux 设备中字符设备与块设备的理解？

字符设备：字符设备是个能够像字节流（类似文件）一样被访问的设备，由字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少实现 open,close,read 和 write 系统调用。字符终端、串口、鼠标、键盘、摄像头、声卡和显卡等就是典型的字符设备。

块设备：和字符设备类似，块设备也是通过/dev 目录下的文件系统节点来访问。块设备上能够容纳文件系统，如：u 盘，SD 卡，磁盘等。

字符设备和块设备的区别仅仅在于内核内部管理数据的方式，也就是内核及驱动程序之间的软件接口，而这些不同对用户来讲是透明的。在内核中，和字符驱动程序相比，块驱动程序具有完全不同的接口

13、ARM-linux 用户空间和内核空间有什么区别？

用户空间：USR 模式

内核空间：SVC 模式

用户不能直接访问内核空间的地址信息包括代码和数据

对于 4G 的虚拟地址空间的划分：

- 用户空间地址：0x00000000~0xBFFFFFFF
- 内核空间地址：0xC0000000~0xFFFFFFFF

用户空间和内核空间数据的交互通过系统调用（read,write..）

用户空间：包含应用程序和一堆的库，例如：

- printf，malloc:C 库函数
- read,write:系统调用函数，实现的过程在 C 库里

用户空间不能直接访问硬件资源：用户应用程序不能直接访问寄存器对应的物理地址信息

划分用户空间和内核空间的目的：安全的保护，一旦应用程序进行非法的地址访问，操作系统有权将其停止，但是在内核空间编程一定要注意没有人能够管理内核，如果地址非放，系统崩溃！

14、请简述主设备号和次设备号的用途？

主设备号：主设备号标识设备对应的驱动程序。虽然现代的 linux 内核允许多个驱动程序共享主设备号，但我们看待的大多数设备仍然按照“一个主设备对应一个驱动程序”的原则组织。

次设备号：次设备号由内核使用，用于正确确定设备文件所指的设备。依赖于驱动程序的编写方式，我们可以通过次设备号获得一个指向内核设备的直接指针，也可将此设备号当作设备本地数组的索引。

15、linux 内核里面，内存申请有哪几个函数，各自的区别？

Kmalloc()　__get_free_page()　mempool_create()

kmalloc 函数申请的内存可以任意大小，但是 kmalloc 最大只能开辟 128k-16，16 个字节是被页描述符结构占用了。

get_free_page 或 get_free_pages 是申请的内存以页的大小为基准，大小为页的整数倍。get_dma_pages,这个函数除了以上功能外还能支持 DMA 传输。

16、 内核函数 mmap 的实现原理，机制？

mmap 函数实现把一个文件映射到一个内存区域，从而我们可以像读写内存一样读写文件，他比单纯调用 read/write 也要快上许多。在某些时候我们可以把内存的内容拷贝到一个文件中实现内存备份，当然，也可以把文件的内容映射到内存来恢复某些服务。另外，mmap 实现共享内存也是其主要应用之一，mmap 系统调用使得进程之间通过映射同一个普通文件实现共享内存。

17、驱动里面为什么要有并发、互斥的控制？如何实现？

进程同步是一个操作系统级别的概念,是在多道程序的环境下，存在着不同的制约关系，为了协调这种互相制约的关系，实现资源共享和进程协作，从而避免进程之间的冲突，引入了进程同步。

进程互斥是进程之间的间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。

18、什么是链接脚本？其作用是什么？请编写一个简单的链接脚本？

连接器每一个链接过程都由链接脚本(linker script, 一般以 lds 作为文件的后缀名)控制，链接描述脚本描述了各个输入文件的各个 section 如何映射到输出文件的各 section 中，并控制输出文件中 section 和符号的内存布局。

```
{{{
        SECTIONS
        {
              . = 0x33f00000;
              .text : { *(.text) }
              .data ALIGN(4) : { *(.data) }
              .bss ALIGN(4) : { *(.bss) *(COMMON) }
        }
  }}}
```

19、对大端模式、小端模式的理解？

大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。所以低地址保存着高位数 2，前面补 0。