指针和动态内存

指针、结构与函数

关于指针、结构使用的细节

函数返回指针

一个返回指针的函数能返回指向谁的指针?

指针可以指向谁

- 本地变量（包括函数参数）
- 全局变量（包括静态本地变量）
- 动态申请的内存
- 指针值就是传入的指针参数

```
int c = 0;

int *f(int a, int *p)
{
        int b = 0;
        int *p1 = &b;
        int *p2 = &a;
        int *p3 = &c;
        int *p4 = (int*)malloc(sizeof(int));
        int *p5 = p;
        return p5;
}
```

w2-1.c

不可以返回指向本地变量和参数的指针

```c
int *f()
{
        int b = 0xcafebabe;
        int *p1 = &b;
        return p1;
}

void g()
{
        int b = 0xdeadbeef;
}

int main()
{
        int *p = f();
        printf("%X\n", *p);
        g();
        printf("%X\n", *p);
}
```

w2-2.c

静态本地变量

静态本地变量是访问受限的全局变量

```
int a = 0;

void f()
{
        static int b=0;
        int c = 0;
        printf("&a=%p\n", &a);
        printf("&b=%p\n", &b);
        printf("&c=%p\n", &c);
}
```

w2-3.c

返回指向全局变量和静态本地变量的指针

- 返回指向全局变量的指针是为了通过函数选择不同的全局变量
- 返回指向静态本地变量的指针是为了让外界能访问内部的变量
  - 返回静态全局变量的用意也是如此
  - 这是不好的做法，因为突破了访问限制
- 通常出现这样的设计都暗示着设计不良
- 理想的做法是
  - 将数据隐藏起来，作为静态的全局变量或本地变量，让外部不能直接访问
  - 将对这些数据的操作封装在函数中，只露出函数给外部访问
- 返回指向这些变量的指针的做法，无异于直接将这些变量暴露给外部，从而形成外部对这些数据的紧耦合

全局变量之觞

“有着 1 万个全局变量的一大坨代码”：Bookout 和 Schwarz 起诉丰田案件“丰田车突然加速导致严重车祸”

http://blog.jobbole.com/100881/

返回动态分配的内存指针

- 是安全的，但是存在设计隐患
- 关键问题是，该由谁、在什么时候释放（free）这个内存?
- 返回动态分配的内存指针就是指望调用这个函数的那方能安排释放内存的代码
- 绝大多数程序员不会深究从一个函数得到的内存是否需要由自己来释放
- 文档中写得再明白也没有用，大多数程序员不看文档
- 这样的函数埋下了内存泄露的隐患

书上的例子

```
char *concat(const char *s1, const char *s2)
{
    char *result;

    result = malloc(strlen(s1)+strlen(s2)+1);
    if ( result == NULL ) {
        printf("Error: malloc failed in concat\n");
        exit(EXIT_FAILURE);
    }
    strcpy(result, s1);
    strcat(result, s2);
    return result;
}
```

这个例子有两处错误，请指出来。

例子的错误

- 在这个函数中malloc的内存应该在哪里free?
- 在这样的底层处理函数中，直接使用printf是不合适的，为什么这个函数执行的环境一定有标准输出可以用呢?
- 同样直接exit也是不合适的，为什么可以这么粗暴地结束程序呢?
- 在这样底层的函数中出现的异常情况，有必要判断，但是不应该在这个函数里处理

返回指针的最佳设计

- 返回传入的指针
- 让调用函数的那方来决定存储
  - 传入指向本地变量的指针
  - 传入动态分配的内存的地址
- 让动态内存的申请和释放能出现在一个函数中

那为什么还需要返回指针

为了能chain起来，把函数的返回放在另一个表达式中计算：

```
int *f(int *p)
{
        *p = *p + *p;
        return p;
}

int main()
{
        int *p = (int*)malloc(sizeof(int));
        *p = 10;
        printf("%d\n", *f(f(f(p))));
        free(p);
}
```

w2-4.c

传入函数的数组vs结构

- 数组作为参数传入时传入的是地址（指针）
- 结构作为参数传入时传入的是值
    - 新建了结构并拷贝了值

```c
typedef struct {
        int x;
        int y;
} point;

void f(int a[], point pt)
{
        printf("In f()\n");
        printf("sizeof(a) =%2lu,  &a=%p\n", sizeof(a), a)
        printf("sizeof(pt)=%2lu, &pt=%p\n", sizeof(pt), &
}

int main()
{
        int a[] = {1,2,3,4,5};
        point pt = {6,7};
        printf("sizeof(a) =%2lu,  &a=%p\n", sizeof(a), a)
        printf("sizeof(pt)=%2lu, &pt=%p\n", sizeof(pt), &
        f(a, pt);
}
```

w2-5.c

函数返回数组?

- 禁止从函数中返回数组:

```
int[] f() // 这里会编译错误
{
        int a[] = {1,2,3,4,5};
        printf("in f():&a= %p\n", a);
        return a;
}
```

- 只能从函数中返回指针

从函数返回的结构

结构从函数中返回的是拷贝（新建的结构）

```
typedef struct {
        int x;
        int y;
} point;

point g()
{
        point pt = {6,7};
        printf("in g():&pt=%p\n", &pt);
        return pt;
}

int main()
{
        point pt = g();
        printf("in main():&pt=%p\n", &pt);
}
```

w2-6.c

传入传出结构本身vs指针

- 每次传入传出结构本身都需要重新分配内存和拷贝值
- 当结构很大时有很大的内存和时间开销
- 传入传出结构的指针更为高效
- 但是传入结构的指针时，函数内部可能不小心修改了结构的值
- 所以如果逻辑上不会修改结构的值时，采用const修饰
    - `void f(const point *pt);`
- 传出结构的指针时要遵循返回指针的一般原则

返回cosnt的结构指针

可以避免返回的结构被用做左值：

```
const point *f(const point* p)
{
        //      do something to *p
        return p;
}

point *g(point* p)
{
        //      do something to *p
        return p;
}

int main()
{
        point pt = {1,2};
        pt = *f(&pt);   //      fine as right-value
        // f(&pt)->x = 6;       //      error
        g(&pt)->x = 6;  //      fine as left-value
}
```

受限指针（1:318）

- `restrict`是程序员告诉编译器这个指针指向的空间不会有其他指针来访问,编译器可以大胆优化。编译器不会来帮程序员检查是不是有其他指针也来访问,程序员应该知道自己在干什么

```
int main(void)
{
        int *restrict p;
        int *restrict q;
        p = malloc(4);
        *p = 1;
        q = p;
        *q = 3; //       违反了restrict
        printf("x:%d\n", *p);
}
```

w2-8.c

- 这段错误的代码没有报错，因为编译器不会限制`*q`做左值

灵活结构成员（1:319）

- 结构中的最后一个元素是未知大小的数组称为灵活结构成员，结构中的灵活数组成员前面必须至少有一个其它成员

```
typedef struct
{
    int len;
    char content[];
}string;
```

- 这样定义的结构，最后一个成员是一个未确定大小的数组
- `string* str = (string*)malloc(sizeof(int)+32);`
- 就实际上为`*str`里的`content`分配了32个字节

变长参数函数

顾名思义，就是函数的参数长度（数量）是可变的。比如 C 语言的 printf 系列的（格式化输入输出等）函数，都是参数可变的。下面是 printf 函数的声明：

```
int printf ( const char * format, ... );
```

可变参数函数声明方式都是类似的。

变长参数函数

- `printf`函数可以有多个参数，参数的个数不确定
- 汇编函数其实没有参数的概念，C函数的参数其实是由函数自己从堆栈中取出来的
- 调用函数的时候可以填入任意数量的参数值，只要被调函数有办法知道究竟有多少个参数值，每个值的类型就可以了
- `printf`的第一个参数一定是格式字符串，就是起到了这个作用
- 如果有其他形式的约定也是可能的，比如
    - 所有的参数都是`int`，最后一个参数的值是`-1`

利用标准库函数实现变长参数函数

```c
#include <stdarg.h>

void std_vararg_func(const char *fmt, ... )
{
    va_list ap;
    va_start(ap, fmt);

    printf("%d\n", va_arg(ap, int));
    printf("%f\n", va_arg(ap, double));
    printf("%s\n", va_arg(ap, char*));

    va_end(ap);
}

int main()
{
    std_vararg_func("%d %f %s\n", 4, 5.4, "hello world");
}
```

w2-9.c

stdarg.h

- `void va_start ( va_list ap, paramN );`
  - 初始化可变参数列表（把函数在 paramN 之后的参数地址放到 ap 中）
  - ap: 可变参数列表地址, paramN: 确定的参数
- `void va_end ( va_list ap );`
  - 关闭初始化列表（将 ap 置空）
- `type va_arg ( va_list ap, type );`
  - 返回下一个参数的值
- va_list：
  - 存储参数的类型信息
- 用 va_start 获取参数列表（的地址）存储到 ap 中，用 va_arg 逐个获取值，最后用 va_arg 将 ap 置空

log库的例子

```
#define LOG_DEBUG       0
#define LOG_INFO        1
#define LOG_PANIC       2

void logSetLevel(int level);
int logGetLevel(void);

void log(int level, const char* fmt, ...);
```

log.h

log库的例子

```
void log(int level, const char* fmt, ...)
{
        static char buf[200];

        if ( level >= _logLevel ) {
                va_list va;
                snprintf(buf,200,"%02d%02d%02d:",HOUR,MIN
                va_start(va, fmt);
                vsnprintf(buf+7 ,200-7, fmt, va);
                va_end(va);
                uartPrintln(UART0,buf);
        }
}
```

log.c

动态内存分配再研究

- calloc
- realloc
- memset

calloc

```
void *calloc(size_t count, size_t size);
```

- The calloc() function contiguously allocates enough space for count objects that are size bytes of memory each and returns a pointer to the allocated memory. The allocated memory is *filled with bytes of value zero*.
- 我们常利用calloc可以填充0的特性

realloc

```
void *realloc(void *ptr, size_t size);
```

The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr. If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory. If ptr is NULL, realloc() is identical to a call to malloc() for size bytes. If size is zero and ptr is not NULL, a new, minimum sized object is allocated and the original object is freed. When extending a region allocated with calloc, realloc does not guarantee that the additional memory is also zero-filled.

memset

在string.h中：

```
void *memset(void *b, int c, size_t len);
```

The memset() function writes len bytes of value c (converted to an unsigned char) to the string b.

常用来给一段内存清零：

```
memset(p, 0, length);
```

维护零件数据库（1:309）

作业

- 实现一个可自动管理内存的字符串库
- 用以下结构来表达一个字符串：

```
typedef struct {
        int length;
    char *content;
} string;
```

- 这里的length表达字符串中的字符数量，同时也就是content所指的内存的大小
- 这里的content指向一块动态分配的内存，其中的每一个字节都是一个字符，并没有C字符串的表示结尾的`'\0'`

mystring.h

字符串函数：初始化和释放

```
/**
        创建（初始化）字符串
        @param s 要被初始化的字符串
        @param source 用来做初始化内容的C字符串
        @return 初始化了的字符串s，其中的内容为source中的字符串
*/
string* str_create(string* s, const char* source);

/**
        释放字符串
        @param s 要释放的字符串
*/
void str_free(string* s);
```

字符串函数：长度

```
/**
        字符串长度
        @param s 要计算长度的字符串
        @return 字符串的长度
*/
unsigned int str_len(const string* s);
```

字符串函数：拷贝和连接

```
/**
        字符串拷贝
        @param dest 目的地字符串
        @param source 来源字符串
        @return 目的地字符串
*/
string* str_copy(string* dest, const string* source);

/**
        字符串连接
        @param dest 目的地字符串
        @param source 来源字符串
        @return 目的地字符串
*/
string* str_concat(string* dest, const string* source);
```

字符串函数：输入输出

```
/**
        在标准输出上输出字符串
        @param s 要输出的字符串
*/
void str_print(const string* s);

/**
        从标准输入读入一行
        @param s 要写入的字符串
        @return 写入的字符串s
*/
string* str_readline(string* s);
```

扩展实验

- 用灵活结构成员代替结构中的指针，实现以上函数