C程序提高

1 内存分区

内存里面分区主要有四大类：

- 代码区
- 全局区、静态区
- 堆
- 栈

1.1 代码区

代码区：主要存储编译后产生的代码。

1.2 栈

栈：主要储存运行代码时产生的形参、实参、临时变量。这里面的变量存活时间很短，当代码运行超过生存的时间后就会自动消失，比如：代码运行出子函数内部后，所有形参、函数内部定义的变量则会消失。

```
char* getString(){
        char str[] = "hello world!\n";      //str是getString函数中定义的存储、在栈里面的char指针变量，
        return str;                         //其指向常量区保存的hello world!\n中首字母地址
}                                           //一旦程序运行到getString函数外部，则其值会改变。

void test2(){
        char* p = NULL;
        p = getString();                    //p得到的地址值有可能已经不指向
                                            //常量区保存的hello world!\n中首字母地址了
        printf("%s\n", p);
}
```

1.3 堆

1.3.1 堆的基本情况与注意事项

堆：用于存储用户自己申请空间的变量，比如用`malloc`申请的变量。这个变量存活时间较长，只有用户自己释放空间（`free`）或者程序停止才会消失。

```
int* getSpace(){
        int* p = malloc(sizeof(int) * 5);
//由于p是malloc申请的int型指针
        if(p == NULL){
//于是其存储在堆中，需要用户手动释放
                printf("falid to malloc space");
                return NULL;
        }
        for(int i = 0; i < 5; ++i){
                p[i] = i + 100;
        }

        return p;
}

void test1(){
        int* p = getSpace();
//getSpace返回的地址变量存储在堆中，需要手动释放
        for(int i = 0; i < 5; ++i){
                printf("%d\n", p[i]);
        }
        if(p != NULL){
                free(p);
//手动释放堆中的指针变量p
        }
}
```

小心函数调用时，形参、实参的传递关系。函数的传递是值传递，也就是拷贝变量值再将其传入函数中。

```
void allocSpace(char* pointer){
        char* temp = malloc(sizeof(char) * 100);
        memset(temp, 0, 100);
        strcpy(temp, "hello world!\n");
        pointer = temp;
}

void test2(){
        char * p = NULL;
        allocSpace(p);
        printf("%s\n", p);
}
```

下面是正确方法一：

```
void allocSpace2(char** pointer){
        char* temp = malloc(sizeof(char) * 100);
        memset(temp, 0, 100);
        strcpy(temp, "hello world!\n");

```

```
        *pointer = temp;
}

void test3(){
        char * p = NULL;
        allocSpace2(&p);                    //主调函数没有分配内存，被调函数需要高级指针修饰低级指针

        printf("%s\n", p);
}
```

下面是正确方法二：

```
char* allocSpace3(){
        char* temp = malloc(sizeof(char) * 100);
        memset(temp, 0, 100);
        strcpy(temp, "hello world!\n");

        return temp;
}

void test4(){
        char * p = NULL;
        p = allocSpace2();                  //也可以使用函数返回值修饰主调函数中的空指针

        printf("%s\n", p);
}
```

1.3.2 内存申请函数

- `malloc`函数

  用法：`void *malloc(size_t size)`

  作用：申请一块空间为`size`大小的空间，并且返回空间头指针。

  说明：并没有给申请的空间赋初始值，所以需要赋初始值。

- `calloc`函数

  用法：`void* calloc（unsigned int num，unsigned int size）`

  作用：申请一块单位空间为`size`数量为`num`的空间，并且返回空间头指针。

  说明：给申请的空间赋初始值`0`。

- `realloc`函数

  用法：`void *realloc(void *mem_address, unsigned int newsize)`

  作用：重新分配空间，将`mem_address`指向的空间改变为`newsize`大小，并返回一个指向新空间头地址的指针。

说明：重新分配空间有两种方法，使用哪一种方法与内存余下的空间是否满足需求相关。如果余下的空间足够，那么就在利用内存余下的空间分配；如果余下的空间不够，那么就会在内存中寻找并使用一块足够的空间。由此可见，如果内存足够那么`mem_address`与返回的指针一致，否则不一致。

- `free`函数

  用法：`void free(void *ptr)`

  作用：释放由`malloc`、`calloc`、`realloc`申请的空间。

  说明：如果不使用`free`函数释放空间，那么程序运行结束后空间会自动释放。

1.4 全局区、静态区

全局区、静态区主要存储三种数据：

- 全局变量
- 静态变量
- 常量

该区域的数据会一直保存到程序停止运行，其还有如下的性质：

- 程序运行期间其一直存在。
- 已初始化的数据处在`data`段，没有的放到`bss`段。
- 如果没有初始化，则会有默认的初始值。

1.4.1 全局变量

全局变量是定义在任何函数之外（包括main函数）的变量，其将会在整个程序周期内存活。如果没有对其赋初值，那么其默认初始值为`0`。下面两种语法都是定义了同样的全局变量。

```
int a = 0;
extern int a = 0;
```

如果需要跨文件调用，需要在使用变量的文件中再次声明其为外部变量，需要有如下的形式：

```
// test.c文件
int g_a;              //未初始化的全局变量，编译器赋值其为0
```

```
// 需要使用g_a变量的文件
void test1(){
    extern int g_a;
    printf("g_a = %d\n", g_a);
}
```

1.4.2 静态变量

静态变量类似于全局变量，但其只能在本文件中使用，不可跨文件使用。下面的代码与运行结果可以说明静态变量初始化只有一次。

```
void fun(){
    static int a;   //未初始化的静态变量，编译器赋值其为0
    a++;
    printf("%d\n", a);
}

void test2(){
    fun();          //运行结果：1
    fun();          //运行结果：2
    fun();          //运行结果：3
}
```

1.4.3 常量

1.4.3.1 `const`修饰的全局变量

`const`修饰的全局变量不可直接修改（对其值直接修改）或者间接修改（利用指针修改），具体例子如下。

```
const int c_a = 10;
void test3(){
    c_a = 20;           //直接修改失败

    int* p = &c_a;
    *p = 20;            //间接修改失败
}
```

`const`修饰的全局变量存储在常量区，一旦有任何修改其值的表现，程序就会崩溃。

1.4.3.2 `const`修饰的局部变量

`const`修饰的局部变量不可以直接修改，但是可以间接修改。

```
void test4(){
    const int c_b = 10;

    c_b = 20;           //直接修改失败

    int* p = &c_b;
    *p = 20;            //间接修改成功
}
```

由于局部变量存储在栈区，所以间接修改可以成功，但是由于变量是被`const`修饰的，所以不可以直接修改。

1.4.3.3 字符串常量

用字符串指针接受字符串常量，那么就不可以改变其值；如果是字符串数组接受字符串常量，那么就可以改变其值。下面是具体代码示例

```
void test5(){
    char* p1 = "hello world\n";
    printf("%c\n", p1[0]);
    p1[0] = 'x';                //字符串指针接受字符串常量，不可以改变其值
    printf("%c\n", p1[0]);

    char p2[] = "hello world\n";
    printf("%c\n", p2[0]);
    p2[0] = 'x';                //字符串数组接受字符串常量，可以改变其值
    printf("%c\n", p2[0]);
}
```

字符串指针指向字符串常量的首地址，而字符串常量存储在常量区不可被修改；字符串数组接受字符串常量，是将字符串常量复制一次后将其赋值给字符串数组。

1.5 函数调用模型

1.5.1 宏函数

宏定义可以定义函数，并且使用宏函数将会节省普通函数出入栈的时间成本，但建议所定义的函数较为简短。由于宏定义是在预处理阶段将代码进行替换，所以在使用宏函数时一定注意运算符的优先性，下面则是具体案例的代码。

```
#define myADD(x,y) ((x)+(y))

int myAdd(int a, int b){
    return myAdd(a, b) * 2;
}
```

注意，写宏定义函数时不要键入多余的空格键，否则编译将会出错。

1.5.2 函数调用流程

先给出函数具体案例

```
int func(int a, int b){
    int ta = a;
    int tb = b;
    return ta + tb;
}

int main(){
    int ret = 0;
    ret = func(10, 20);
    return 0;
}
```

由于所有局部变量均储存与栈中，于是下面先开辟一个栈空间。首先将变量`ret`压入栈，接着遇见`fun`函数，于是记录当前为位置为函数的返回地址。进入函数时首先需要将函数形参压入栈中，但是入栈顺序是一个需要进一步讨论的问题（这里假设`a`先`b`后），接着将`ta`、`tb`分别压入栈。计算完`ta+tb`后，由于结果小于四个字节，于是将结果保存在寄存器中。什么时候将函数中的局部变量清除是另一个需要进一步讨论的问题（是在离开函数时清除，还是回到主函数后清除）。将计算结果返回给`ret`后，主函数停止运行`ret`变量清除。

 函数调用时栈流程图

1.5.3 调用惯例

为了处理函数形参入栈的顺序以及被调函数中变量释放时间等问题，人们认为主调函数与被调函数调用时须有一致的约定，这个约定称为调用惯例。下表是常见的调用关系表。

调用惯例	形参入栈顺序	被调函数中变量释放者
`cdecl`	从右到左	主调函数
`stdcall`	从右到左	被调函数
`fastcall`	前两个由寄存器传递，余下从右到左	被调函数
`pascal`	从左到右	被调函数

1.5.4 栈的生长方向

搞清楚栈的生长方向，也就是搞清楚局部变量是按照地址高位到低位排列还是反之。下面代码的运行结果也就说明了结论。

```c
void test1(){
    int a = 10;
    int b = 10;
    int c = 10;
    int d = 10;

    printf("%d\n",&a);
    printf("%d\n",&b);
    printf("%d\n",&c);
    printf("%d\n",&d);
}
```

其运行结果为

```
2129943832          //a
2129943836          //b
2129943840          //c
2129943844          //d
```

由代码运行的结果来看，局部变量是按照地址从高到低存储的，也就是变量`a`处于栈底，是高地址；变量`d`栈顶，是低地址。

1.5.5 内存存储方式

c语言中低位数据存放在低地址，高位数据存放在高地址，下面是利用十六进制数字11223344举例说明。

```c
void test2(){
    int a = 0X11223344; //一个（比如1，1，2，2，3，3，4，4）十六进制数字可以用四个二进制表示，而8个二进制数字为一个字节
    char* p = &a;       //字符串指针指针步长为1，所以可以做到每一次只会往前移动一个字节

    printf("%x\n", *p);             //44    低位数据低地址
    printf("%x\n", *(p+1));         //33
    printf("%x\n", *(p+2));         //22
    printf("%x\n", *(p+3));         //11    高位数据高地址
}
```

代码运行结果为

```
44              //*p
33              //*(p+1)
22              //*(p+2)
11              //*(p+3)
```

也就是在内存中由低地址到高地址数字依次为44、33、22、11。

2 指针进阶

2.1 空指针与野指针

2.1.1 空指针

空指针是不指向任何东西的指针，不可对其进行解引用操作，否则将会直接报错。建议在初始化指针变量时，将其赋值为`NULL`——空指针。

2.1.2 野指针

野指针指向已经释放的内存空间或者是没有申请的内存空间。同样的，不可对其进行解引用操作，否则程序崩溃。下面的案例说明了不可对空指针和野指针进行操作。

```
void test(){
    char* p = NULL;
    strcpy(p, "1111");      //不要操作空指针

    char* q = 0X1122;
    strcpy(q, "1111");      //不要操作野指针
}
```

其中`strcpy`是往指定内存中写入字符串，第一个参数为需要写入的首地址指针，第二个是需要写入的字符串常量。

2.1.3 常见的野指针

常见的野指针有如下的三类，在使用的过程中应当注意：

- 指针变量未初始化

  创建指针后应该尽量初始化。

  ```
  int* p;
  ```

- 释放堆区空间后指针未置空

  在释放堆区空间后指针可能随机指向某一个内存空间，此时在对其解引用可能会导致程序崩溃。所以在堆区空间释放后指针应该赋值为`NULL`。

```c
int* p = malloc(sizeof(int));
*p = 1000;
printf("%d\n", *p);

free(p);
//printf("%d\n", *p);    错误使用方法，可能导致程序崩溃
p = NULL;
```

注意空指针可以`free`，而野指针不可以`free`。

- 指针超过变量作用域

  例如函数的返回值为局部指针变量，那么这样返回得到的指针就是野指针，使用它就会导致程序出错甚至崩溃。

```c
int* func(){
    int a = 10;
    int* p = &a;

    return p;
}

int main(){
    int* p = func();
    printf("%d\n", *p);

    return 0;
}
```

2.2 指针步长

不同类型的指针有如下的不同点：

- 指针变量+1后在内存中跳跃的字节数目不同
- 解引用的时候，取出的字节数不同

下面的代码案例将会解释以上的不同点。第一个案例说明了跳跃的字节数目。

```c
void test(){
    char* p = NULL;
    printf("%d\n", p);              //假设结果为0
    printf("%d\n", p+1);            //那么此处的结果为1

    int* p1 = NULL;
    printf("%d\n", p1);             //假设结果为0
    printf("%d\n", p1+1);           //那么此处的结果为4
}
```

第二个案例说明了解引用的时候，取出的字节数不同。下图是图解说明。其中`memcpy`函数的参数有三个，第一个是目标地址指针`*diestin`，第二个是源地址指针`*source`，第三是拷贝字节大小`num`。其作用为从源地址`*source`开始拷贝指定字节大小`num`的内存空间到目标地址`*diestin`。

```
void test2(){
    char buf[1024] = {0};
    int a  = 1000;
#if(test == 1)
    memcpy(buf, &a,  sizeof(int));

    char* p = buf;
    printf("%d\n", *p);     //结果为-24，不知道是什么结果。
    printf("%d\n", *(int*)p1);  //结果为1000
#else
    memcpy(buf+1, &a,  sizeof(int));

    char* p = buf;
    printf("%d\n", *(p+1));     //结果为-24，不知道是什么结果。
    printf("%d\n", *(int*)(p1+1));  //结果为1000
#endif
}
```

2.2.1 指针步长练习

下面我们定义一个结构体以练习指针步长：

```
struct Person
{
    char a;         //0 - 3
    int b;          //4 - 7（内存对齐要求int必须放在4的倍数上）
    char buf[64];   //8 - 71
    int d;          //72 - 75
}

int main(){
    struct Person p = {'a', 10, "hello", 1000};

    printf("d is %d\n", *(int*)((char*)&p + 72));
    return 0;
}
```

需要注意到的是，`char*`的步长为1，所以后面需要加上内存偏移量为72，再用`int*`强制转化指针类型，最后解引用。

2.3 指针的间接赋值

变量有两种修改方法，分别是：直接修改、间接修改。通过指针可以做到间接赋值，其成立的条件为：

- 两个变量（普通变量+指针变量）or （实参+形参）
- 建立关系
- 通过*操作指针指向的内存

```c
void changeNum(int* p){//两个变量——实参+形参（int* p = &a）并且建立关系
    *p = 200;
}

void test(){
    //两个变量（普通变量+指针变量）/ （实参+形参）
    int a = 10;
    int* p = NULL;
    //建立关系
    p = &a;
    //通过*操作指针指向的内存
    *p = 20;
    printf("%d\n", a);

    int a1 = 10;
    changeNum(&a);
    printf("%d\n", a2);
}
```

2.4 指针做函数参数输入输出特性

输入特性：主调函数分配内存、被调函数使用内存

```c
void func1(char* p){
    strcpy(p, "abc");
}

void func2(char* p){
    printf("%s\n", buf);
}

void test(){
    //栈上面分配内存
    char buf[1024] = {0};

    func1(buf);
    printf("%s\n", buf);

    //堆上面分配内存
    char* p = malloc(sizeof(char) * 64);
    func2(p);
}
```

输出特性：被调函数分配内存、主调函数使用内存。其中`memset`函数有三个传入参数，第一个是指向要填充的内存块首地址指针`*str`。第二个是要被设置的值`c`；该值以`int`形式传递，但是函数在填充内存块时是使用该值的无符号字符形式。第三个是要被设置为该值的字符数`n`。其作用为复制字符`c`（一个无符号字符）到参数`str`所指向的字符串的前`n`个字符。

```
void allocSpace(char** pp){
    char* temp = malloc(sizeof(char) * 64);
    memset(temp, 0, 64);
    strcpy(temp, "hello!");

    *pp = temp;
}

void test(){
    char* p = NULL;
    allocSpace(&p);                    //主调函数没有分配内存，被调函数需要高级指针修饰低级指针
    printf("%s\n", p);
}
```

2.5 指针易错点

2.5.1 指针越界

```
void test(){
    char buf[8] = "12345678";    //有八个字符串，但是字符串数组也只有8个，没有办法存放'\0'结束标志
    printf("%s\n", buf);
}
```

2.5.2 返回局部变量地址

```
char* getString(){
    char str[] = "abc";
    printf("%s\n",str);
    return str;          //返回处于栈区的局部指针变量，也就是返回了一个野指针，有可能导致程序错误甚至崩溃
}

void test(){
    char* str = getString();
    printf("%s\n", str);
}
```

2.5.3 同一块内存多次释放

```c
void test(){
    char* p = malloc(64);
    free(p);            //free(p)之后没有置空p，那么p成为野指针
    free(p);            //不可对野指针free，空指针可以free
}
```

2.5.4 释放偏移后的指针

```c
void test(){
    char str[] = "hello";
    char* p = malloc(p);
    for(int i = 0; i <= strlen(str); ++i){
        *p = str[i];
        ++p;
    }
    free(p);        //p已经不是当时申请空间的首地址了，这样释放空间会出问题
}
```

2.6 二级指针输入输出特性

2.6.1 二级指针输入特性

主调函数分配内存，被调函数使用内存，这和普通指针一致。

```c
void printArray(int** arr, int len){
    for(int i = 0; i < len; ++i){
        printf("%d\n", *arr[i]);
    }
}

void test1(){
    //堆区分配内存
    int** pArray = malloc(sizeof(int *) * 5);

    int a1 = 100;
    int a2 = 100;
    int a3 = 100;
    int a4 = 100;
    int a5 = 100;

    pArray[0] = &a1;
    pArray[1] = &a2;
    pArray[2] = &a3;
    pArray[3] = &a4;
    pArray[4] = &a5;

    printArray(pArray, 5);
```

```
}

void test2(){
    //栈区分配内存
    int* arr[5];

    for(int i = 0; i < 5; ++i){
        arr[i] = malloc(sizeof(int));
        *arr[i] = 100 + i;
    }

    printArray(arr, 5);
}
```

2.6.2 二级指针输出特性

被调函数分配内存，主调函数使用内存，这和普通指针也是一致的。

```
void allocateSpace(int** p){
    int* arr = malloc(sizeof(int) * 10);
    for(int i = 0; i < 10; ++i){
        arr[i] = i;
    }
    *p = arr;
}

void printArray(int** p, int len){
    for(int i = 0; i < len; ++i){
        printf("%d\n", (*p)[i]);
    }
}

void freeSpace(int** p){
    if((*p) != NULL){
        free(*p);
        *p = NULL;
    }
}

void freeSpace_wrong(int* p){
    if((p) != NULL){
        free(p);
        p = NULL;
    }
}

void test1(){
    int *p = NULL;
    allocateSpace(&p);                    //主调函数没有分配空间，被调函数只能用更高级的指针修饰

```

```
    printArray(&p, 10);
#if 0
    freeSpace_wrong(&p);                    //如果执行此代码后p为野指针
#else
    freeSpace(&p);                          //此时p为空指针
#endif
}
```

3 位运算

4 数组进阶

4.1 一维数组与数组名

数组是在一片连续的内存空间中，存放相同数据类型的数据元素。在数组知识中，最不解的就是一维数组名是不是**指向数组第一个元素的指针**？答案是模糊的，可以是，也可以是不是。下面用具体的代码案例说明什么情况下一维数组的名不等价于指向一维数组第一个元素的指针。

- 对数组名取`sizeof`，获取的是整个一维数组的大小

```
void test(){
    int arr[5] = {1,2,3,4,5};
    printf("size of arr is %d\n", sizeof(arr));
}
```

- 对一维数组名取地址后并不是一个二级指针`int**`，其指针步长为整个数组的内存长度（在上面的案例中内存长度为20）

```
void test(){
    int arr[5] = {1,2,3,4,5};
    printf("%d\n", &arr);
    printf("%d\n", &(arr+1));
}
```

一维数组名`arr`是一个指针常量——也就是类似于`int* cont p`的指针（注意`const int* p`为常量指针，其可以改变`p`指向的空间，但不可以改变其中的值），其不能改变`arr`指向的空间，但可以改变其中的值。

接下来我们需要讨论的问题是：访问一维数组中的元素时，其指标可不可以是负数。在不越界的情况下答案是可以的，下面代码可以说明。

```
void test(){
    int arr[5] = {1,2,3,4,5};
    int* p = arr;

    p = p + 3;
    printf("%d\n", p[-1]);      //result:3
    printf("%d\n", *(p-1));     //result:3
    printf("%d\n", -1[p]);      //result:3
}
```

上面代码也说明了以下三种写法是等价的：

- `p[i]`
- `*(p + i)`
- `i[p]`，但是这种写法及其不推荐。

下文是讨论的是将一维数组作为参数传入等价的两种写法

```
void printArray(int* arr, int len);          //可读性强
void printArray(int arr[], int len);         //可读性弱
```

4.2 数组指针定义方法

定义一个数组指针用于接受一维数组名取地址的值，也就是`&arr`，有三种方法

- 先定义数组的类型，再通过数组类型定义数组指针变量

```
void test(){
    arr[5] = {1,2,3,4,5};

    typedef int(ARRAY_TYPE)[5];
    ARRAY_TYPE* arrP = &arr;
//*arrP 等价于 arr

    for(int i = 0; i < 5; ++i){
        printf("%d ",(*arrP)[i]);
    }
    printf("\n");
}
```

- 先定义数组指针的类型，再通过数组指针类型定义数组指针变量

```
void test(){
    arr[5] = {1,2,3,4,5};

    typedef int(*ARRAY_TYPE)[5];
    ARRAY_TYPE arrP = &arr;
//*arrP 等价于 arr

    for(int i = 0; i < 5; ++i){
        printf("%d ",(*arrP)[i]);
    }
    printf("\n");
}
```

- 直接定义数组指针变量

```
void test(){
    arr[5] = {1,2,3,4,5};

    int (*arrP)[5] = &arr;
//*arrP 等价于 arr

    for(int i = 0; i < 5; ++i){
        printf("%d ",(*arrP)[i]);
    }
    printf("\n");
}
```

我们这里进一步的总结对于数组与指针相互的关系，目前我知道的有三种值得总结：

1. `int* p`。这个指针变量可以指向`int`类型变量——赋值方法为

```
int a = 0;
int* p = &a;
```

其也可以指向`int`类型数组的首地址——赋值方法为

```
int arr[5] = {1,2,3,4,5};
int* p = arr;
```

由于`int`类型的指针变量指针步长为`sizeof(int)=4`字节，而数组在内存中是连续存储的，所以让指针变量自增加是可以取到每一个数组元素的。

2. `int (*p)[m]`。同样的这个指针变量也可以指向两种不同的类型，第一种是上文种提及的，第二种是下一个部分要讲的内容——指向二维数组名，赋值方法为

```
int arr[2][2] = {1,2,3,4};
int (*p)[2] = arr;
```

3. `int* p[m]`。这种类型的指针变量是指针数组，其种有`m`个数据，每一个数据是指向`int`类型数据的指针。以下是具体案例

```
int a = 100;
int b = 100;

int* p[2];
p[1] = &a;
p[2] = &b;
for(int i = 0; i < 2; ++i){
    printf("%d ", *p[i]);
}
printf("\n");
```

4.3 二维数组与数组名

首先定义二维数组方法有三种，下面的案例具体的说明了定义方法

```
int arr1[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
int arr2[3][3] = {1,2,3,4,5,6,7,8,9};
int arr[][3] = {1,2,3,4,5,6,7,8,9};
```

二维数组名与一维数组名情况类似——二维数组名除了两种情况以外，其就可以理解为一个**指向第一个一维数组地址的数组指针**`(int (*p)[m])`，具体情况参见下面代码

```
int (*p)[3] = arr;

printf("%d\n", arr[1][2]);
printf("%d\n", *(*(p+1)+2));
//p指向第一个一维数组地址，其为数组指针
//其类似于一维数组名取地址得到的值，指针步长为一维数组字节数
//其解引用后为指向int类型数据的指针，此时的指针步长变为int的字节数4
//再解引用一次就是具体的数值
printf("%d\n", *(*p+5));
//内存中二维数组也是按顺序排列的，*p的步长为4，所以(1,2)处的元素偏移量
//为5*4字节，所以再*p之后加上5，最后再次解引用得到具体数值
```

下面是第一个例外——对二维数组名取`sizeof`操作，得到的是整个二维数组的字节大小

```
arr[3][3] = {1,2,3,4,5,6,7,8,9};
printf("%d\n", sizeof(arr));
```

第二个例外——对二维数组取地址操作，其指针步长为整个二维数组的字节大小

```
int (*p)[3][3] = &arr;
printf("%d\n", p);
printf("%d\n", p+1);
```

至于如何将二维数组传入函数中，下面有三种方法可以使用

- ```
  void printArray(int (*arr)[3], int row, int col);
  ```
- ```
  void printArray(int arr[3][3], int row, int col);
  ```
- ```
  void printArray(int arr[][3], int row, int col);
  ```
  ```

其中行数和列数可以这样获取

```
int row = sizeof(arr) / sizeof(arr[0]);
int col = sizeof(arr[0]) / sizeof(int);
```

4.4 数组与指针步长

在数组与指针中较为让人不解的是如下有关二维数组的三个地址是一样的，那么它们有什么区别吗

```
printf("%d\n", &arr[0][0]);
printf("%d\n", &arr);
printf("%d\n", arr);
```

它们的区别在于其指针步长不一样，`&arr`的指针步长为整个数组字节数，`arr`的指针步长为数组列的字节数，`&arr[0][0]`为`int`的字节数4。同样的对于一维数组来说，`arr`的指针步长为整个数组的字节数，`&arr[0]`为`int`的字节数4，但是它们指向同一个地址。值得提醒的是以下的几种写法是等价的，可以相互替换

```
{
    int arr[3] = {1,2,3};
    int* p = arr;                   //p    等价   arr
    int (*p)[5] = &arr;             //*p   等价   arr
}

{
    int arr[][3] = {1,2,3,4,5,6,7,8,9};
    int (*p)[3] = arr;              //p    等价
arr
    int (*p)[3][3] = &arr;          //*p   等价
arr
}
```

所以上面每一个p对应的指针步长就很容易推断出来了。

5 结构体进阶

5.1 结构体基本使用

结构体的作用：内置的数据类型不够用时，我们可以通过结构体创建新的数据类型。

结构体中不可包含函数并且结构体定义中不可以赋初始值。下面时结构体的基本使用：

1. 结构体的定义

```c
struct Person{
    //c语言中不可以赋初始值，不可以包含函数
    char name[64];
    int age;
};
```

2. 结构体起别名

```c
typedef struct Person{
    char name[64];
    int age;
}myPerson;
//myPerson为结构体别名

//or

struct Person{
    char name[64];
    int age;
};
typedef struct Person myPreson;
```

如果前面没有加上typedef那么就不是起别名，而是在定义的同时创建一个结构体变量，并且可以给这个结构体变量赋初始值，示例见如下的代码：

```c
struct Person{
    char name[64];
    int age;
}myPerson = {"name", 18};
//myPerson为结构体变量

void test(){
    printf("name:%s,age:%d",
myPerson.name,MmyPerson.age);
    myPerson.name = "mk";myPerson.age =
19;
    printf("name:%s,age:%d",
myPerson.name,MmyPerson.age);
}
```

3. 匿名结构体。匿名结构体就是只可以创建一次的结构体变量的结构体，创建变量的时间就是定义结构体的时间。下面的代码就是具体的示例。

```c
struct {
    char name[64];
    int age;
}myPerson;
//myPerson为匿名结构体变量
```

下面是分别在栈和堆分别创建结构体变量以及数组

```c
struct Person{
    char name[64];
    int age;
};

void test(){
    //栈区创建结构体
    struct Person myPerson = {"aaa", 1};
    printf("name:%s,age:%d",
myPerson.name,MmyPerson.age);

    //堆区创建结构体
    struct Person* mk = (struct
Person*)malloc(sizeof(struct Person));
    mk->name = "mk";mk->age = 24;
    printf("name:%s,age:%d", mk.name, mk.age);
    if(mk != NULL){
        free(mk);
        mk = NULL;
    }

    //栈区创建结构体数组
    struct Person personArray[3] = {{"aaa", 1},{"bbb",
2},{"ccc", 3}};
```

```
    int len = sizeof(personArray) / sizeof(struct Person);
    for(int i = 0; i < len; ++i){
        printf("name:%s,age:%d", personArray[i].name, personArray[i].age);
    }

    //堆区创建结构体数组
    struct Person* arr = (struct Person*)malloc(sizeof(struct Person) * 3);
    int len_2 = sizeof(arr) / sizeof(struct Person);
    for(int i = 0; i < len_2, ++i){
        strcpy(arr[i].name, "Tom");
        arr[i].age = i;
    }
    for(int i = 0; i < len_2; ++i){
        printf("name:%s,age:%d", arr[i].name, arr[i].age);
    }
    free(arr);
    arr = NULL;
}
```

5.2 结构体赋值问题及其解决

结构体赋值问题主要是浅拷贝（也就是**逐字节拷贝**）的问题，下面给出一段错误代码，运行代码会出错：

```
struct Person{
    char *name;
    int age;
}myPerson;

void test(){
    myPerson p1;
    p1.name = (char*)malloc(sizeof(char) * 64);
    strcpy(p1.name, "Tom");
    p1.age = 18;
    printf("name:%s,age:%d", p1.name, p1.age);

    p2.name = (char*)malloc(sizeof(char) * 128);
    strcpy(p2.name, "Jerry");
    p2.age = 19;
    printf("name:%s,age:%d", p2.name, p2.age);

    p1 = p2;
    if(p1.name != NULL){
        free(p1.name);
        p1.name = NULL;
    }
    if(p2.name != NULL){
```

```
        free(p2.name);                                  //此时p2为野指针，所以对其进行free那就就会报错
        p2.name = NULL;
    }
}
```

下图是上面代码运行到释放空间之前内存的结构与分布变化图。

可以看见浅拷贝只是将`p2`指向的内容全部复制给`p1`指向的内容，这样会导致两个问题：第一、`p1`原本指向的空间没有被释放掉，这会导致内存泄漏；第二、如果对`p1.name`释放后再次释放`p2.name`那么会导致——野指针被释放程序奔溃。解决浅拷贝问题就需要手动写出深拷贝代码，下面代码是正确代码。

```
struct Person{
    char *name;
    int age;
}myPerson;

void test(){
    myPerson p1;
    p1.name = (char*)malloc(sizeof(char) * 64);
    strcpy(p1.name, "Tom");
    p1.age = 18;
    printf("name:%s,age:%d", p1.name, p1.age);

    p2.name = (char*)malloc(sizeof(char) * 128);
    strcpy(p2.name, "Jerry");
    p2.age = 19;
    printf("name:%s,age:%d", p2.name, p2.age);

    //p1 = p2;
    if(p1.name != NULL){
        free(p1.name);
        p1.name = NULL;
    }
    p1.name = (char*)malloc(strlen(p2.name) + 1);
    strcpy(p1.name, p2.name);
    p1.age = p2.age;


    if(p1.name != NULL){
        free(p1.name);
        p1.name = NULL;
    }
    if(p2.name != NULL){
        free(p2.name);
        p2.name = NULL;
    }
}
```

下图是上面代码运行到释放空间之前内存的结构与分布变化图。浅拷贝出现问题的另一个原因是，结构体中定义name是通过堆进行定义的，如果是在栈上进行定义这不会出现问题，这是根本原因。

5.3 结构体偏移量

下面我们构造一个结构体，其中包含一个字符与整数，其中字符占用一个字节，而整数占用四个字节。

```
struct Person{
    char name;
    int age;
};
```

假设地址定义的结构体变量`p`起始地址为0，那么`p.name`就会放在第0地址上；由于`p.age`需要放在4的整数倍上，所以其数据放在第4，5，6，7地址上；第1，2，3地址上不存放任何变量；

下面是利用`stddef.h`中`offsetof`函数计算偏移量的代码示例：

```
struct Person{
    char name;
    int age;
};

void test(){
    struct Person p;

    int age_offset_1 = offsetof(struct Person, age);
    int age_offset_2 = (int)(&p.age) - (int)(&p);
//利用地址转化为整数再求差得到偏移量
    printf("offset of p.age in way a:%d,offset of p.age in way b:%d\n",age_offset_1,age_offset_2);
}
```

下面代码是利用偏移量得到结构体中的数据

```c
struct Person{
    char name;
    int age;
};

void test(){
    struct Person p = {'a', 10};

    int age_offset = offsetof(struct Person, age);
    printf("get p.age in way a:%d\n", *(int*)((char*)&p + age_offset) );
    printf("get p.age in way b:%d\n", *(int*)((int*)&p + age_offset / sizeof(int)) );
    printf("get p.age in way c:%d\n", p.age);
}
```

下面考虑结构体嵌套，尝试使用地址偏移获取到变量值

```c
struct Person{
    char name;
    int age;
};

struct Person_father{
    char name;
    int age;
    struct Person son;
};

void test(){
    struct Person_father mk = {'a', 10, {'b', 2}};

    int son_offset = offsetof(struct Person_father, son);
    int age_offset = offsetof(struct Person, age);
    printf("get mk.son.age in way a:%d\n", *(int*)((char*)&p + son_offset + age_offset) );
    printf("get mk.son.age in way b:%d\n", (struct Person*)((char*)&p + son_offset)->age );
    printf("get mk.son.age in way c:%d\n", mk.son.age);
}
```

5.4 内存对齐

5.4.1 内存对齐的意义

内存中最小的单元是字节。CPU读取内存的时候是按照区块读取的，这个块的大小可能是2，4，8，16字节。如果内存中的所有数据都是密集排列，那么就会导致一个变量数据需要二次访问，示意图如下。有了内存对齐，虽然浪费了一定的空间，但是提升了操作系统访问内存的效率。

对于内置数据类型，数据只需放在该类型的整数倍上即可；对于自定义的数据类型有一套规则，这就是下面的内容。

5.4.2 内存对齐的计算方法

内存对齐主要有如下的三条规则

1. 第一个属性从偏移量零开始储存。
2. 第二个属性开始，放在min(该数据类型大小，对齐模数)的整数倍上。
3. 整体计算完成后需要做二次对齐——结构体大小必须是min(该结构体最大的数据类型，对齐模数)的整数倍，不足需要补足对齐。

一般的对齐模数为8（可以用`#pragma pack(show)`显示对齐模数，用`#pragma pack(2^n)`修改齐模数——只能是2的幂）。接下来运用具体的代码说明上述规则：

```
struct Student{
    int a;          //0 - 3
    char b;         //4 - 7     min(1,8) = 1
    double c;       //8 - 15    min(8,8) = 8
    float b;        //16 - 19   min(4,8) = 4
};                  //0 - 23    min(8,8) = 8

void test(){
    printf("%d\n", sizeof(struct Student));
}
```

如果遇见结构体嵌套，那么嵌套进去的结构体需要放在：min(该结构体最大的数据类型，对齐模数)的整数倍上；并且考虑被嵌套的结构体的大小时，只考虑嵌套进去的结构体中最大的数据类型，例如

```
struct Student{
    int a;                      //0 - 3
    char b;                     //4 - 7     min(1,8) = 1
    double c;                   //8 - 15    min(8,8) = 8
    float b;                    //16 - 19   min(4,8) = 4
};                              //0 - 23    min(8,8) = 8

struct Teacher{
    int a;                      //0 - 7
```

```
    struct Student b;           //8 - 31    min(8,8) = 8
    double c;                   //32 - 39   min(8,8) = 8
};                              //0 - 40    min(8,8) = 8

void test(){
    printf("%d\n", sizeof(struct Teacher));
}
```

6 文件

7 函数指针

7.1 函数指针的定义

7.1.1 函数指针的定义方法

函数的名称也就是一个地址，也就是函数名称也就是一个函数指针，其指向函数的入口地址，下面的案例就能说明这个问题：

```
int func(int a, int b){
    return a + b;
}

void test(){
    printf("%d\n", func);
}
```

下面考虑函数指针有几种命名方法。类似于数组指针，函数指针也是三种定义方法。

1. 先定义函数类型，再通过函数类型定义函数指针

```
int func(int a, int b){
    return a + b;
}

void test(){
    typedef int (FUNC_TYPE)(int, int);
    FUNC_TYPE* fun = func;
    printf("%d\n", fun(1,2));
}
```

2. 先定义函数指针类型，再通过函数指针类型定义函数指针

```
int func(int a, int b){
    return a + b;
}

void test(){
    typedef int (FUNC_TYPE*)(int, int);
    FUNC_TYPE fun = func;
    printf("%d\n", fun(1,2));
}
```

3. 直接函数指针类型

```
int func(int a, int b){
    return a + b;
}

void test(){
    int (*fun)(int, int) = func;
    printf("%d\n", fun(1,2));
}
```

函数指针与指针函数的区别：

- 函数指针是指向函数入口的地址，如`int (*p)(int, int)`。
- 而指针函数是一个返回指针的函数，如`int* p(int a, int b)`。

7.1.2 函数指针数组的定义方法

函数指针数组是一个数组，数组中每一个元素都是存储了一个函数指针，以下的代码揭示如何定义一个函数指针数组。

```
int func1(int a, int b){
    return a + b;
}
int func2(int a, int b){
    return a - b;
}
int func3(int a, int b){
    return a * b;
}

void test(){
    int (*fun[3])(int, int);                //注意数组元素个数的位置
    func[0] = func1;
    func[1] = func2;
    func[2] = func3;

    for(int i = 0; i < 3; ++i){
        prinf("%d", func[i](i,i+1));
    }
}
```

7.2 函数指针做函数参数

目前我有一个需求——提供一个函数可以打印任意类型数据，下面是具体代码。

```
void printData(void* data, void (*myPrint)(void*)){
    myPrint(data);
}
```

我们需要确定函数`myPrinf`函数中具体的内容。由于开发者并不知道数据类型的具体结构，所以需要用户自己定义具体的`myPrinf`，用户和开发者结合起来才可以将其发挥作用。下面提供几个特殊数据类型的`myPrinf`。

```
void printInt(void* data){
    int* num = (int*)data;
    printf("%d\n", num);
}
void printChar(void* data){
    char* c = (char*)data;
    printf("%c\n", c);
}
void printDouble(void* data){
    double* num = (double*)data;
    printf("%f\n", num);
}

struct Person{
    char name;
    int age;
}myPerson;
void printPerson(void* data){
    myPerson* num = (myPerson*)data;
    printf("%c,%d\n", myPerson->name, myPerson->age);
}

void test(){
    int a = 10;
    char b = 'c';
    double c = 1.9;
    myPerson mk = {'a',24};

    printData(&a,printInt);
    printData(&b,printChar);
    printData(&c,printDouble);
    printData(&mk,printPerson);
}
```

其实这个`printData`函数叫做回调函数。这个案例很简单，并且感觉并不需要这么麻烦就可以打印出任意类型数据，但是这并没有影响到回调函数的重要性，下面的案例就会进一步突出回调函数重要性。

7.3 回调函数案例一

目前我有一个需求——提供一个函数可以打印任意类型数组，下面是具体代码。

```
void printfAllarray(void* arr, int eleSize,int len,
void (*myPrint)(void*)){
    char* p = (char*)arr;
    for(int i = 0; i < len; ++i){
        char* eleAddr = p + eleSize * i;
//获取到每一个元素的首地址地址
        myPrint(eleAddr);
//让用户自己定义函数，以打印每一个数组元素
    }
}
```

我们需要确定函数`myPrinf`函数中具体的内容。由于开发者并不知道数组元素的具体结构，所以需要用户自己定义具体的`myPrinf`，用户和开发者结合起来才可以将其发挥作用。下面提供几个特殊数据类型的`myPrinf`。

```
void printInt(void* data){
    int* num = (int*)data;
    printf("%d\n", num);
}
void printChar(void* data){
    char* c = (char*)data;
    printf("%c\n", c);
}
void printDouble(void* data){
    double* num = (double*)data;
    printf("%f\n", num);
}

struct Person{
    char name;
    int age;
}myPerson;
void printPerson(void* data){
    myPerson* person1 = (myPerson*)data;
    printf("%c,%d\n", person1->name, person1->age);
}

void test(){
    int a[3] = {1,2,3};
    char b[3] = 'abc';
    double c[3] = {1.9,2.2,3.7};
    myPerson d[3] = {{'a',24},{'b',19},{'c,30'}};
    int lenInt = sizeof(a) / sizeof(int);
    int lenChar = sizeof(b) / sizeof(char);
    int lenDouble = sizeof(c) / sizeof(double);
    int lenmyPerson = sizeof(d) / sizeof(myPerson);

    printfAllarray(&a,sizeof(int),lenInt,printInt);
    printfAllarray(&b,sizeof(char),lenChar,printChar);
```

```
35
    printfAllarray(&c,sizeof(double),lenDouble,printDoubl
    e);
36
    printfAllarray(&d,sizeof(myPerson),lenmyPerson,printP
    erson);
37  }
```

这个案例就比上一个复杂一些了，本案例中需要开发者自己维护每一个数组元素的首地址，然后答应每一个元素的工作就交给了用户自己完成。这个案例进一步的突出了回调函数的作用，开发者一定程度上完成了用户需求，同时也降低了用户自己完成需求的工作量。

7.4 回调函数案例二

目前我有一个需求——提供一个函数可以查找任意类型数组中的元素，下面是具体代码。

```
1   int findElement(void* arr, void* data,int eleSize, int
    len, void (*compareElement)(void*, void*)){
2       for(int i = 0; i < len; ++i){
3           char* temp = (char*)arr + eleSize * i;
4           if(compareElement(temp, data)){
5               printf("i have found data in arr!\n");
6               return 1;
7           }
8       }
9       return 0;
10  }
```

由于程序员开发代码的时候无法知道具体的数据类型，所以输入时都只能以`void*`接受数据。由于不知道数据结构，所以需要让用户自己提供比较函数（`compareElement`），以比较两个元素是否相等。代码中还利用了`char*`的指针步长为1的性质，以获取每一个数组元素首地址。

下面我们需要用户填写`compareElement`具体的内容，目前我写了几个不同类型数据对应的函数。

```
1   void compareIntelement(void* data1,void* data2){
2       int num1 = *(int*)data1;
3       int num2 = *(int*)data2;
4
5       return num1 == num2;
6   }
7   void compareDoublelement(void* data1,void* data2){
8       double num1 = *(double*)data1;
9       double num2 = *(double*)data2;
10
11      return num1 == num2;
12  }
```

```
13
14  struct Person{
15      char name;
16      int age;
17  }myPerson;
18  void comparePersonelement(void* data1,void* data2){
19      myPerson* person1 = (myPerson*)data1;
20      myPerson* person2 = (myPerson*)data2;
21      return (strcmp(person1->name, person2->name) == 0
    && person1->age == person2->age);
22  }
23
24  void test(){
25      int a[3] = {1,2,3};
26      double b[3] = {1.9,2.2,3.7};
27      myPerson c[3] = {{'a',24},{'b',19},{'c,30'}};
28      int lenInt = sizeof(a) / sizeof(int);
29      int lenDouble = sizeof(b) / sizeof(double);
30      int lenmyPerson = sizeof(c) / sizeof(myPerson);
31      finda = 2;
32      findb = 2.4;
33      findc = {'b', 18};
34
35      int resInt =
    findElement(&a,&finda,sizeof(int),lenInt,compareIntele
    ment);
36      int resDouble =
    findElement(&b,&findb,sizeof(double),lenDouble,compare
    Doublelement);
37      int resPerson =
    findElement(&c,&findc,sizeof(myPerson),lenmyPerson,com
    parePersonelement);
38  }
```

7.5 回调函数案例三

目前我有一个需求——提供一个函数可以对任意类型数组进行冒泡排序，下面是具体代码。

```
1   int bubbleSort(void* arr,int eleSize, int len, void
    (*compareElement)(void*, void*)){
2       char* temp = malloc(eleSize);
3       for(int i = 0; i < len - 1; ++i){
4           for(int j = 0; j < len - i - 1; ++j){
5               char* tempi = (char*)arr + eleSize * i;
6               char* tempiplus1 = (char*)arr + eleSize *
    (i + 1);
7               if(compareElement(tempiplus1, tempi)){
8                   memcpy(temp,tempiplus1,eleSize);
9                   memcpy(tempiplus1,tempi,eleSize);
10                  memcpy(tempi,temp,eleSize);
```

```
            }
        }
    }
    return;
}
```

由于程序员开发代码的时候无法知道具体的数据类型，所以输入时都只能以`void*`接受数据。由于不知道数据结构，所以需要让用户自己提供比较函数（`compareElement`），以比较两个元素大小。代码中还利用了`char*`的指针步长为1的性质，以获取每一个数组元素首地址。最后在交换数据的时候利用了`memcpy`函数，其第一个传入参数为被拷贝内存的首地址、第二个是拷贝内容的首地址、第三个是拷贝地址大小，这个函数主要是将内存整块拷贝，所以不用担心数据的具体类型。

下面我们需要用户填写`compareElement`具体的内容，在此函数中大于或者小于符号的选择最终会影响到数据是升序还是降序排列。目前我写了几个不同类型数据对应的函数。

```
void compareIntelement(void* data1,void* data2){
    int num1 = *(int*)data1;
    int num2 = *(int*)data2;

    return num1 > num2;
}
void compareDoublelement(void* data1,void* data2){
    double num1 = *(double*)data1;
    double num2 = *(double*)data2;

    return num1 > num2;
}

struct Person{
    char name;
    int age;
}myPerson;
void comparePersonelement(void* data1,void* data2){
    myPerson* person1 = (myPerson*)data1;
    myPerson* person2 = (myPerson*)data2;
    return (person1->age > person2->age);
}

void test(){
    int a[3] = {1,2,3};
    double b[3] = {1.9,2.2,3.7};
    myPerson c[3] = {{'a',24},{'b',19},{'c,30'}};
    int lenInt = sizeof(a) / sizeof(int);
    int lenDouble = sizeof(b) / sizeof(double);
    int lenmyPerson = sizeof(c) / sizeof(myPerson);

    int resInt =
bubbleSort(&a,sizeof(int),lenInt,compareIntelement);
```

```
33      int resDouble = bubbleSort(&b,sizeof(double),lenDouble,compareDoubleelement);
34      int resPerson = bubbleSort(&c,sizeof(myPerson),lenmyPerson,comparePersonelement);
35  }
```

8 预处理指令

8.1 预处理概念

- 程序变成可执行文件有以下步骤：预处理、编译、汇编、链接
- 预处理是在程序源代码被编译之前，由预处理器对程序源代码进行的处理
- 这个阶段并不会对源代码的语法进行解析，只是为下一步编译做准备

8.2 文件包含指令

- 文件包含可以源文件可以将一个另一个程序的全部内容给包含进来
- c语言提供了#include操作命令用于实现文件包含的操作

下图是文件包含的具体示意图

 无标题

8.2 宏定义

8.3.1 宏常量

具体的语法形式为：`#define name value`

具体代码示例为

```
#define PI 3.14
void test(){
    printf("%f", PI);
}
```

宏常量的定义一般有如下特性：

- 宏名一般是大写
- 宏定义不做语法方面的检查，只有编译被宏展开的源程序才会报错
- 宏定义从定义出到程序结束都可以使用，并且不注重作用域
- 宏定义的常量没有数据类型
- 可以使用`#undef`命令解释宏定义例如

```
#define PI 3.14
...
#undef PI
```

8.3.2 宏函数

具体的语法形式为：`#define func(parameter) expresion`

具体代码示例为

```
#define ADD(x,y) (( x ) + ( y ))
void test(){
    printf("%f", ADD(10,2) * 2);
}
```

宏定义的函数一般有如下特性：

- 在项目中可以将一些短小而又常用的程序写成宏函数
- 宏函数没有出入栈的时间开销，节约运算时间，提升程序效率
- 宏函数定义过程中一定注意不要多敲空格，但是在表达式可以敲空格
- 注意在编译时程序中出现宏定义的函数部分会直接用定义中的表达式替换，所以一定要使用空号以保证运算的优先级与宏函数的整体性
- 用大写字母表示宏函数名称

8.4 条件编译

一般的情况下源程序的所有代码都会被编译，但是也有例外，比如使用条件编译。程序员需要部分源代码在满足一定条件时才编译，这个时候就需要使用条件编译。

条件编译有三种使用方法：

1. 测试存在 `#ifdef`
2. 测试不存在`#ifndef`
3. 自定义条件`#if`

下面是使用案例。

```
//#ifdef
void test1(){
#define debug
#ifdef debug
    printf("%s", "debug versoin!");
#else
    printf("%s", "release versoin!");
#endif
}
```

```
//#ifndef
//多用于避免头文件（.h）重复包含
#ifndef _TEST_H
#define _TEST_H
...
#endif
//现在很多编译器都可以使用#pa
```

```
void test(){
#if 0
    printf("%s", "0!");
#else
    printf("%s", "1!");
#endif
}
```

8.5 特殊宏

有如下的特殊宏

- `__FILE__`宏所在的源文件绝对路径
- `__DATE__`代码编译的日期
- `__TIME__`代码编译的时间
- `__LINE__`宏所在的行号

9 动态链接库

10 递归函数

10.1 普通函数调用

在学习递归函数之前我们先查看如下的函数调用关系

```c
void func1(int a){
    printf("func1 a:%d\n",a);
}

void func2(int a){
    func1(a - 1);
    printf("func2 a:%d\n",a);
}

void test(){
    func2(2);

    printf("main\n");
    return 0;
}
```

函数运行结果为

```
func1 a:1
func2 a:2
main
```

10.2 递归函数调用

递归函数是自身调用自身的函数，其有如下的特性：

- 递归函数一开始有一个停机条件以避免无限递归，即类似于如下代码

  ```c
  if(stop condition is satisfied){
      ...
      return return_value;
  }
  ```

- 按照逻辑规则排列语句

以下代码是递归函数的简单示例：计算阶乘

```c
int A(int a){
    if(a == 1 || a == 0){
        return a;
    }

    return a * A(a - 1);
}
```

下面是第二个示例：逆序打印字符串数组

```c
void reversePrint(char* p){
    if(*p = '\0'){
        return;
    }

    reversePrint(p + 1);            //逆序打印是先打印后面的
    printf("%c", *p);               //再打印现在的
}
```

逆序打印字符串是先打印后面的字符再打印现在的字符，所以代码逻辑结构是先自身调用打印后面一个字符，再打印当前字符。

10.3 递归函数案例

对于递归函数，最为经典的案例之一就是费波纳希数列——后一个数是前两之和，其中数组最开始的两个元素为1。以下是用代码实现的计算费波纳希数列每一个元素值。

```c
int fibonacci(int a){
    if(a == 1 || a == 2){
        return 1;
    }

    return fibonacci(a - 1) + fibonacci(a - 2);
}
```