C语言提高笔记

Day1

1、数据类型

- 编译器指定出的数据类型，为了更好的分配内存：其本质是：**固定大小内存块的别名**

2、C语言标准

- ANSI美国国家标准协会制定出来的标准，在89年制定出第一套C89标准

3、typedef使用

- 主要用途：给类型起别名
- 可以简化struct关键字
- 可以区别数据类型
- 提高代码移植性——在不同的标准下迁移C89&C99

4、void的使用

- 无类型，不可创建变量，无法分配内存
- 限定函数返回值
- 限定函数中参数列表
- void*万能指针，可以不需要强制类型转换 给其他指针赋值

5、sizeof的使用

- 本质：不是函数，而是一个操作符
  - 当统计类型占的内存空间的时候，必须加上 小括号
  - 当统计变量占空间的时候，可以不加小括号
- 返回类型是无符号类型 unsigned int ——无符号（正数）有符号（0、正数、负数）
- 可以统计数组的长度
  - 数组名称若果在参数列表中，会退化为指针，指向数组的首元素地址。（指向数组的第一个元素)

6、变量的修改方式

- 直接修改—>`int a = 10; a = 20;`
- 间接修改
  - 通过指针对内存进行修改

```
int a = 10;
int *p =&a;
*p = 20;
```

- 对自定义数据类型进行修改

```
struct Teacher
{
    int a;//0-3
    char b;//4-7
    char c;//8-11
    int d;//12-15
};
struct Teacher T1 = {10,"a","20",30};
T1.a = 20;//直接修改
struct Teacher *p = &T1;
*P->a = 2000;
char *pT = p;
*(int*)(pT + 12);
*(int*)((int*)pT+3);
```

7、内存分区

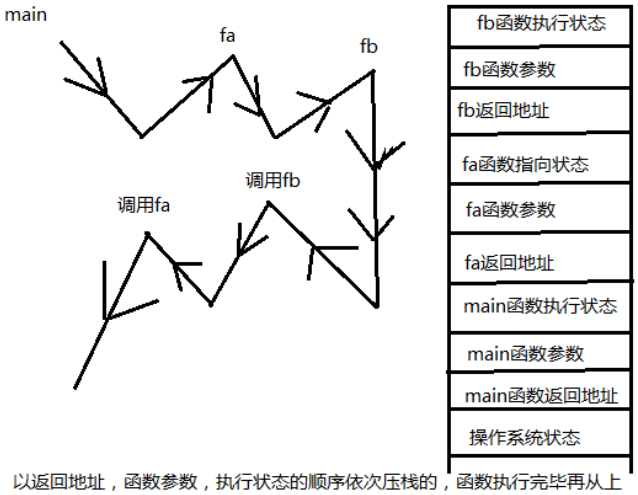
- 运行前
  - 代码区
    - 共享区（比如字符数组，可能公用一块内存地址）
    - 只读区（只能读取，不能修改）
  - 数据区
    - data 已经初始化的全局变量，静态变量，常量
    - bss未初始化的全局变量，静态变量，常量
- 运行后
  - 栈区
    - 属于先进后出的数据结构
    - 由编译器管理数据开辟内存和释放内存
    - 变量的生命周期在该函数结束后自动释放掉
  - 堆区
    - 容量远远大于栈
    - 没有先进后出这样的数据结构
    - 由程序员管理开辟和管理释放（malloc开辟，free释放）
    - 记住手动释放开辟的要手动释放，开辟和释放的时候如果用到指针，主要置空，避免野指针

8、栈区

- 不要返回局部变量的地址，因为局部变量在函数执行后就释放了，我们没有权限取操作释放后的内存。

9、堆区

- 在堆区开辟的数据，记得手动开辟，手动释放
- 注意事项
  - 如果在主调函数中没有给指针分配内存，那么被调函数中需要利用高级指针给主调函数中指针分配内存
  - 主调函数和背调函数
    - 主调函数—>主动发生函数去调用别人（被调函数）的函数
    - 被调函数—>被别人（主调）调用的函数。
    - 主动与被动的关系。现在有A、B两个函数，A函数调用了B函数，那么，A函数就是主调函数，B函数就是被调函数。



以返回地址，函数参数，执行状态的顺序依次压栈的，函数执行完毕再从上到下依次弹栈。

1、main函数中，可以在栈区、堆区、全局区开辟空间，这些内存空间都可以在fa和fb函数中使用

结论：可以被fa fb使用

2、fb函数中，在栈上开辟的内存空间，不能在fa和main函数中使用，因为出了fb函数就被析构掉了。

3、fb函数中，在堆区开辟的内存空间，可以在fa和main函数中使用。

4、fb函数中，在全局区开辟的内存空间，可以在fa和main函数中使用。

10、数据区

- 放入静态变量、全局变量、常量
- static和extern区别
  - static静态变量，编译阶段分配内存，只能在当前文件内使用，只初始化一次
  - extern全局变量，C语言默认的全局变量前都隐藏的加了该关键字
- const修饰的变量
  - 全局变量
    - 直接修改 失败
    - 间接修改 失败，原因：放在常量区，受到保护
  - 局部变量
    - 直接修改 失败
    - 间接修改 成功 原因：放在栈上
    - 伪常量，不可以初始化数组（C++进行了强化）
- 字符串常量
  - 不同的编译器可能有不同的处理方式
  - ANSI没有制定出标准
  - 有些编译器可以修改字符串常量，有些不可以
  - 有些编译器将相同的字符串常量看成同一个（共用同一块内存地址）

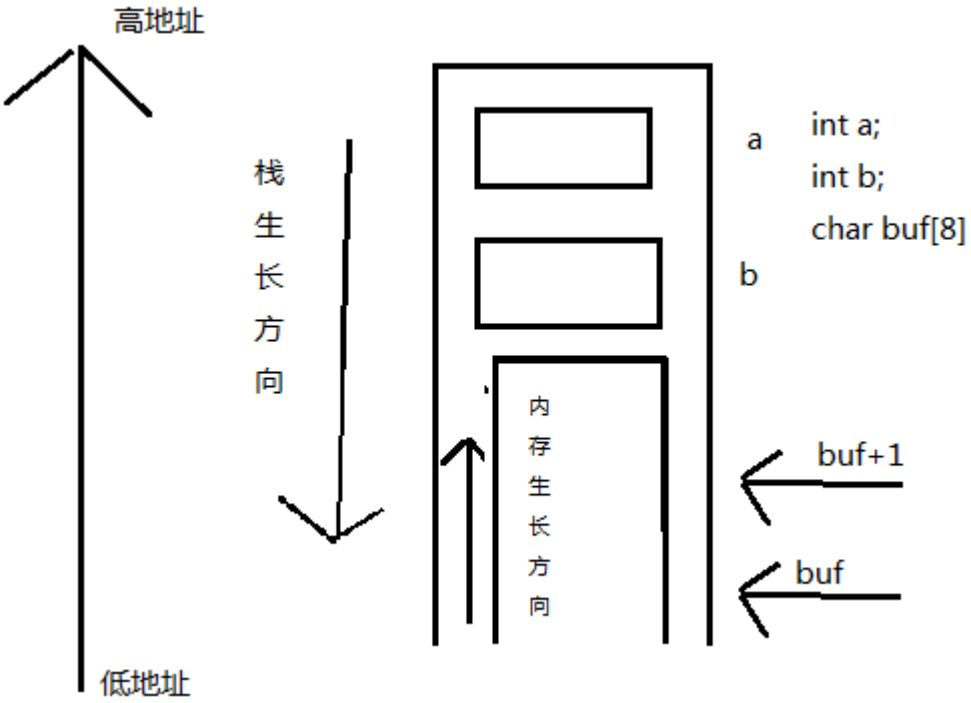
Day2

1、函数调用流程

- 宏函数
  - 在一定程度会比普通函数效率高，普通函数会有出栈入栈的时间开销
  - 将比较繁琐短小的函数 写成宏函数，直接跑源码。
  - 特点：以空间换时间
- 调用惯例
  - 函数调用过程：
    - 函数的返回地址
    - 函数的参数
    - 临时变量
    - 保存的上下文：包括函数调用后西药保持不变的寄存器
  - 主调函数和被调函数都必须有一致的约定，才可以正确的调用函数，这个约定我们称为调用惯例
  - 调用惯例包含的内容：出栈方，参数的传入顺序、函数名称的修饰
  - C和C++下默认的调用惯例为：cdecl
  - cdecl
出栈方	参数传递	名字修饰
函数调用方	从右至左参数入栈	下划线 + 函数名
  -

2、栈的生长方向以及内存存储方式

- 生长方向
  - 栈低——高地址
  - 栈顶——低地址
- 内存存储方式
  - 高位字节数据——高地址
  - 地位字节数据——低地址
  - 小端对齐



3、空指针和野指针

- 空指针—>本身没有问题但是**不允许向NULL和非法地址拷贝内存**
- 野指针
  - 未初始化的指针——定义就初始化，置为空。
  - malloc后也free了，但是空指针没有置空
  - 指针操作空间超越变量作用域
  - 注意：初始化时置为NULL，释放以后也置为NULL。
- 空指针也可以释放，但是野指针不可以释放

4、*号的相关定义

- 在指针声明时，* 号表示所声明的变量为指针。
- 在指针使用的时，* 号表示操作指针所指向的内存空间
  - *相当于通过地址（指针变量的值）找到指针指向的内存，再操作内存。
  - *放在等号的**左边赋值**（给内存赋值，写内存）。
  - *放在等号的**右边取值**（从内存中取值，读内存）。

5、指针的步长

- 指针变量+1之后，跳跃的字节数量
- 解引用的时候，取的字节数
- 对自定义数据类型进行练习
  - 如果获取自定义数据类型中属性的偏移

- offsetof(结构体，属性)
- 头文件 #include<stddef.h>

6、指针的间接赋值

- 满足条件
  - 1、一个普通变量和一个指针变量（或者一个实参一个形参）
  - 2、建立关系——把实参取地址传给形参
  - 3、通过* 进行赋值——形参间接的修改了实参的值

```
int i = 0;//变量一
int *p = NULL;//变量二
p = &i;//建立关系    指针指向谁，就把谁的地址赋给指针
*p = 20;//通过*操作内存
```

  - 1 2 3 三个条件写在一个函数里面
  - 12写在一块 3单独写在另外一个函数里面，=====>函数调用
  - 1 23写一块=====>C++中的&，来替代*,编译器将23帮程序员写在一起。

7、指针做函数参数的输入输出特性

- 主调函数 被调函数
  - 主调函数可以把**堆区、栈区、全局数据**内存地址传递给被调用函数
  - 被调用函数只能返回**堆区，全局区**数据。
- 输入特性
  - 在主调函数中分配内存，被调函数使用
- 输出特性
  - 被调函数中分配内存，主调函数使用
- 做函数参数的时候要注意：不要轻易改变形参的值，要引入一个辅助指针变量，把形参给接过来.....

8、字符串强化训练

- 字符串是有结束标志\0的，C语言没有字符串，是通过字符数组来实现字符串的。
- 利用三种方式对字符串进行拷贝
  - 利用[]

```
void mystrcpy(char *to,char *from)
{
    int i = 0,len = strlen(from);

    for(i = 0; i <len;++i)
    {
      to[i] = from[i];
    }
    to[len] = '\0';
}
```

- 利用指针

```
void mystrcpy(char *to,char *from)
{
    while(*to != '\0')
    {
       *to = *from;
        *from++;
        *to++;
    }
    *to = 0;
}
```

- 利用while

```
void mystrcpy(char *to,char *from)
{
    char *dest = from;
    char *surce = to;
    if(dest=NULL||surce==NULL)return -1;
    while( *surce++ = *dest++);
}
```

- 利用两种方式对字符串进行反转
  - 利用[]

```
void mystrres(char *str)
{
    int len = strlen(str);
    int start = 0;
    int end = len - 1;
    while(start<end)
    {
       char tmp = str[start];
        str[start] = str[end];
```

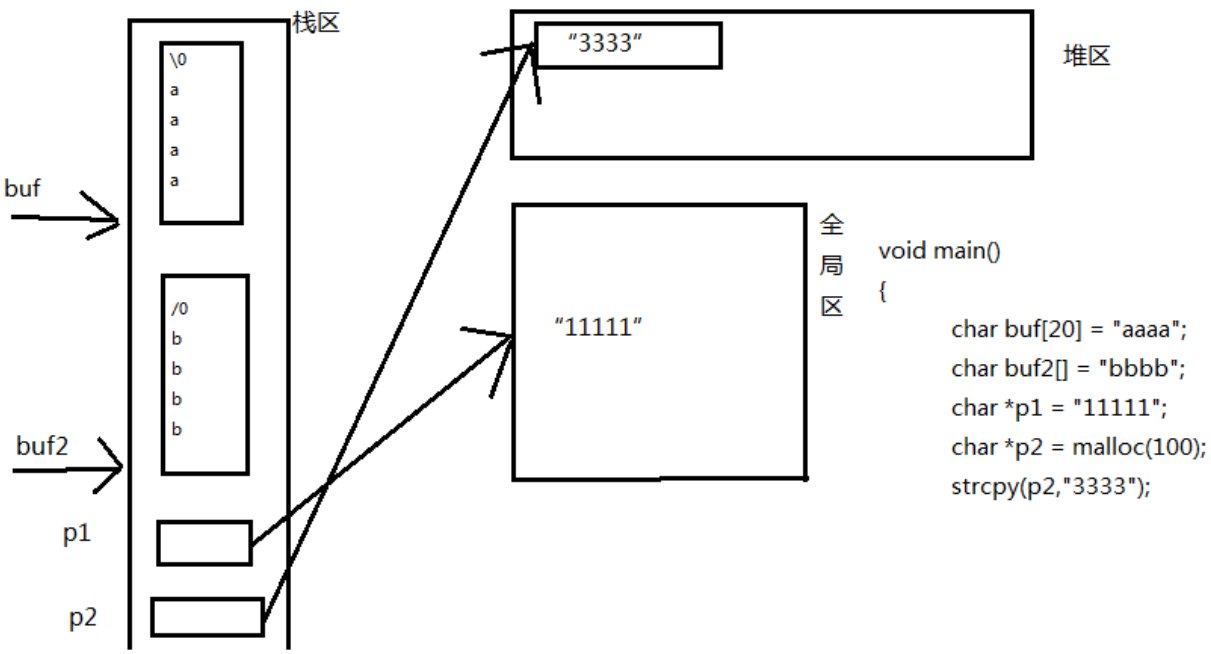
```
        str[end] = tmp;
        start++;
        end--;
    }
}
```

- 利用指针

```
void mystrres(char *str)
{
   int len = strlen(str);
    int *start = str;
    int *end = str + len - 1;
    while(start<end)
    {
       char tmp = *start;
        *start = *end;
        *end = tmp;
        start++;
        end--;
    }
}
```

9、字符串一级指针的内存模型

- 通过字符串的不同初始化方式来理解字符串的形式。



-

10、格式化字符串

- 可以利用sprintf对字符串进行格式化
- sprintf（目标字符串，“格式”,占位参数）

```
int sprintf(char *str, const char *format, ...);
/*功能：
     根据参数format字符串来转换并格式化数据，然后将结果输出到str指定的空间中，直到     出现字符串结束符 '\0'  为止。
参数：
    str：字符串首地址
    format：字符串格式，用法和printf()一样
返回值：
    成功：实际格式化的字符个数
    失败： - 1
*/
//1. 格式化字符串
    char buf[1024] = { 0 };
    sprintf(buf, "你好,%s,欢迎加入我们!", "John");
    printf("buf:%s\n",buf);
//2. 拼接字符串
    memset(buf, 0, 1024);
    char str1[] = "hello";
    char str2[] = "world";
    int len = sprintf(buf,"%s %s",str1,str2);
    printf("buf:%s len:%d\n", buf,len);
//3. 数字转字符串
    memset(buf, 0, 1024);
    int num = 100;
    sprintf(buf, "%d", num);
    printf("buf:%s\n", buf);
    //转成16进制字符串 小写
    memset(buf, 0, 1024);
    sprintf(buf, "0x%x", num);
    printf("buf:%s\n", buf);
```

Day3

1、calloc和realloc

- calloc和malloc一样是在堆区分配内存
- 不同点：calloc会将分配的空间初始化为0.
- realloc重新在堆区分配内存

- 如果分配的内存比原内存要大，这个时候有两种情况出现
- 原有空间后序有足够大的内存闲空间，那么直接在原有空间继续开辟内存，返回原有空间的首地址
- 原有空间后序没有足够大空闲空间，重新会分配一个足够大的空间，并且将原有空间的内存拷贝到新空间下，释放原有空间，将新空间的首地址返回

2、sscanf的使用

- 将已知的字符串通过格式化匹配出有效信息

格式	作用
%*s或*%d	跳过数据
%[width]s	读取指定宽度的数据
%[a-z]	匹配a到z中任意字符（尽可能多的匹配）
%[aBc]	匹配a、B、c中一员，贪婪性
%[^ a]	匹配非a的任意字符，贪婪性
%[^ a-z]	表示读取除a-z以外的所有字符

- 案例
  - 匹配char *pi = “127.0.0.1"将中间数字匹配到num1—num4中

    ```
    ```

  - 字符串char * str = “abcde#momo@163.com”中间的momo匹配出来

3、查找字串

- 实现自己的查找字串的功能，需要在字符串中查找对应的字串，如果有返回字符串第一个字母的位置，如果没有返回-1；

```
```

4、指针的易错点

- 指针容易越界操作
- 指针叠加会不断改变指针指向的空间——如果释放还是原来的空间会报错
- 返回局部变量地址，——运行函数运行完毕后局部变量被释放，再返回就报错
- 不可以释放野指针。——只能将野指针置为NULL。

5、const的使用场景

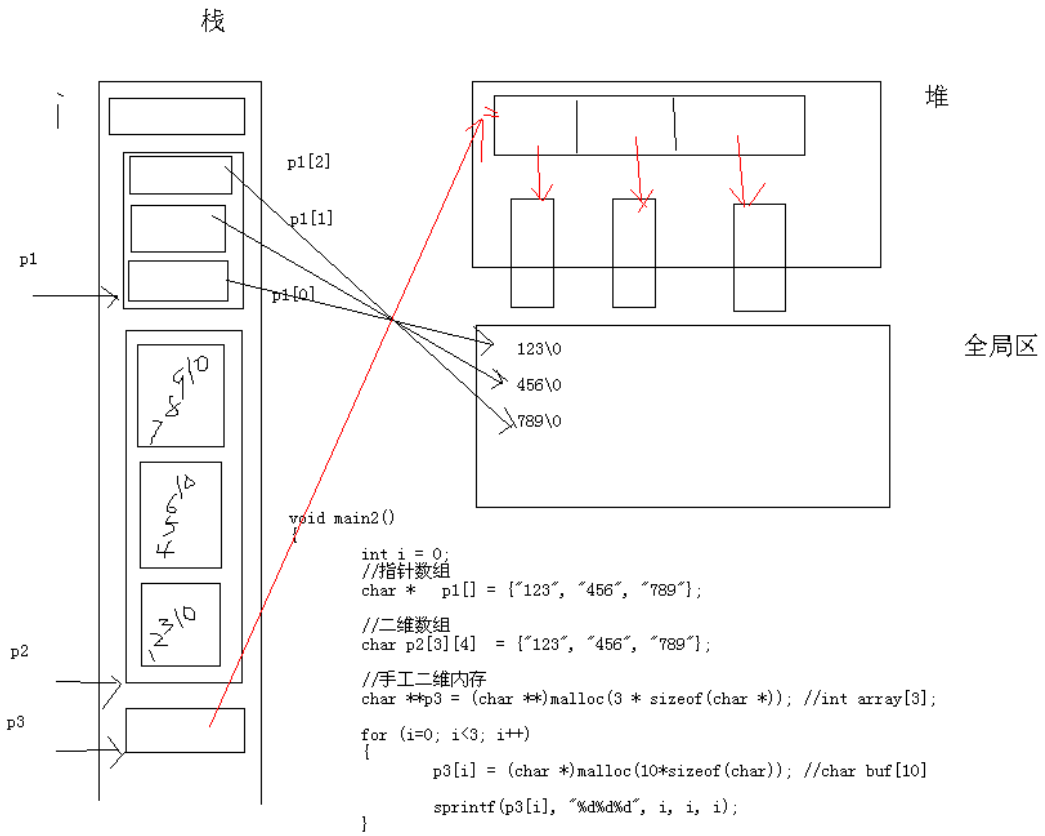
- 用来形式函数中的形参，防止误操作。

- 清楚的分清参数的输入和输出特性
- 要看const放在*的左边还是右边，看const修饰的是指针变量，还是修饰指针所指向的内存空间变量。

```
int main()
{
const int a;  //
int const b;
const char *c;
char * const d; char buf[100]
const char * const  e ;
return 0;
}
Int func1(const )
/*初级理解：const是定义常量==》const意味着只读
含义：
第一个第二个意思一样 代表一个常整形数
第三个 c是一个指向常整形数的指针(所指向的内存数据不能被修改，但是本身可以修改)
第四个 d 常指针（指针变量不能被修改，但是它所指向内存空间可以被修改）
第五个 e一个指向常整形的常指针（指针和它所指向的内存空间，均不能被修改）
*/
```

6、二级指针做函数参数的输入输出特性

- 三种内存模型



- 输出特性
  - 在主调函数分配内存，被调函数使用。

    - 在堆区创建
    - 在栈上创建
- 输出特性
    - 在被调用函数中分配内存，主调函数使用。

7、二级指针练习—文件读写

- 需求：从文件中读取数据，并且将数据放到堆的数组中
    - char **pArray = malloc(sizeof(char *)*len)
- 获取有效行数
    - 将文件光标置为文件首`fseek(file,0,SEEK_SET)`
- 读取文件数据并且放入到pArray中
- 显示数组pArray
- 释放数组pArray

```
```

8、位运算

描述	符号	变化
按位取反	~	0变1 1变1
按位与	&	全1为1 一0为0
按位或	|	全0为0 一1为1
按位异异或	^	相同为0 不同为1

9、位移运算

- 左移 << 。（ <<x等价于 乘以2的x次方，用0填充位 ）
- 右边>>。（ >>等价于 除以2的x次方，如果有符号，不同的机器可能有不同的结果 ）

Day4

1、一维数组名称

- 本质并不是一个指针
- 数组首元素的地址和数组地址是不同的两个概念。首元素地址：第一个元素的地址，数组地址：整个数组所在的地址。
- 有两种情况
    - 对数组名称取sizeof（得到连续内存空间的大小，即数组整体大小 = 类型 * 个数 ）

    - 对数组名称取地址，获取的指针步长是整个数组长度
- 除两种特殊情况外，都是指向数组中首元素的地址 的指针。
- 数组名——指针常量，指针的指向不可以修改
- 如果将数组名传到函数参数中，为了提高可读性通常写为`int arr[]`
- 访问数组元素的时候，下标可以为负数，也可以通过指针的偏移来访问数组
- 数组首元素的地址和数组的地址值相等

2、如何定义数组的指针

- 先定义出数组的类型，再通过类型创建数组指针`typedef int(Aarray_type)[];`
- 先定义数组指针的类型，再创建数组指针变量`typedef int(*Array_type)[];`
- 直接创建数组指针变量```int(*Parr)[] = &arr;

3、二维数组名称

- 除了两种特殊情况外，都是指向第一个一维数组的指针
- 两种特殊情况
    - sizeof统计整个二维数组长度
    - 对数组名取地址，int (*p)[3][3] = &arr;
- 二维数组做函数参数传递方式
    - `void printArray(int p[][3],int row,int col);`
    - `void printArray(int p [3][3],int row,int col);`
    - `void printArray(int(*p)[3],int row,int col);`
- 数组指针 指针数组区别
    - 数组指针：是一个指向数组的指针int （ *p ）[10];
    - 指针数组：是一个存放指针的数组int *p[10];

4、数组指针的排序

- 选择排序
    - 假设排序规则为从小到大
    - 先选定一个最小值下标为i，通过j = i + 1找到真实最小值下标
    - 判断计算出的真实最小值下标 和开始认定的i是否相等，如果不相等，交换i和min下标的两个元素
    - 对指针数据从大到小排序

5、结构体基本使用

- 如果有typedef定义结构体，那么后面跟着的单词是类型的别名
- 没有typedef，定义结构体，后面跟着的单词是一个结构体变量

- 结构体数组
    - 在栈上开辟空间
    - 在堆上开辟空间

6、结构体赋值问题以及解决

- 系统提供赋值操作是简单的值拷贝，最字节拷贝——浅拷贝
- 如果属性中有指向堆区的内容，在释放期间会导致堆区重复释放，并且还有内存泄漏
- 解决方案：利用深拷贝，手动赋值

7、结构体嵌套一级指针

- 设计结构体 `struct Person{char *name,int age};`
- 在堆区创建结构体指针数组 `malloc(sizeof(struct Person *) * 3);`
- 给每个结构体也分配到堆区
- 给每个结构体的名字分配到堆区
- 打印数组中所有人的信息
- 释放堆区内存

Day5

1、结构体嵌套二级指针练习

2、结构体偏移量

- 可以利用offsetof来计算结构体中属性的偏移
- 也可以通过地址的相加运算 计算偏移量
- 接头体嵌套结构体

3、内存对齐

- 内存对齐原因：以空间换时间
- 对于自定义数据类型对齐规则：
    1. 从第一个属性开始 偏移为0
    2. 从第二个属性开始，地址要放在 该类型整数倍 与对齐模数比 取小的值 的整数倍上
    3. 所有的属性都计算结束后，整体再做二次对齐，整体需要放在属性中最大类型 与对齐模数比 取小的值的整数倍上
- 如何查看对齐模数
    - #pragma pack (show)
    - 默认对齐模数 是 8，可以将对齐模数改为2的n次方
- 当结构体嵌套结构体的时候，只需要看结构体中最大数据类型就可以了

4、文件读写回顾

- 按照字符进行读写
  - 写文件 fputc
  - 读文件 fgetc
  - 文件结尾EOF END OF FILE
- 按行读写
  - 些文件fwrite
    - 参数1 数据地址 参数2 块大小 参数3 块个数， 参数4 文件指针
  - 读文件 fread
- 按格式化读写
  - 写文件fprintf
  - 读文件fscanf
- 设计位置读写
  - fseek（文件指针，偏移，起始位置 SEEK_SET SEEK_END SEEK_CUR)
  - rewind （文件指针）将文件光标置首
  - error宏 全局变量， preeor打印宏的提示错误信息

5、文件读写注意事项

- 当按照字符的方式读文件的时候，通常利用判断EOF获取是否读到文件尾
- 当对自定义数据类型写入文件时，不要将指针写入到文件里，要将指针指向的内容写入

6、配置文件读写

- 需求：将文件中的有效内容截取出来，并且放入到一个键值对的数组中
  - struct ConfigInfo{char key[64];char value[64]};
  - 获取有效行数getlin（ ）
  - 判断当前行是否有效
  - 分析数据parseFile
    - 将有效数据放入到数组中，数组在堆区开辟
  - 根据key获取value getInfoByKey
  - 释放内存 freeSpace

Day6

1、链表的基本概念

- 链表的引出

    - 数组有缺陷
    - 静态空间，一旦分配内存就不可以动态扩展，要不分配不够，要不分配过多
    - 对于数组头部进行插入和删除效率很低
- 链表的组成
    - 链表是由节点组成的
    - 节点由数据域和指针域组成
    - struct LinkNode{int num; struct LinkNode *next;}
- 链表的分类
    - 方式一 静态链表 动态链表
    - 方式二 单向链表 双向链表 单向循环链表 双向循环链表

2、静态链表和动态链表

- 静态链表 创建在栈上
- 动态链表 创建在堆区

3、链表的基本应用

- 带头结点链表 好处在于 头结点永远都是固定的
- 初始化链表 `struct LinkNode *pHeader = init LinkList();`
- 遍历链表 `void foreach_LinkList(struct LinkNode * pHeader);`
- 插入链表 `void insertLinkeList(struct LinkNode *pHeader,int oldval,int newval)`
    - 在oldval前插入 newval,ruguo没有oldval就进行尾插入
- 删除链表 `void delete LinkNode(struct LinkNode *pHeader,int val)`
    - 用户提供的有效数据 删除掉
    - 无效数据 直接return
- 清空链表
    - void clear LinkList(struct LinkNode *pHeader)
    - 将所有 有数据的节点释放掉
- 销毁链表
    - `void destroy LinkList(struct LinkNode*pHeader)`
    - 将整个链表都释放掉

4、函数指针的定义

- 先定义出函数类型，再通过类型定义出函数指针
    - `typedf void(FUNC_TYPE)`
    - `FUNC_TYPE *pFunc = func`
- 先定义函数指针类型，再定义函数指针
    - `typedef void(*FUNC_TYPE)();`
    - `FUNC_TYPE pFUNC = func;`

- 直接定义函数指针变量
  - `void(*pFunc)() = func`
- 函数指针和指针函数的区别
  - 函数指针：是指向函数的 指针
  - 指针函数：函数的返回值是一个指针 的 函数
  - 扩展

```
//一个指向整型数据的指针
int *p;
//一个指针的指针，它指向的指针指向一个整型数据
int **p;
//一个有十个指针的数组，该指针指向整型数据
int *p[10];
//一个指向有十个整型数据数组的指针
int (*p)[10];
//就一个函数（不是函数指针），该函数有一个整型参数，返回值为一个指向整型的指针
int *p(int);
//一个函数指针，该函数有一个整型的参数，返回值为整型类型
int (*p)(int);
//一个有十个指针的数组，该数组中的指针指向一个函数，该函数有一个整型参数并返回一个整型数
int (*p[10])(int);
/*
1.一个指向整型数据的指针
2.一个指针的指针，它指向的指针指向一个整型数据
3.一个有十个指针的数组，该指针指向整型数据
4.一个指向有十个整型数据数组的指针
5.就一个函数（不是函数指针），该函数有一个整型参数，返回值为一个指向整型的指针
6.一个函数指针，该函数有一个整型的参数，返回值为整型类型
7.一个有十个指针的数组，该数组中的指针指向一个函数，该函数有一个整型参数并返回一个整型数
*/
```

- 函数指针的数组定义
  - `void(*pFunc[3])()`

5、函数指针做函数参数（回调函数）

- 提供一个通用函数，可以打印任意的数据类型

6、回调函数案例

- 提供一个函数，打印任意类型的数组
- 提供一个查找数组中元素的函数

7、作业

- 提供一个函数，对任意类型的数组进行排序，排序规则，选择排序

Day7

1、链表作业

- 反转链表 通过3个赋值指针变量实现链表的反转
- 统计链表的长度 `int size_LinkList(struct LinkNode *pHeader)`

2、回调函数案例作业

- 对任意数据的数组进行排序

3、预处理指令

- 头文件包含#include
  - 注意“ ” <> 区别
  - <> 系统头文件 ； “ ” 自定义头文件
- 宏定义
  - 不重视作用域
  - 可以利用#undef卸载宏
  - 宏常量 没有数据类型
  - 宏函数 注意表达式完整性
- 条件编译
  - 测试存在 #ifdef
  - 测试不存在 #ifndef
  - 自定义条件测试 #if
- 特殊宏
  - 编译所在文件 —FILE—
  - 编译所在行号 —FLNE—
  - 编译日期 —DATE—
  - 编译时间 —TIME—

4、静态库配置

- 创建项目——配置属性——常规——配置类型——静态库
- 重新生成项目，创建出后缀名为.lib的静态库文件
- 测试静态库

5、动态库配置流程

- 静态库的优缺点

    - 优点：生成的exe程序包含了静态库中的内容，与静态库无瓜葛
    - 缺点：浪费资源，更新发布比较麻烦
- 动态库
    - 运行阶段采取链接函数
    - 配置流程：创建项目——配置属性——常规——配置类型——动态库
    - 程序生成解决方案，生成.dll .lib库文件
    - 导入函数 只能在当前项目下使用
    - 导出函数 可以在外部使用
    - `__declspec(dllexport)int mySub(int a,int b);`
    - 测试 引入`#pargma comment(lib,"./modll.lib")`

6、递归函数

- 函数调用自身，必须有结束条件退出循环
- 案例
    - 实现字符串逆序遍历
    - 实现斐波那契数列

7、面向接口编程

- 实现公司中编程方式
- 甲方和乙方商定好接口，分别实现自己的功能

Day8

1、数据结构的基本概念

- 算法
    - 5个特性：输入 输出 有穷 确定 可行性
- 数据结构分配
    - 逻辑结构
        - 集合 元素之间没有关系，都是平等，不去讨论
        - 线性1:1关系，除了第一元素没有前驱，最后一个元素没有后继，其他元素都是唯一的前驱和唯一的后继
        - 树形 1：n
        - 图形 n：n
    - 物理结构
        - 顺序存储
        - 链式存储

2、动态数组

- 初始化
- 插入数据
  - 判断是否已经满了，如果满了，动态开辟内存
  - 插入数据
- 遍历数据
  - 将数组总的每个元素进行遍历，利用到回调函数
- 删除数据
  - 按位置进行删除
  - 按值进行删除
- 销毁数组
- 分文件编写

3、单向链表

- 节点结构体
  - 数据域 void*
  - 指针域 struct LinkNode *Next
- 链表结构体
  - struct LinkNode pHeader 头结点
  - int m_Size链表长度
- 初始化链表
- 插入链表
- 遍历链表
- 删除节点
  - 按照位置进行删除
  - 按值进行删除 利用连个辅助指针变量
- 清空链表
- 返回链表长度
- 销毁链表

Day9

1、单向链表—企业级版本

- 节点 只维护指针域 不维护数据域
- 用户数据 需要预留出4个字节空间 给底层链表使用
- 对链表提供 对外接口
- 初始化链表
- 插入链表
- 遍历链表

- 删除链表
- 销毁链表

2、栈的顺序存储——利用数组

- 栈 属于先进后出的数据结构
- 初始化栈 init
- 入栈 push
- 出栈 pop
- 访问栈顶元素 top
- 返回栈大小 size
- 判断是否为空 isEmpty
- 销毁栈 destroy

3、栈的链式存储——利用链表

- 和顺序存储的对外结构完全一致

4、栈的应用，就近匹配案例

- 扫描字符串
- 准备栈
- 如果左括号 入栈
- 如果是右括号
    - 栈是否为空
    - 如果栈不为空 出栈
    - 如果栈为空 立即停止 并报错
- 遍历所有字符后，判断栈是否为空
    - 如果为空 没有问题
    - 如果不为空 报错，左括号没有匹配到对应的右括号
- 销毁栈

5、中缀表达式和后缀表达式

- 中缀是符合人类习惯
- 后缀符合计算机使用
- 中缀如何转为后缀表达
- 计算机如果利用后缀表达式进行计算

Day10

1、队列——顺序存储

- 利用写好的动态数组实现 顺序存储的队列数据结构
- 对外提供的接口
- 初始化队列
- 入队
- 出队
- 返回队头
- 返回队尾
- 销毁队列
- 判断队列是否为空

2、 队列链式存储

- 和顺序存储对外接口一致
- 利用链表方式实现
- 实现出一种 符合先进先出的数据结构

3、数的基本概念

- 根节点 没有前驱的节点
- 叶子节点 没有后继节点
- 双亲 前驱节点
- 孩子 后继节点
- 节点的度 该节点的直接后继的数量
- 数的度 节点的度中 最大的值
- 数的高度 数的层数

4、二叉树性质

- 性质1：在二叉树的第i层上至多有2i-1个节点（i>0)
- 性质2：深度为k的二叉树至多有2k -1个节点（k > 0)
- 性质3：对于任何一颗二叉树，若度为2的节点数有n2个，则叶子数（n0）必定为n2 + 1 （即n0 = n2 +1）
- 满二叉树 节点数量为2k -1
- 完全二叉树 除了最后一层节点，上面是一颗满二叉树，最后一层的节点尽量往左靠
- 性质4：具有n个节点的完成二叉树的深度必维log2n+1
- (如 log2 (15) 点击 15 log / 2 log =)
- 性质5：完全二叉树，若从上至下，从左至右编号，则编号为i的节点，其左孩子编号为2i，其右孩子编号必须为2i + 1；双亲的编号必须为i/2(i=1时为根，除外)

5、二叉树的递归遍历

- 利用二叉树递归性可以将二叉树进行递归遍历
- 先序遍历 先根 再左 再右

- 中序遍历 先左 再根 再右
- 后序遍历 先左 再右 再根
- 利用代码实现二叉树递归遍历

6、二叉树编程

- 利用递归性质 求出二叉树中的叶子的数量
- 求出二叉树的高度
- 利用递归特性 拷贝出二叉树

7、二叉树的非递归遍历

- 利用栈实现
- 首先将每个节点都设置一个标志，默认表示为假，根据节点的状态进行如下流程
    - 将根节点压入栈中
    - 进入循环：只要栈中元素个数大于0，进行循环操作。
        - 弹出栈顶元素
        - 如果这个栈顶元素标志为真，输出这个元素并且执行下一次循环
        - 如果栈顶元素标志为假，将节点的标志设置为真
        - 将该节点右子树、左子树、根压入栈中
        - 执行下一次循环

8、插入排序

- 外层大循环，从下标1开始
- 内层小循环，满足一定条件后，根据后移
- 将数据插入到相应的位置上 j + 1位置。

强化训练

1、有一个字符串开头或结尾含有n个空格（”

abcdefgdddd ”），欲去掉前后空格，返回一个新字符串。

- 要求1：请自己定义一个接口（函数），并实现功能；70分 要求2：编写测试用例。30分 int trimSpace(char *inbuf, char *outbuf);

2、实现字符串奇偶分离

- 有一个字符串”1a2b3d4z”,； 要求写一个函数实现如下功能， 功能1：把偶数位字符挑选出来，组成一个字符串1。valude；20分 功能2：把奇数位字符挑选出来，组成一个字符串2，valude 20 功能3：把字符串1和字符串2，通过函数参数，传送给main，并打印。 功能4：主函数能测试通过。 int getStr1Str2(char *souce, char *buf1, char *buf2);

3、键值对（”key = valude”）字符串，在开发中经常使用

- 要求1：请自己定义一个接口，实现根据key获取valude；40分 要求2：编写测试用例。30分 要求3：键值对中间可能有n多空格，请去除空格30分 注意：键值对字符串格式可能如下： “key1 = valude1” “key2 = valude2 “key3 = valude3” “key4 = valude4” “key5 = “ “key6 =“ “key7 = “

  int getKeyByValude(char *keyvaluebuf, char *keybuf, char *valuebuf, int * valuebuflen); int main() { getKeyByValude(“key1 = valude1”, ” key1”, buf, &len); }