# *Android工程师如何进行C和C++的学习?*

我们知道NDK是越来越重要，作为机器学习、AI移动端落地都需要使用C和C++编译的库函数。在正式学习NDK之前，我们有必要对常用的工程C和c++编译、链接、排错有一个系统而全面的认识和了解。举个简单的例子，原来加载图片一般使用Universal-Image-Loader或者Picasso这些，但Facebook做的图片加载库fresco性能 秒杀之前的所有图片库，在部分数据上甚至可以达到一个数量级的性能提升，那Facebook是怎么做到的？它使 用了之前图片库没有使用过的匿名共享内存（Ashmem，Anonymous Shared Memory），这部分内存不在Android的GC范围之内，需要自己管理，但通过这种方式可以突破Android分配的堆内存限制，极大的提高效率，对于Ashmem的使用完全是通过C++实现，更别提很多核心算法，多媒体库等等都是用NDK做的。其实也不止C++，很多时候也要求助于Framework和架构重构。归根到底只想说明一点，如果只是做个普通的Android码农，那Java够用，但如果你的理想比这个还要大一点，C++是你进阶的必备技能。

目的：面试（细节 bat常用的点）bitmap mmap /sdcard/***/file fis fos

对比思维 java ===c

1.书

2.我讲的知识搞定

3.原理性知识 os 内存 jvm 线程

# *开班之前想说的话*

怎么学？怎么做？怎么思考？

- 一定要跟上老师的思路。直播一定要好好的听，课下要好好的学习
- 多问多思考，注意的是意而不是形
- 多看源码，分写总结

# *编辑器的选择和学习资料*

- window平台 Visual Studi、 Eclipse+CDT+MinGW CodeBlocks
- mac平台 CLion&Xcode
- linux平台 CLion&原生编译tools GCC + vim
- https://en.cppreference.com/w/

# C基础回顾

```
#include <stdio.h>  //预编译处理命令
main()
{
    int a,b,sum; /*定义变量a,b，sum为整型变量*/
    a=20; /*把整数20赋值给整型变量a*/
    b=15; /*把整数15赋值给整型变量b*/
    sum=a+b; /*把两个数之和赋值给整型变量sum*/
    printf("a=%d,b=%d,sum=%d\n",a,b,sum);
}
```

变量名：

- 标识符只能由字母、数字、下划线组成
- 标识符的第一个字母必须是字母和下划线
- 标识符区分大小写字母，如If和if是两个完全不同的标识符
- 不能使用关键字（32个）

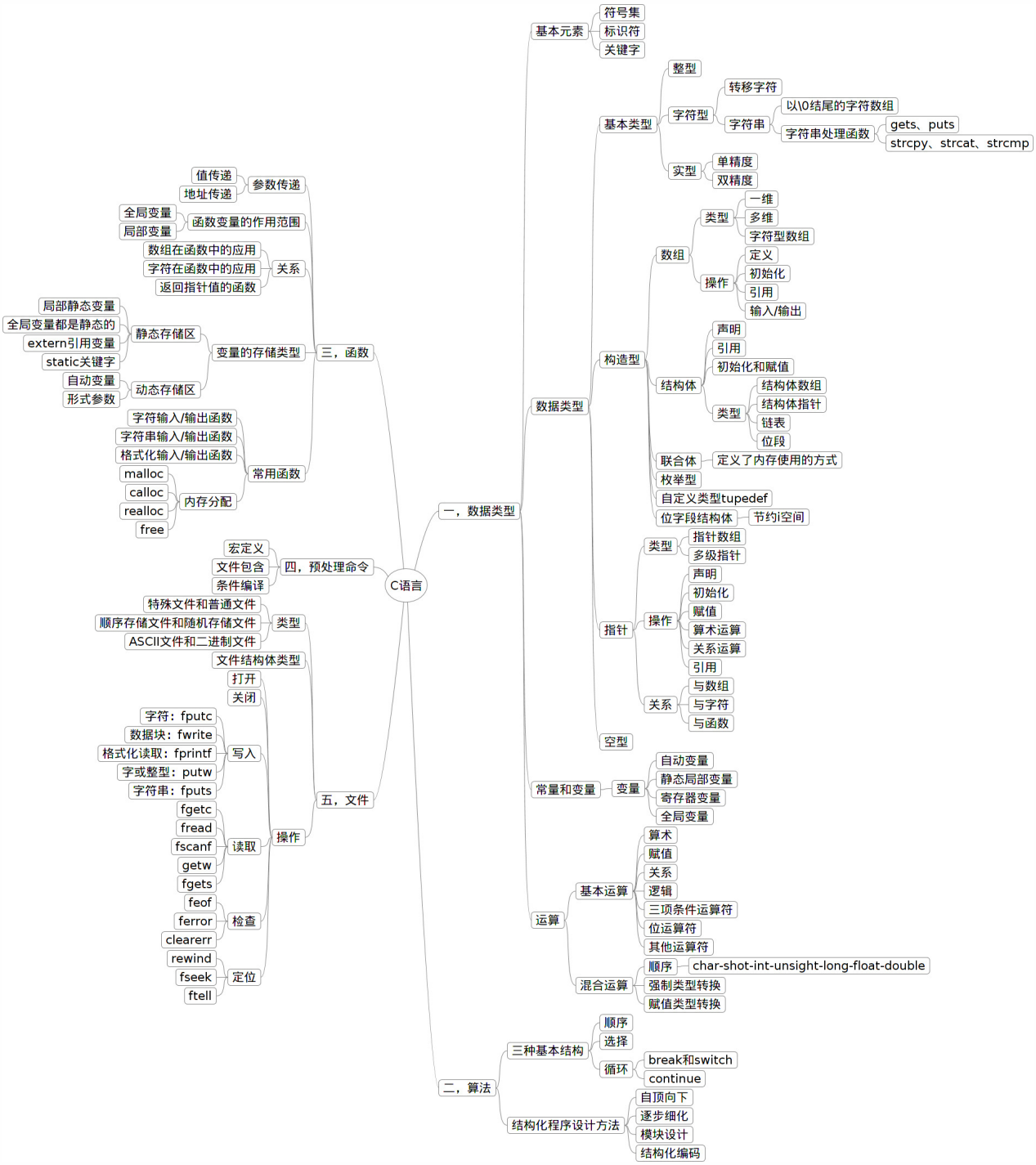
变量本质：

- 程序通过变量来申请和命名内存空间 int a = 0，通过变量名访问内存空间。
- 修改变量有几种方法？直接/间接（指针修改）
- 变量三要素（名称、大小、作用域）

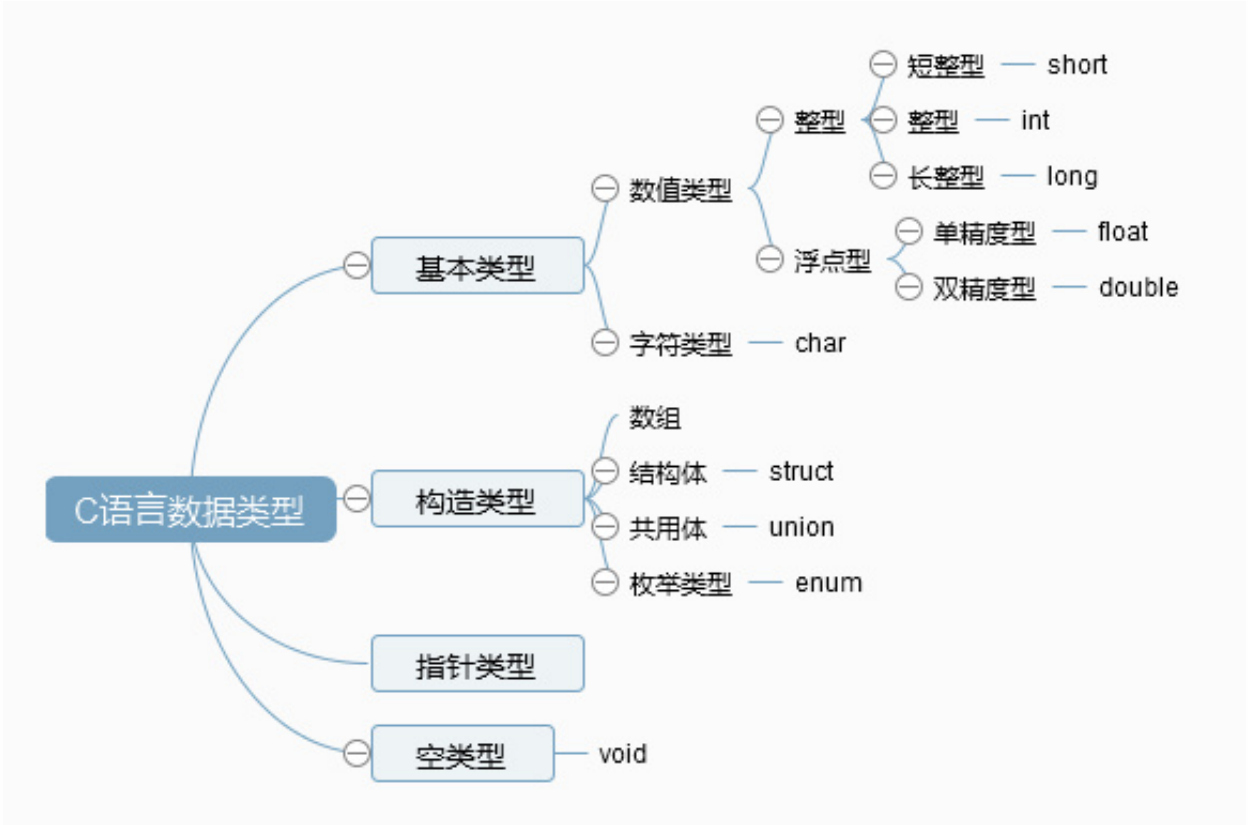
关键字：

```
auto break case char union do double else enum extern goto if int long short
signed static
sizof struct switch unsigned void for while typedef continue float return
typedef default
```

C语言技能树：


C语言
一，数据类型
基本元素
符号集
标识符
关键字
数据类型
基本类型
整型
字符型
转移字符
字符串
以\0结尾的字符数组
字符串处理函数
gets、puts
strcpy、strcat、strcmp
实型
单精度
双精度
构造型
数组
类型
一维
多维
字符型数组
操作
定义
初始化
引用
输入/输出
结构体
声明
引用
初始化和赋值
类型
结构体数组
结构体指针
链表
位段
联合体
定义了内存使用的方式
枚举型
自定义类型tupedef
位字段结构体
节约i空间
指针
类型
指针数组
多级指针
操作
声明
初始化
赋值
算术运算
关系运算
引用
关系
与数组
与字符
与函数
空型
常量和变量
变量
自动变量
静态局部变量
寄存器变量
全局变量
运算
基本运算
算术
赋值
关系
逻辑
三项条件运算符
位运算符
其他运算符
混合运算
顺序
char-shot-int-unsight-long-float-double
强制类型转换
赋值类型转换
二，算法
三种基本结构
顺序
选择
循环
break和switch
continue
结构化程序设计方法
自顶向下
逐步细化
模块设计
结构化编码
三，函数
参数传递
值传递
地址传递
函数变量的作用范围
全局变量
局部变量
关系
数组在函数中的应用
字符在函数中的应用
返回指针值的函数
变量的存储类型
静态存储区
局部静态变量
全局变量都是静态的
extern引用变量
static关键字
动态存储区
自动变量
形式参数
常用函数
字符输入/输出函数
字符串输入/输出函数
格式化输入/输出函数
内存分配
malloc
calloc
realloc
free
四，预处理命令
宏定义
文件包含
条件编译
五，文件
类型
特殊文件和普通文件
顺序存储文件和随机存储文件
ASCII文件和二进制文件
文件结构体类型
操作
打开
关闭
写入
字符：fputc
数据块：fwrite
格式化读取：fprintf
字或整型：putw
字符串：fputs
读取
fgetc
fread
fscanf
getw
fgets
检查
feof
ferror
clearerr
定位
rewind
fseek
ftell

C语言数据类型：




C语言的格式输出：%d %ld %lld %c %f %lf %u %x %u %o %s 自己复习整理下

%u unsigned short

常量：记住宏常量(符号常量) #define PI 3.1415926

字符串常量 char str[] = "ABCD";

const int a = 100;

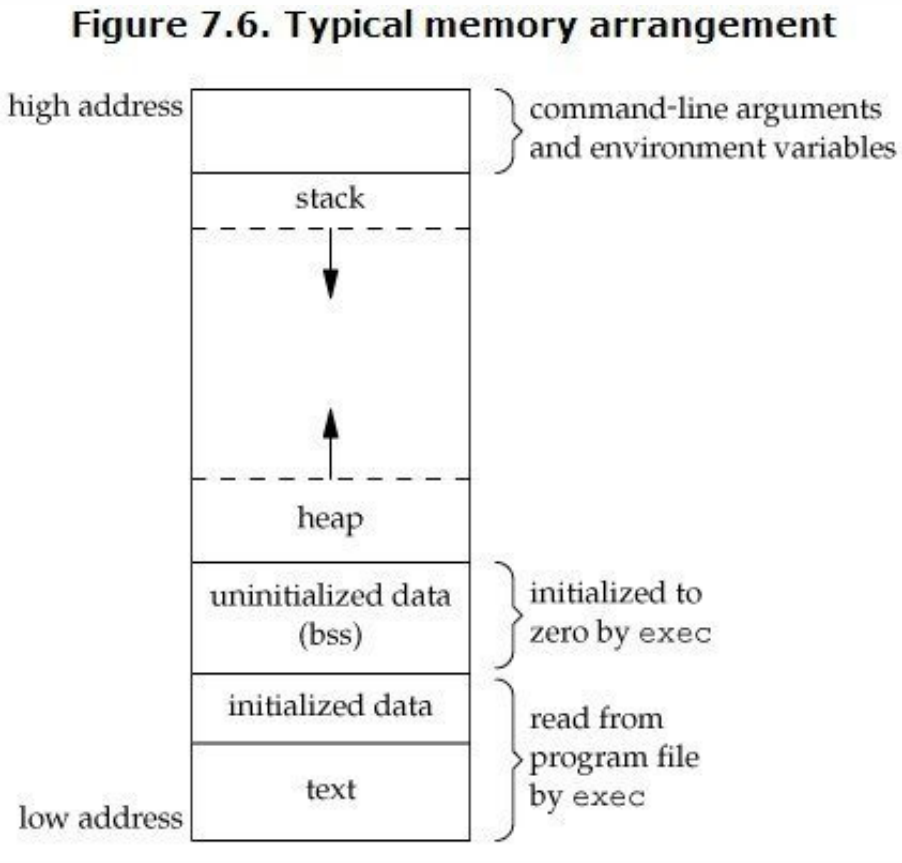
c99 定义了boolean，但是它不是基本类型。int就可以表示boolean 需要引入 #include <stdbool.h>

***TIPS***:

- void* 代表指针地址存储的类型可以多种。 C语言规定只有相同类型的指针才可以相互赋值 void* 指针作为左值用于“接收”任意类型的指针, void* 指针作为右值赋值给其它指针时需要强制类型转换
- 数据类型本质是固定内存大小的别名。大小计算用sizeof，数据类型的封装一般我们使用void *
- 可以给已存在的数据类型起别名typedef
- 我们在开始中一般建议使用重载的int32_t,在不同的操作系统使用sizeof表达式

# *内存管理*

一般我们对于内存的分类也就这几种：栈区（stack area）、堆区（heap area）、全局区（静态区）（存放全局变量与静态变量static）、BSS段（存放未初始化的全局变量，未初始化的全局变量默认值为0）、文字常量区、数据区（data area）、代码区（code area）等。




代码理解：

```
# include<stdio.h>
# include<stdlib.h>
# include <unistd.h>

int num1;/*BSS段*/
int num2 = 20;/*全局区*/
char * str1 = "str1";/*文字常量区*/

int main(void){
    printf("%d\n",getpid());/*获取当前进程id号*/
    int num3 = 3;/*栈区*/
    static int num4 = 4;/*全局区*/
    const int num5 = 5;/*栈区*/
    char * str2 = "str2";/*文字常量区*/
    char str3[] = "str3";/*栈区*/
    int * p = malloc(sizeof(0));/*&p在栈区，p在堆区*/


 printf("num1:%p\nnum2:%p\nnum3:%p\nnum4:%p\nnum5:%p\n",&num1,&num2,&num3,&nu
m4,&num5);
    printf("str1:%p\nstr2:%p\nstr3:%p\n",str1,str2,str3);
    printf("&p:%p\np:%p\n",&p,p);

    while(1){}/*死循环以保证进程不会结束，方便查看/proc/pid/maps文件*/
    free(p);
    return 0;
}
```

vim /proc/process id/maps

```
00400000-00401000 r-xp 00000000 fd:00 38574235                           /home/hadoop/code/testmem2
00600000-00601000 r--p 00000000 fd:00 38574235                           /home/hadoop/code/testmem2
00601000-00602000 rw-p 00001000 fd:00 38574235                           /home/hadoop/code/testmem2
02034000-02055000 rw-p 00000000 00:00 0                                  [heap]
7fca79ca8000-7fca79e6a000 r-xp 00000000 fd:00 73705                      /usr/lib64/libc-2.17.so
7fca79e6a000-7fca7a06a000 ---p 001c2000 fd:00 73705                      /usr/lib64/libc-2.17.so
7fca7a06a000-7fca7a06e000 r--p 001c2000 fd:00 73705                      /usr/lib64/libc-2.17.so
7fca7a06e000-7fca7a070000 rw-p 001c6000 fd:00 73705                      /usr/lib64/libc-2.17.so
7fca7a070000-7fca7a075000 rw-p 00000000 00:00 0
7fca7a075000-7fca7a097000 r-xp 00000000 fd:00 73698                      /usr/lib64/ld-2.17.so
7fca7a27d000-7fca7a280000 rw-p 00000000 00:00 0
7fca7a294000-7fca7a296000 rw-p 00000000 00:00 0
7fca7a296000-7fca7a297000 r--p 00021000 fd:00 73698                      /usr/lib64/ld-2.17.so
7fca7a297000-7fca7a298000 rw-p 00022000 fd:00 73698                      /usr/lib64/ld-2.17.so
7fca7a298000-7fca7a299000 rw-p 00000000 00:00 0
7ffcbaddc000-7ffcbadfd000 rw-p 00000000 00:00 0                          [stack]
7ffcbadfd000-7ffcbadff000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

备注:

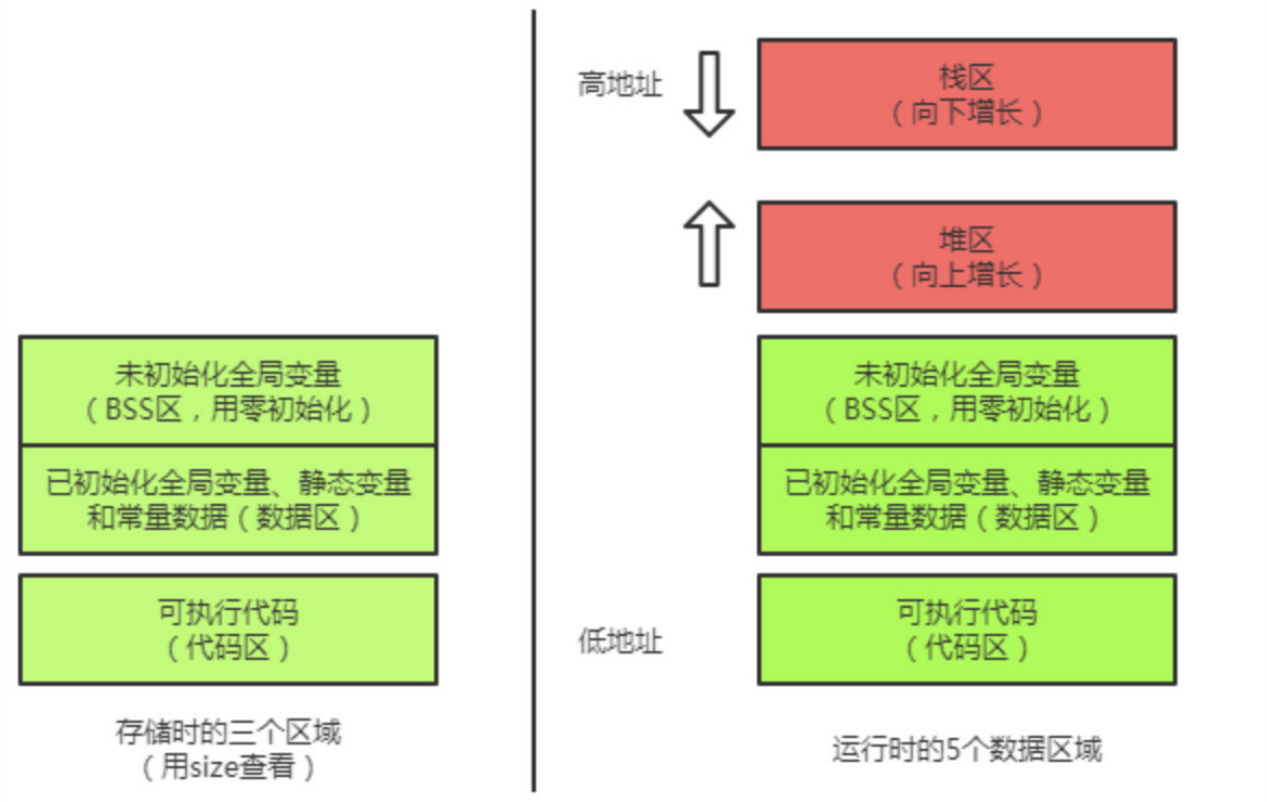
第一列代表内存段的虚拟地址
第二列代表执行权限，p=私有 s=共享 heap和stack段不应该有x
第三列代表在进程地址里的偏移量
第四列映射文件的主设备号和次设备号 通过 cat /proc/devices得知fd
第五列映像文件的节点号，即inode
第六列是映像文件的路径
补充资料：https://www.cnblogs.com/jiayy/p/3458076.html

首先看代码:

```c
 #include <stdlib.h>
 int var1 = 1;
  void test();
void test2(){
      int var2 = 2;
    test1();
}
  int main(void) {
  int var2 = 2;
  printf("hello, world!\n");
  exit(0);
 }
```

```
 Jesson ?~/cainiaowo/c ?ls -al
total 32
drwxr-xr-x  4 wuyue  staff   128 11 17 00:11 .
drwxr-xr-x  3 wuyue  staff    96 11 17 00:08 ..
-rwxr-xr-x  1 wuyue  staff  8512 11 17 00:11 a.out
-rw-r--r--  1 wuyue  staff   135 11 17 00:09 mem.c
 Jesson ?~/cainiaowo/c ?ls -al a.out
-rwxr-xr-x  1 wuyue  staff  8512 11 17 00:11 a.out
 Jesson ?~/cainiaowo/c ?file a.out
a.out: Mach-O 64-bit executable x86_64
 Jesson ?~/cainiaowo/c ?size a.out
__TEXT   __DATA   __OBJC  others  dec     hex
4096     4096     0       4294971392      4294979584      100003000
 Jesson ?~/cainiaowo/c ?
```

不同的OS看到的红色圈圈可能不一样。可执行文件在存储（也就是还没有载入到内存中）的时候，分为：代码区、数据区和未初始化数据区3个部分。下图所示为可执行代码存储时结构和运行时结构的对照图。一个正在运行着的C编译程序占用的内存分为代码区、初始化数据区、未初始化数据区、堆区和栈区5个部分。




- 代码区（text segment）。代码区指令根据程序设计流程依次执行，对于顺序指令，则只会执行一次（每个进程），如果反复，则需要使用跳转指令，如果进行递归，则需要借助栈来实现。代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。代码区的指令中包括操作码和要操作的对象（或对象地址引用）。如果是立即数（即具体的数值，如5），将直接包含在代码中；如果是局部数据，将在栈区分配空间，然后引用该数据地址；如果是BSS区和数据区，在代码中同样将引用该数据地址。另外，代码段还规划了局部数据所申请的内存空间信息。
- 全局初始化数据区/静态数据区（Data Segment）。只初始化一次。数据段： 数据段（data segment

）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。data段中的静态数据区存放的是程序中已初始化的全局变量、静态变量和常量。

- 未初始化数据区（BSS）。在运行时改变其值。BSS 段： BSS 段（bss segment ）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS 是英文Block Started by Symbol 的简称。BSS 段属于静态内存分配，即程序一开始就将其清零了。一般在初始化时BSS段部分将会清零。
- 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。存放函数的参数值、局部变量的值，以及在进行任务切换时存放当前任务的上下文内容。其操作方式类似于数据结构中的栈。每当一个函数被调用，该函数返回地址和一些关于调用的信息，比如某些寄存器的内容，被存储到栈区。然后这个被调用的函数再为它的自动变量和临时变量在栈区上分配空间，这就是C实现函数递归调用的方法。每执行一次递归函数调用，一个新的栈框架就会被使用，这样这个新实例栈里的变量就不会和该函数的另一个实例栈里面的变量混淆。栈(stack)： 栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧"{}"中定义的变量（但不包括static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/ 恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。
- 堆区（heap）。用于动态内存分配。堆在内存中位于bss区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时有可能由OS回收。堆(heap)： 堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。在将应用程序加载到内存空间执行时，操作系统负责代码段、数据段和BSS段的加载，并将在内存中为这些段分配空间。栈段亦由操作系统分配和管理，而不需要程序员显示地管理；堆段由程序员自己管理，即显式地申请和释放空间。

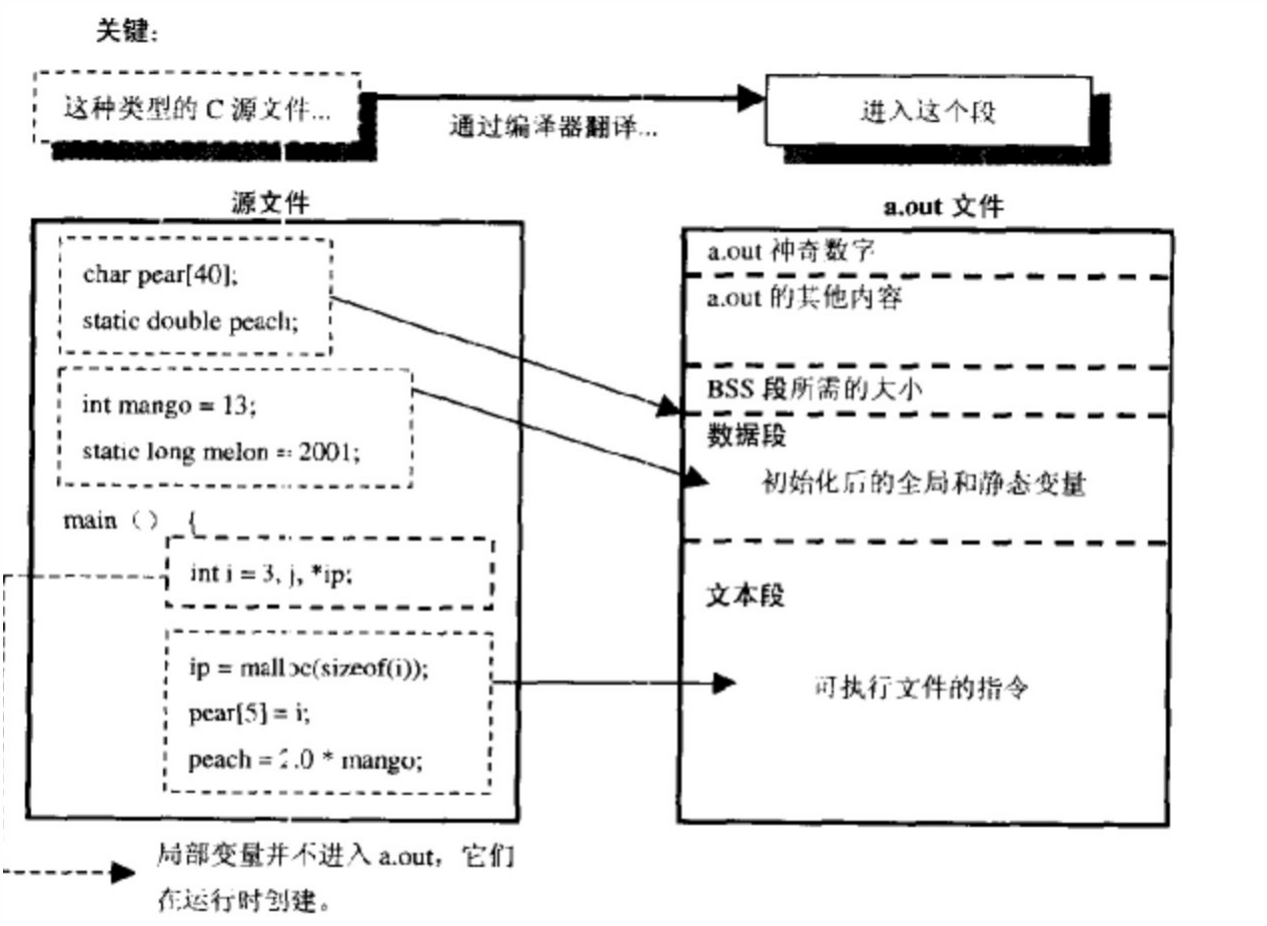



图 6-1　C 语句的各部分会出现在哪些段中

*思考*：我们的c的函数是如何像操作系统申请内存的？和jvm游什么区别？

https://www.gnu.org/software/libc/

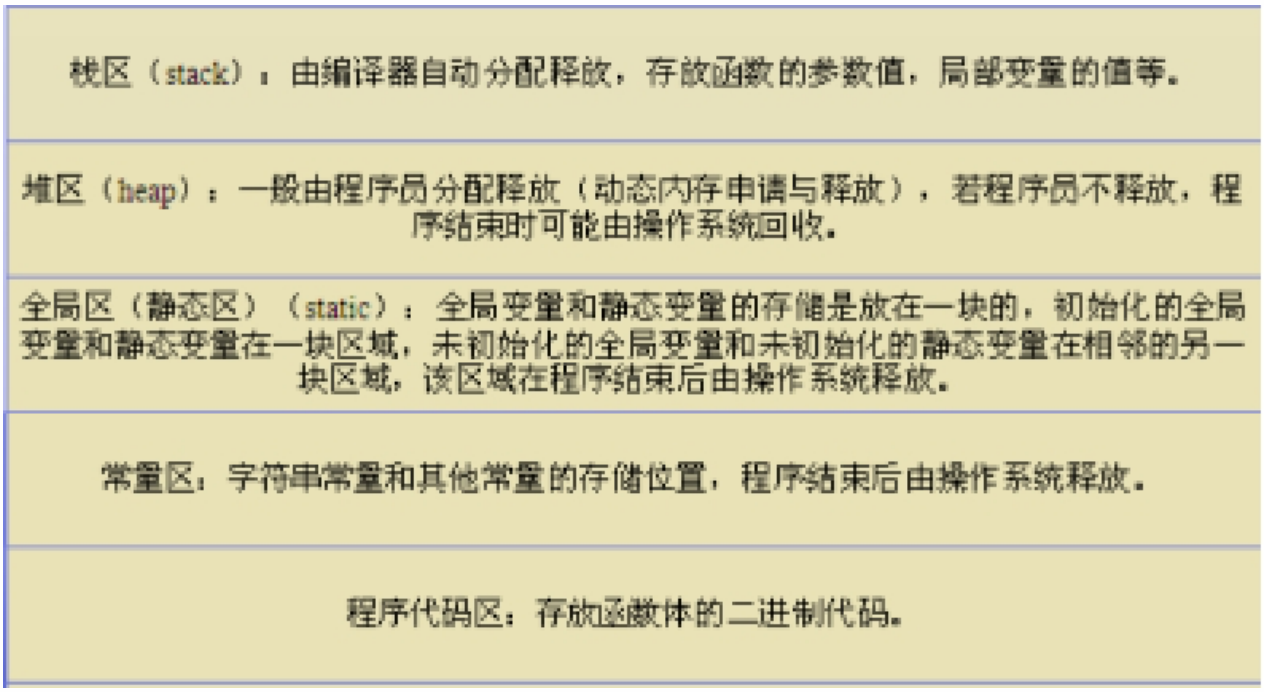
https://www.gnu.org/software/libc/manual/pdf/libc.pdf

我们一般使用malloc来分配内存，属于c标准库函数。malloc分配的内存是从堆中分配的。linux系统向用户提供申请的内存有brk(sbrk)和mmap函数。

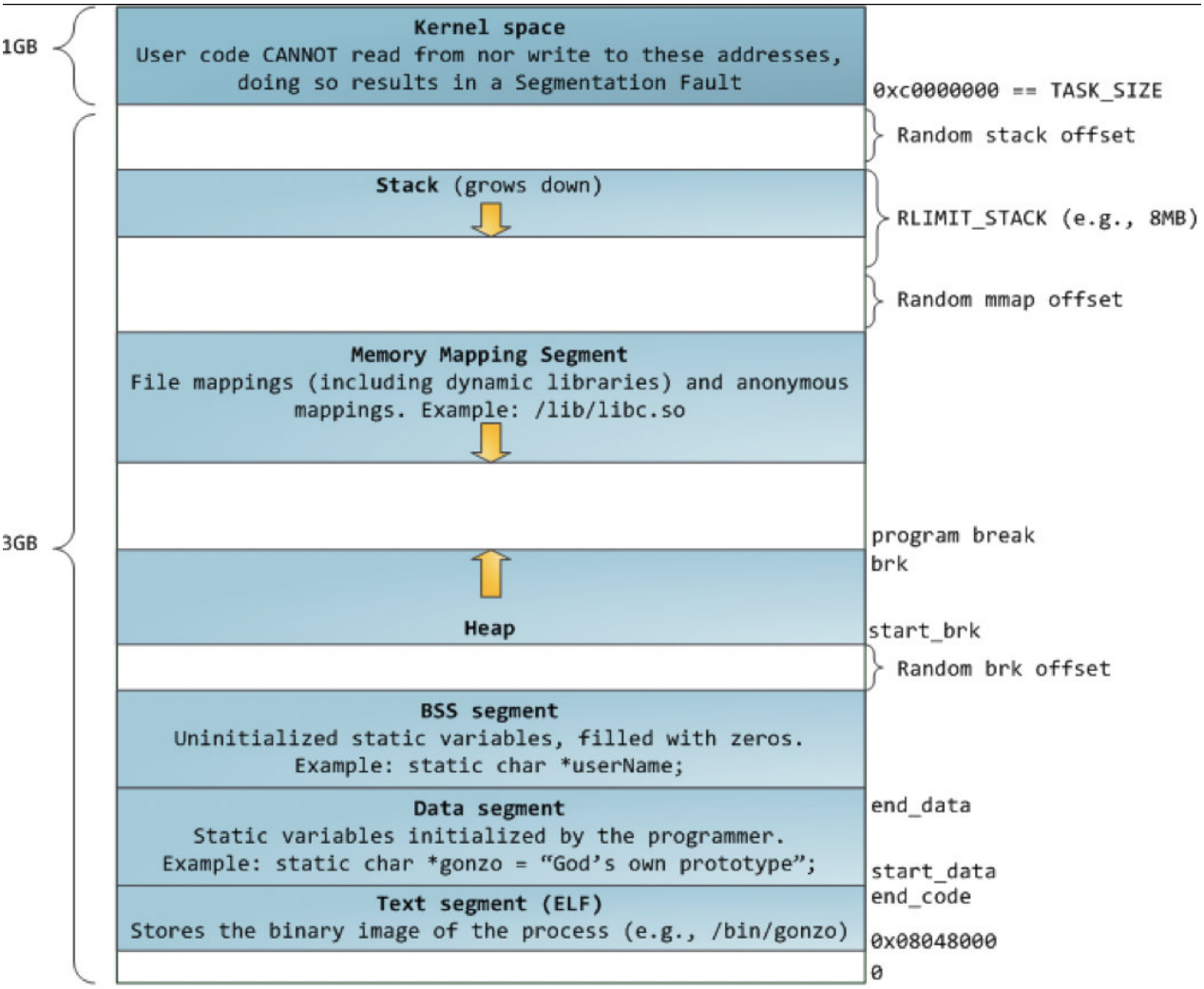
案列1：手动分析代码在内存分布

```
/* memory_allocate.c用于演示内存分布情况 */

  int a = 0;                          /* a在全局已初始化数据区 */
  char *p1;                           /* p1在BSS区(未初始化全局变量) */

  int main(void) {
    int b;                            /* b在栈区 */
    char s[] = "abc";                 /* s为数组变量, 存储在栈区 */
    /* "abc"为字符串常量, 存储在已初始化数据区 */
   char *p1, p2;                      /* p1、p2在栈区 */
   char *p3 = "123456";               /* "123456\0"已初始化在数据区, p3在栈区 */
   static int c = 0;                  /* c为全局(静态)数据, 存在于已初始化数据区 */
   /* 另外, 静态数据会自动初始化 */
   p1 = (char *)malloc(10);           /* 分配的10个字节的区域存在于堆区 */
   p2 = (char *)malloc(20);           /* 分配得来的20个字节的区域存在于堆区 */

   free(p1);
   free(p2);
 }
```

总结：




进程内存布局：




作业：brk mmap 看源码分析

动态内存申请总结：

**malloc** 没有初始化内存的内容,一般调用函数memset来初始化这部分的内存空间.free（）

**calloc** 申请内存并将初始化内存数据为null.

```
int *pn = (int*)calloc(10, sizeof(int));
```

**realloc**

对malloc申请的内存进行大小的调整.

```
char *a = (char*)malloc(10);
realloc(a,20);
```

特别的： **alloca** 在栈申请内存,因此无需释放. `int *p = (int *)alloca(sizeof(int) * 10);`

面试题目：malloc(0)； 将返回什么?

```c
#include <stdio.h>
#include <malloc.h>

int main()
{
    int* p = (int*)malloc(0);
    printf("%p\n", p);
}
```

参见代码 动态内存分配.c

## *数组和指针*

数组：int a[5];

数组是内存中一组连续的空间，上面的数组a是内存中连续的5个int型的空间。这段空间在数组初始化的时候就开辟出来了。数组名a就是这段连续空间的起始地址，同时也是第一个元素的地址。相当于&a[0]。下标表示了每个元素的具体位置，同时也表示这个元素与第一个元素之间的距离。于是a[i]相当于a[0 + i]。

```c
//main.c first.c
int a[5];//数组下标从0开始的，a【5】是不存在的。为什么从0开始计数呢？数组的大小也不能动态的定义，编译器是不识别的
int i;
for(i = 0; i < 5; i++)
{
    a[i] = i + 1;
}
for (i = 0; i < 5; i++)
{
    printf("%d ", a[i]);
}
```

tips:一定要关注数据在内存中保存的基本形式，比如int a[100],那么int 4个字节，100*4是不是开辟了400个字节的内存空间。搞明白数组的引用和数组的声明。

案例：实现函数类部的冒泡排序(c++实现，同学们用c来做一遍)

```c
//maopao.cpp
#include <iostream>
using namespace std;

int main()
{
    int n, a[1000];
    cin >> n;//输入n个数
```

```
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    //冒泡，不断比较相邻的两个数，如果顺序错了，那么就交换
    for (int i =0; i < n; i++)
    {
        for (int j = 1; j < n-i; j++)
        {
            if (a[j-1] > a[j])
            {
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
    //依次输出
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << endl;
    }
    return 0;
}
```

指针：指针是个地址，它是内存中一段数据的地址。可以理解就是指针就是变量，数值是地址

思考问题：数组和指针有什么关联，又有什么区别呢?

数组名是数组的首地址。

*数组指针和指针数组*

```
# include<stdio.h>
# include<stdlib.h>
# include <unistd.h>

void main(){
    int array[2][3] = {{1,2,3},{4,5,6}};
    //数组指针（也称行指针）
    //int (*p)[n]; p是一个指针，指向一个整型的一维数组，这个一维数组的长度是n
    int (*array_p)[3] = array;
    //取出5
    printf("%x\n", (unsigned int) array_p);
    printf("%x\n", (unsigned int) (array_p + 1));
    //printf("%x\n", (unsigned int) (array_p + 2));
    printf("取出2元素==>%d\n",*(*(array_p+0)+1));

    //普通遍历
    for(int i=0;i<2;i++){
        for(int j=0;j<3;j++){
            printf("%d\n",array[i][j]);
```

```
        }
    }
    //求二维数组元素的最大值。
    int *p,max;
    for(p=array[0],max=*p;p<array[0]+6;p++)
        if(*p>max)
            max=*p;
    printf("MAX=%d",max);
}
```

案列2：代码分析

```
#include "demo2.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
   int *a,*b;
   a = (int*)malloc(sizeof(int));
   b = (int*)malloc(sizeof(int));
   *a = 100;
   *b = 200;
   printf("a的地址是:%p,%p,%d\n",&a,a,*a); //地址空间占用8个字节
   printf("b的地址是:%p,%p,%d\n",&b,b,*b);

   int c[3] = {0,1,2}; //计算机中 数组就是一个地址了，注意和变量的区别
   printf("c的地址是:%p,%p,%d,%d,%d\n",&c,c,c[0],c[1],c[2]);

   return 0;
}
```

操作系统管理内存原理：

栈空间：堆空间：内存映射：

stack由系统自动分配，heap需要程序员自己申请，C中用函数malloc分配空间，用free释放，C++用new分配，用delete释放。对于堆空间来说，默认是没有软限制的，只依赖硬限制。栈空间可以通过ulimit -a命令进行查找。

内存泄漏：申请的内存一直申请不释放。

野指针：占用别的程序申请的内存。

函数指针:

```
int func(int x);/*申明一个函数*/
int (*f)(int x);/*申明一个函数指针*/
f = func;/*将函数func的首地址赋值给f*/
```

指针强化问题:

- 理解指针关键在内存，没有内存哪来的内存首地址，没有内存首地址，哪来的指针啊。
- 指针也是一种数据类型，指针的数据类型是指它所指向内存空间的数据类型。
- 指针指向谁就把谁的地址赋给指针。
- 间接赋值（*p）是指针存在的最大意义 。

案列3：间接赋值 指针推导

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

//指针做函数参数间接改变我们的运算结果
//int getFoleLen(int*p)
//{
//    *p = 40;
//    return true;
//}
////这样只能修改形参的值  不能改变实参的值
////不通过操作地址的话  编译器是将实参的值复制给形参 而不是把实参传递进来
int getFoleLen(int b)
{
    //形参变量和函数里面的变量本质是一样的  只不过形参变量具有对外的属性
    b = 100;
    return true;
}
////return只能返回一个结果
//int getFoleLen()
//{
//    int a = 100;
//    return a;
//}

//一级指针到0级指针
void main()
{
    int* p = NULL;
    int a = 10;
    //修改a的值
    //直接修改
    a = 12;
    //间接修改
    p = &a;
    *p = 20;//p的值是a的地址 *就像一把钥匙  通过地址 找到一块内存空间  就间接修改了a的值
    printf("a,%d\n", a);
    getFoleLen(&a);
    printf("a1,%d\n",a);
    getFoleLen(a);
    printf("a2,%d\n", a);
```

```
}
```

## *指针细节总结*

指针到底是什么？本质是什么？

指针组合背诵：

```
void main(){
    char str[] = "jesson";
    const char *p = str;
    // p[0] = 'a'; 不能通过指针修改char的内容
    char const *p3 = str; //和上面一样的 const 只能修饰char

    char * const p4 = str;
    p4[0] = 'a';//能通过指针修改char的内容
    //p4 = "sfsdf";

    const  char * const p5 = str;
    char   const * const p6 = str;
//    p5[0] = 'a';
//    p6[0] = 'a';
}
```

回归本质，char* p = null。本质上指针就是一个变量，一般来说含有指针的变量我们称之为指针变量，指针变量占用4个字节。注意指针变量和它指向的内存块是两个不同的概念。分析一个简短的代码：

```
    int *a = NULL;
    int b = 100;
    a = &b;
    *a = 200;
    int c =0;
    c = *a;
    printf("b的数值是：%d,a的数值是：%d,C的数值是：%d",b,*a,c); //间接修改变量的数值

    char * str = getStr();
    printf("字符信息是：%s\n",str);
    *(str+1)='r'; //str指向的是常量池，这里肯定会报错
    printf("字符信息是：%s\n",str);
```

为什么C语言要设计指针？指针的最重要的作用是什么？*答案就是--->间接赋值*

```
int getlength(int* p){
    *p = 300;
}
void main(){
    int a = 10;
    int *p = NULL;
    p = &a;
    *p = 30;
    printf("a的数值是：%d",a);
    getlength(p); //是不是通过函数来间接修改其他变量的数值？形参可以修改实参的数值，这就是指针的魅力
    printf("a的数值是：%d",a);
}
```

我们再来看看多级指针？

```
int main(void)
{
    int i = 10;
    int *p = &i;
    int **q = &p;
    int ***r = &q;
    printf("i = %d\n", ***r); //10
    return 0;
}
```

案列3：二级指针操作文件

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

void readFileData(FILE *pFILE, int num, char **pString);
void printData(char **pString, int num);
int getFilelinenum(FILE* file){
    if(file ==NULL){
        printf("文件打开失败");
        return -1;
    }
    int num = 0;
    char buff[1024];//定义1024缓冲区
    while(fgets(buff,1024,file)!=NULL){
        printf("%s",buff);
        num++;
    }
    return num;
}

/**
```

```
 *
 * @param pFILE  文件指针
 * @param num 有效行数
 * @param pString 把有效的数据放到堆区
 */
void readFileData(FILE *pFILE, int num, char **pString) {
    if(pFILE ==NULL||num<0||pString==NULL){
        printf("参数错误");
        return ;
    }
    int index =0;
    char buff[1024];//定义1024缓冲区
    while(fgets(buff,1024,pFILE)!=NULL){
        int currentLen = strlen(buff)+1;
        char * current = malloc(sizeof(char*)*currentLen);
        strcpy(current,buff);
        pString[index++] = current;
        //清空缓冲区
        memset(buff,0,1024);
    }
}

void printData(char **pString, int num) {
    for (int i = 0; i < num; ++i) {
        printf("第%d行的数据是%s",i+1,pString[i]);
    }
}

int main(){
    printf("======BEGIN=========\n");
    FILE* file = fopen("../test.txt","r");
    if(file == NULL){
        printf("文件打开失败");
        return -1;
    }
   //注意这里把函数注释掉，程序就好了，为什么？要学员思考？(文件指针移动)
   // int num = getFilelinenum(file);
    int num = 5;
    char** pArrays = malloc(sizeof(char*)*num);
    readFileData(file,num,pArrays);
    printData(pArrays,num);
}
```

案列4：二级指针作为函数参数的输出特性

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

//释放内存给大家做
```

```
void testfunction();
void allocaMem(int **pInt);
void printArray(int **pInt, int i);

int main(){
    printf("======BEGIN=========\n");
    testfunction();
}

void testfunction() {
    int* p = NULL;
    allocaMem(&p);
    printArray(&p,10);
}

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

void allocaMem(int **p) {
    int* arr = malloc(sizeof(int)*10);
    for (int i = 0; i < 10; ++i) {
        arr[i] = i+100;
    }
    *p = arr;
}
```

# *函数*

- 可变函数 stdarg.h头文件正是解决这个问题而生，不过其用法比较繁琐，必须按照如下步骤进行：

1. 一个使用省略号的函数原型；
2. 在函数定义中创建一个va_list类型的变量；
3. 用宏把该变量初始化为一个参数列表；
4. 用宏访问参数列表；
5. 用宏清理

该函数原型必须有一个形参和一个省略号，例如：

void foo1(int lim, ...); / / 有效 void foo2(const char *s, int k, ...); / / 有效 char foo3(char c1, ... , char c2); / / 无效，省略后不在后面 double foo4(...); / / 无效，没有形参

参见代码 函数.c

- 函数指针

**函数指针变量**

一个数据变量的内存地址可以存储在相应的指针变量中，函数的首地址也以存储在某个函数指针变量中。这样，我就可以通过这个函数指针变量来调用所指向的函数了。

在C系列语言中，任何一个变量，总是要先声明，之后才能使用的。函数指针变量也应该要先声明。

函数指针变量的声明：

```
void (*funP)(int) ;   //声明一个指向同样参数、返回值的函数指针变量。
```

## *预处理器*

**C 预处理器**不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，他们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

| 指令 | 描述 |
| --- | --- |
| #define | 定义宏 |
| #include | 包含一个源代码文件 |
| #undef | 取消已定义的宏 |
| #ifdef | 如果宏已经定义，则返回真 |
| #ifndef | 如果宏没有定义，则返回真 |
| #if | 如果给定条件为真，则编译下面代码 |
| #else | #if 的替代方案 |
| #elif | 如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码 |
| #endif | 结束一个 #if……#else 条件编译块 |
| #error | 当遇到标准错误时，输出错误消息 |
| #pragma | 使用标准化方法，向编译器发布特殊的命令到编译器中 |

常用举例：

```
#define MAX_ARRAY_LENGTH 20

#include <stdio.h>
#include "myheader.h"

#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

```
printf("File :%s\n", __FILE__ );
printf("Date :%s\n", __DATE__ );
printf("Time :%s\n", __TIME__ );
printf("Line :%d\n", __LINE__ );
printf("ANSI :%d\n", __STDC__ );


#if defined(OS_UNIX)
    #ifdef OPTION1
      unix_version_of_option1();
    #endif
    #ifdef OPTION2
      unix_version_of_option2();
    #endif
#elif defined(OS_MDOS)
    #ifdef OPTION2
          msdos_version_of_option2();
   #endif
#endif
```

# *字符串函数和指向字符串指针总结*

C语言中提供了大量丰富的字符串处理函数，大致可以分为字符串的输入，输出，合并，修改，比较，转换，复制，搜索等几类。 用于输入输出的函数包含在stdio.h中而其他的字符串处理函数一般包含在string.h中。下面是编程中常用的函数，需要掌握。

- 字符串的输出函数 int puts(char const*strPtr); 输出字符串到显示器中去，传入的实参可以是字符串数组也也可以是字符指针 一般情况下我们用fputs来代替
- 字符串的输入函数 char *gets(char *strptr); char *fgets(); 从标准键盘输入一个字符串存放到指针strptr指向的字符数组，一般我们使用fgets来操作
- 获取字符串的长度 int strlen(char const*string); 获取字符串的长度，返回的是字符的个数，但是不会包括'\0',结束符
- 字符串拷贝函数 char *strcpy(char*des,char*src); char *strncpy(char *des,char *src,int size); 将src指向的字符串拷贝到des指向的字符串数组中去，结束符也一同进行拷贝，size参数也可以拷贝制定长度的字符串，建议des为字符数组 注意:目标尽量使用字符数组，因为如果是字符指针的话，分配的内存在常量池中，是不允许进行改变的，容易造成段错误
- 字符串的连接函数 char * strcat(const *char str1,const *char str2); char *strncat(const *char str1,const *char str2,int size); 将str2指向的字符串连接到str1指向的字符后面，同时会删除str1后面的'\0',返回的是str1指向字符串的首地址重点内容
- 字符串比较函数 int strcmp(const char * str1,const char *str2); int strncmp(const char*str1,const char *str2,int size); 按照ascii码来进行比较，并由函数返回值进行判断 返回0,字符串1等于字符串2, 大于0,字符串1大于字符串2, 小于0,字符串1小于字符串2,
- 字符串分割函数 char* strtok(char *str,const char *delimiters); 根据delimiters指向的分割符号，将str指向的字符串分割成若干个子字符串，同时返回的是分割出的子字符串
- 字符串中是否包含字符 char * strchr(const char *,int c);//从做往右寻找字符c char * strrchr(const

char *,int c); / /从有往左开始寻找字符c 判断字符串中是否包含字符串；

- 内存的初始化 memset(void *s,int c,size_t n);
- 内存的拷贝

memcpy(void *des,void *src ,size_t ,n);

代码实战：

```
#include<stdio.h>
#include<string.h>

int main(){
    char str_array[20] = {'\0'};
    //gets(str_array);
    //puts(str_array);

    fgets(str_array,sizeof(str_array),stdin);//从标准输入中读入字节数-1个字节的字符
    fputs(str_array,stdout);

    char *ptr_string = "helloworld";
    printf("ptr_string len:%d\n",strlen(ptr_string));

    printf("==============strcpy=================\n");
    char str_array2[20] = {'\0'};
    strcpy(str_array2,ptr_string);
    printf("str_array2:%s\n",str_array2);
    printf("--------------------------\n");

    char str_array3[10] = {"1234567890"};
    strncpy(str_array3,ptr_string,7);//将ptr_str的前7位拷贝进去
    printf("str_array3:%s\n",str_array3);

    printf("===============strcmp===============\n");

    char *ptr_string_1 = "helloworld";
    char *ptr_string_2 = "helloworld";

    printf("value:%d\n",strcmp(ptr_string_1,ptr_string_2));//等于0相等，大于零，
即前一个大于后一个，小于0,前一个小于后一个，按ascii码比较
    char *ptr_string_3 = "nuli";
    printf("value:%d\n",strcmp(ptr_string_3,ptr_string_2));

    printf("=============strcat=================\n");
    char str_array4[10] = "hello";
    strcat(str_array4,str_array3);
    printf("str_array4:%s\n",str_array4);

    printf("============strchr=================\n");//判断字符串中是否包含字符，返
回的是这个字符的指针(从左往右边)
    char *ptr_string_4 = strchr(str_array4,'l');
    printf("ptr_string4:%s,%c\n",ptr_string_4,*ptr_string_4);
```

```
    char *ptr_string_5 = strrchr(str_array4,'l');//从右往左寻找
    printf("ptr_string4:%s,%c\n",ptr_string_5,*ptr_string_5);

    if(strchr(ptr_string_5,'o')!=NULL){
        printf("ptr_string_5 contains o!\n");
    }else{
        printf("ptr_string_5 not contains o!\n");
    }

    printf("==================strstr===========\n");//字符串中是否包含另外一个字符
串
    char *str9 = "hello";
    char *str10 = "first helloworld";
    if(strstr(str10,str9) != NULL){
        printf("str10 contains str9\n");
    }else{
        printf("str10 not contains str9\n");
    }

    printf("=================strtok============\n");

    char str[] ="hello:zzf:hello:tom";
    int counter = 0;
    char *p = strtok(str,":");
    while(p!=NULL){
        counter++;
        printf("%s",p);
        p = strtok(NULL,":");
    }

    printf("===============memset,memcpy==========");//内存清空和拷贝
    char src[] = "i love you";
    char des[] = "you love me";
    memset(des,0,sizeof(des));//清空或者初始化
    memcpy(des,src,sizeof(des)-1);//拷贝des字节数-1,最后一个符号用于存储结束符号
    return 0;
}
```

*自己需要思考？自己如何实现C的库函数？*

我们来实现几个吧？学习要善于思考啊。

```
#include <stdlib.h>
#include <printf.h>
#include <assert.h>


int mystrlen(char *s1);

void teststrcat();

char* mystrcat(char* dest, const char* str);
```

```
char* mystrstr(const char* dest, const char* str);

void testfindStr();

int main()
{
    char* s1 = "jesson";
    int ret = mystrlen(s1);
    printf("字符串长度为%d", ret);
    teststrcat();
    testfindStr();
    getchar();
    return 0;
}

void testfindStr() {
    const char* s1 = "abcdefgh";
    const char* s2 = "ab";
    char* ch = mystrstr(s1, s2);
    if (ch != NULL)
    {
        printf("%s\n", ch);
    }
    else
    {
        printf("not exist\n");
    }
}

void teststrcat() {
    char s1[20] = "abcde";
    const char* s2 = "fghil";
    printf("%s\n", mystrcat(s1, s2));
}

int mystrlen(char *s1) {
    int count = 0;
    while (*s1!='\0')
    {
        count++;
        *s1++;
    }
    return count;
}

//char* strcat(char* dest, const char* str)
char* mystrcat(char* dest, const char* str)
{
    assert(dest);
    assert(str);
    char *ret = dest;
```

```
    while (*dest)
    {
        dest++;
    }
    while (*dest++ = *str++)
    {
        ;
    }
    return ret;
}

//char* strstr(const char* dest, const char* str)
char* mystrstr(const char* dest, const char* str)
{
    char* ptr = dest;
    char* p1 = NULL;
    char* p2 = NULL;
    while (*dest)
    {
        p1 = ptr;
        p2 = str;
        while (*p1++ == *p2++)
        {
            p1++;
            p2++;
            if (*p2=='\0')
            {
                return ptr;
            }
        }
        ptr++;
    }
    return NULL;
}
```

# *结构体*

C 数组允许定义可存储相同类型数据项的变量，结构体是 C 语言中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

定义：

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

如何使用？简单回忆

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <string.h>


//结构体是一种构造数据类型
//把不同的数据类型整合起来成为一个自定义的数据类型

struct Man {
    //成员
    char name[20];
    int age;
};
void main() {
    //初始化结构体的变量
    //第一种方式
    struct Man m1 = {"Jack",21};
    printf("%s, %d \n", m1.name, m1.age);
    //第二种方式
    struct Man m2;

    m2.age = 23;
    //m2.name = "rose"; 不能这样赋值
    strcpy(m2.name, "rose");
    //或者
    //sprintf(m2.name, "Jason");

    //不能再赋值
    //m1 = {};//类似JavaScript字面量赋值，只能在变量声明时赋值
    printf("%s, %d", m2.name, m2.age);
    getchar();
}
```

结构体的嵌套：

```
struct Teacher
{
    char name[20];
```

```
};

struct Student
{
    char name[20];
    int age;
    struct Teacher t;
};

void main() {
    //字面量的方式
    struct Student s1 = { "jack" ,21,{"Jeason"} };
    printf("%s, %d, %s\n", s1.name, s1.age, s1.t.name);
    system("pause");
}
```

结构体指针：

```
struct Teacher
{
    char name[20];
};

struct Student
{
    char name[20];
    int age;
    struct Teacher t;
};

void main() {
    //字面量的方式
    struct Student s1 = { "jack" ,21,{"Jeason"} };
    printf("%s, %d, %s\n", s1.name, s1.age, s1.t.name);
    system("pause");
}
```

# *文件I/O*

- 打开文件

可以使用 **fopen( )** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：

```
FILE *fopen( const char * filename, const char * mode );
```

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

| 模式 | 描述 |
| --- | --- |
| r | 打开一个已有的文本文件，允许读取文件。 |
| w | 打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。 |
| a | 打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。 |
| r+ | 打开一个文本文件，允许读写文件。 |
| w+ | 打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。 |
| a+ | 打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。 |

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

- 关闭文件

为了关闭文件，请使用 fclose( ) 函数。函数的原型如下：

```
 int fclose( FILE *fp );
```

如果成功关闭文件，**fclose( )** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。

C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

- 写入文件

下面是把字符写入到流中的最简单的函数：

```
int fputc( int c, FILE *fp );
```

函数 **fputc()** 把参数 c 的字符值写入到 fp 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。您可以使用下面的函数来把一个以 null 结尾的字符串写入到流中：

```
int fputs( const char *s, FILE *fp );
```

函数 **fputs()** 把字符串 **s** 写入到 fp 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。您也可以使用 **int fprintf(FILE *fp,const char *format, ...)** 函数来写把一个字符串写入到文件中。

- 读取文件

下面是从文件读取单个字符的最简单的函数：

```
int fgetc( FILE * fp );
```

**fgetc()** 函数从 fp 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。下面的函数允许您从流中读取一个字符串：

```
char *fgets( char *buf, int n, FILE *fp );
```

函数 **fgets()** 从 fp 所指向的输入流中读取 n - 1 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。

- 二进制 I/O 函数

下面两个函数用于二进制输入和输出：

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements,
FILE *a_file);                size_t fwrite(const void *ptr, size_t
size_of_elements, size_t number_of_elements, FILE *a_file);
```

这两个函数都是用于存储块的读写 - 通常是数组或结构体。

案例1：仿照zookeeper组件实现配置文件的读写功能。参见项目配置文件项目

案例2：文件加密/解密 我使用的是异或，同学们自己在实现一种 想象aes？