CH1 Quiz*8

1. 关于数组和指针

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

2. static的作用

在C语言中，关键字static有三个明显的作用：

1. 在函数体内，一个被声明为静态的变量仅在本函数内可被使用，但在函数的每次调用中都维持上次修改的值
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有函数访问，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。

3. const的含义与优点

含义：

```
const int a;
int const a;          // 前两个的作用一样，a是一个常整型数。
const int *a;         // a是一个指向常整型数的指针（整型数是不可修改的，但指针可以）
int * const a;        // a是一个指向整型数的常指针（指针指向的整型数是可以修改的，但指针是不可修改的）
```

优点：

- 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。
- 通过给优化器一些附加的信息，使用关键字const也许能产生更紧凑的代码。
- 合理地使用关键字const可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少bug的出现。

4. volatile变量及应用场景

`简介`：当一个对象的值可能会在编译器的控制或监测之外被改变时，该对象应该声明为volatile。因此，编译器执行的某些例行优化行为不能应用在已经指定为volatile的对象上。

`主要目的`:提示编译器，该对象的值可能在编译器未监测到的情况下被改变，因此编译器不能武断地对引用这些对象的代码作优化处理。

应用场景：

- 并行设备的硬件寄存器（如：状态寄存器）
- 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)
- 多线程应用中被几个任务共享的变量

5. 位运算操作

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。

```
#define BIT3 (0x1<<3)
static int a;
void set_bit3(void)
{
  a |= BIT3;
}
void clear_bit3(void)
{
  a &= ~BIT3;
}
```

6. 指针访问绝对地址

求设置一绝对地址为0x67a0的整型变量的值为0xaa66。编译器是一个纯粹的ANSI编译器。

```
int *ptr;
ptr = (int *)0x67a0;
*ptr = 0xaa55;
```

7. 无符号整形数

该无符号整型问题的答案是 ">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

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

8. 不常见结构

以下代码的结果：a = 6, b = 7, c = 12。

```
int a = 5, b = 7, c;
c = a+++b;            // 处理为c = a++ +b;
```

CH2 关键字与运算符

1. 印第安序（P13）

写入0x12345678比较：

Address	0x0000	0x0001	0x0002	0x0003
Big endian	0x12	0x34	0x56	0x78
Little endian	0x78	0x56	0x34	0x12

测试印第安序的代码(**P14**)

2. void关键字（P15）

三个作用：

- 表明一个函数没有返回值
- 表明一个函数没有参数
- 表明一个空类型指针

3. switch case（P18）

- 如缺少break语句，程序将漏斗形下漏执行
- 每个switch需配对自己的default语句

4. break, continue, goto（P19）

5. 存储类型（P23）

- 几个关键字：`auto`, `register`, `static`, `extern`, `struct`, `union`, `enum`.
- 变量存放三个地方：普通内存、运行时的堆栈、CPU内部通用寄存器
- c文件外访问变量，需要使用`extern`关键字，extern两种用法（**P26**）
- 其他类型关键字（**P30-**）：`const`, `sizeof`, `typedef`, `volatile`.

6. 运算符

- 逗号运算符（**P37**）
- 运算优先级（**P39**）

CH3 指针与函数

1. 指针 （P43）

- 指针三要素（**P44**）：指针变量的值、指向地址的内容、变量本身的地址
- ANSI C标准中不允许操作空指针，必须确定指针所指向对象的大小，但GNU C中可以，编译器将其处理为char*

```
void* pvoid;
pvoid ++;            // ANSI C错误，GNU C正确
```

- 指针初始化初始为null，合法无效；有地址，指向另一个变量，合法有效。
- 运算（**P47**）：`减法`、`大小比较`（地址高低）、`加减整数`（偏移地址）、`自增自减`。
- 字符串（**P48**）：

```
char *p, *q;
p = "Hello!";            // 字符串常量的首地址赋给了p
q = p;                    // 同上
q[0]='h';                // 可能映射到ROM等只读内存区
```

若修改定义：

```
char p[]="Hello!";
```

则最后一句话合法

2. 函数（P52-）

1. 断言assert（**P72**）
2. 系统调用和库函数（**P73**）

- `系统调用`：通过软件中断实现的内核代码
- 用户态切换到特权态的唯一方法：中断

CH4 编译、汇编、调试

1. 开发环境图（P84）

2. 目标文件结构图（P88）

3. 编译器和汇编器的作用

`编译器`：编译器将C文件转换成为汇编文件

`汇编器`：汇编器将汇编文件转换成为二进制指令流*.o文件（目标文件）每个目标文件是独立编址的，也就是说每个目标文件的第一条指令都从相同的地址开始存放

4. 链接器的作用

- 将多个目标文件或库文件按照各文件中段进行统一编址
- 生成一个完整的统一的地址映像
- 嵌入式系统中一般生成一个绝对地址映像
- 在有MMU的系统中可以为每个任务单独分配一个地址空间

5. Myproject依赖关系（P100）

6. makefile(P101)

CH5-6 存储器与指针

1. 指针与数组

1. c语言中只支持一维数组，且只支持静态数组
2. 可用指针行为替代数组下标运算

```
int a[4], *p;
p = a;           //等价于 p =&a[0];
*(a+2) = 0;      //等价于 a[2] = 0;
p[2] = 0;        //等价于 a[2] = 0;
```

3. 与二维数组

   以下等价：

```
i = calendar[4][7];
i = *(calendar[4] + 7);
i = *(*(calendar + 4) +7 );
```

4. 数组作为函数参数，传入的是数组首地址指针（**P117**）
5. 与字符串 以下p[0]=a[0]='h'

```
char *p = "hello, world!";
char a[] = "hello, world!";
```

2. 函数指针

1. 定义

```
int (*fp)(int);          //√ 函数指针(函数返回值为整型)
int *fp(int);            //√ 返回值为指针的函数声明
int *(*fp)(int);         //√ 函数指针(函数返回值为整型指针)
int *(*fp[10])(int);     //√ 函数指针数组(函数返回值为整型指针)
int **fp[10](int);       //× 错就是错了,啥也别说了
```

2. 调用

```
int *myfunction(int);        //返回int指针的函数
int *(*fp)(int);             //返回int指针的函数指针
int *ptr;
fp=myfunction;               //fp赋初值，指向myfunction
ptr=(*fp)(3);                //==myfunction(3)
ptr=fp(4);                   //==(*fp)(4)
```

   - 不能将普通变量地址赋值给函数指针
   - 不能将函数调用赋值给函数指针

3. 用途

   - `多态（polymorphism）`：一个接口，多种方法，一个名字定义不同函数，函数执行不同但类似的操作。

     ```
     double add(double,double);
     double sub(double,double);
     double (*oper_func[])(double,double)={add,sub,...};
     result=oper_func[oper](op1,op2);             //替代switch case
     ```

- `回调（call-back）` **P124**：调用者将自己的某段程序通过函数指针的方式传入被调用者,以使后者在某些触发条件下执行该段程序。
- `多线程（multithreading）` **P125**：将函数指针传入负责建立多线程的API，例如win32的CreateThread(...pF...)，所有线程共享进程的状态和资源。

  优势：

  - 比新进程的创建时间开销少
  - 切换时间少
  - 通信效率高，不需要内核的干预

3. 栈

1. 内存陷阱

   局部变量使用时需要注意（**P129**）：

   - 不要对临时变量取地址操作
   - 不要返回临时变量的地址或临时指针变量
   - 不要申请大的临时变量数组

   内存泄漏(申请存储空间但没有正确释放)（**P131**） 原因：

   - 忘记释放已分配内存
   - 之前工作中A分配内存，B使用后但没人释放
   - malloc分配的内存地址只能由free释放，但如果指针被改变，free将失效

    - 缓冲区溢出损害了被分配内存的头信息

2. 4种堆栈组织形式、作用（**P141**）

4. 动态内存分配

算法需求：(P147)

CH7 数据结构与链表

1. 结构体

```
struct pack
{
  unsigned a:2;
  unsigned b:8;
  unsigned c:6;
} pk1, pk2;
```

2. 联合体

在一个结构(变量)里,结构的各成员顺序排列存储,每个成员都有自己独立的存储位置 联合变量的所有成员共享从同一片存储区 因此一个联己独立的存储位置。联合变量的所有成员共享从同一片存储区。因此一个联合变量在每个时刻里只能保存它的某一个成员的值。

联合变量也在可以定义时直接进行初始化,但这个初始化只能对第一个成员做。

```
union data
{
  char n;
  float f;
};
union data u1 = {3};      //只有u1.n被初始化
```

3. 枚举

与给变量指定初始值的形式类似。如果给某个枚举量指定了值,跟随其后的没有指定值的枚举常量也将跟着顺序递增取值,直到下一个有指定值的常量为止。

```
enum color {RED = 1, GREEN, BLUE, WHITE = 11, GREY, BLACK= 15};
```

这时,RED、GREEN,、BLUE 的值将分别是1、2、3,WHITE、GREY的值将分别是11、12,而BLACK 的值是15。

4. 链表

- 单向

```
struct mylink
{
  int a;
  struct mylink *next;
} *Head, ptr;
```

- 双向&双向循环

```
struct mylink
{
  int a;
  struct mylink *next;
  struct mylink *prev;
};
```

双向：

```
struct mylink *p,*p1,*head;
head=p=(struct mylink *)malloc(len);
p->pre=NULL;     //第一个元素无直接前驱
p1->next=NULL;   //最后一个元素没有直接后继
```

双向循环：

```
p1->next=head,head->pre=p1;
```

CH8 中断与设备驱动

1. 设备驱动、BootLoader与BSP（P175）

2. 缓冲区管理（P180）

3. 中断（P182）

1. 分类：硬件中断、软件中断、异常
2. 加快中断处理的方法：（**P185**）
3. 重要性

- 理解处理器对中断的管理以及这其中的堆栈管理对于理解操作系统是至关重要的
- 中断是操作系统的入口，用户访问操作系统提供的服务的唯一途径是依靠中断来实现的
- 实时系统对异步事件的处理，依靠的是中断
- 任务的调度靠的是中断
- 系统调用的实现靠的是中断
- 在有MMU的系统中，虚存的管理也是依靠中断
- 中断是理解操作系统的入口

4. 函数可重入（P194）

1. 重入的原因：

- 中断服务程序中又重新调用了刚才被中断的函数A
- 在一个抢占式多任务的RTOS内核中，中断服务程序激活了一个更高优先级的任务，并且在中断返回时由RTOS内核的调度器将控制权交给了这个高优先级任务，这个任务接着又重新调用了刚才被中断的函数A
- 递归函数的自我递归调用

2. 可重入条件（**P196**）
3. 互斥保护（**P198**）：关中断、禁止做任务切换、利用信号量

CH10 内存访问越界（P245）

1. 堆栈溢出

1. `定义`：由程序中的错误引起的压栈操作超出原来分配的堆栈底部的问题。
2. `原因`：

- 任务的堆栈预留太小
- 任务中开设了大的临时变量
- 过深函数递归消耗了堆栈

2. 堆栈缓冲区溢出（P247）

一些简答

1. 全局数据区、堆与栈的区别

- 堆被称为动态内存，由堆管理器（系统里的大人物，山高皇帝远不用去管它）管理，程序中可以使用malloc函数来（向堆管理器）申请分配堆内存，使用完后使用free函数释放（给堆管理器回收）。堆内存的特点是：在程序运行过程中才申请分配，在程序运行中即释放（因此称为动态内存分配技术）。
- 栈是C语言使用的一种内存自动分配技术（注意是自动，不是动态，这是两个概念），自动指的是栈内存操作不用C程序员干预，而是自动分配自动回收的。C语言中局部变量就分配在栈上，进入函数时局部变量需要的内存自动分配，函数结束退出时局部变量对应的内存自动释放，整个过程中程序员不需要人为干预。

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。
2. `堆区（heap）`一般由程序员分配释放，例如malloc函数分配的数据区， 若程序员不释放，程序结束时可能由OS回收 。
3. `全局区（静态区）`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。程序结束后有系统释放

2. 栈的作用

- 利用堆栈传递函数调用的参数
- 利用堆栈保存函数调用的返回地址（对于中断处理程序还包括程序状态字寄存器）
- 利用堆栈保存在被调函数中需要使用的寄存器的值
- 利用堆栈实现局部变量

3. 内存泄漏的原因

- 忘记释放已分配内存
- 程序员A分配了内存，被程序员B使用，但没人释放
- 由malloc分配的内存地址只能由free释放，如果指针被改变，free将失效
- 缓冲区溢出损害了被分配内存的头信息