

C 语言高级进阶

你懂 c 语言，我不信。

Auth : 王桂林

Mail : guilin_wang@163.com

Org : 能众软件

Web : <http://edu.nzhsoft.cn>

版本信息：

版本	修订人	审阅人	时间	组织
V1.0	王桂林		2016.04.02	
V1.1	王桂林		2016.05.01	能众软件
V1.2	王桂林		2019.05.28	能众软件

更多学习：



1. 数据类型(DataType).....	- 1 -
1.1. 内存.....	- 1 -
1.2. 补码.....	- 1 -
1.2.1. 运算规则.....	- 1 -
1.2.2. 补码特点.....	- 2 -
1.2.3. char(8 位)补码的展示.....	- 2 -
1.3. 数据类型.....	- 2 -
1.3.1. 数据类型.....	- 2 -
1.3.2. 范围计算.....	- 3 -
1.3.3. 数据类型是对内存的格式化.....	- 3 -
1.4. 类型转化.....	- 6 -
1.4.1. 类型转化的原理.....	- 6 -
1.4.2. 隐式转化.....	- 7 -
1.4.3. 显示(强制类型)转化.....	- 7 -
1.5. 练习.....	- 7 -
1.5.1. 下面的代码输出什么?	- 7 -
1.5.2. 以下程序输出什么?	- 8 -
2. 进程空间(Program Space).....	- 9 -
2.1. 进程空间.....	- 9 -
2.2. 进程/程序.....	- 9 -
2.2.1. 程序.....	- 9 -
2.2.2. 进程.....	- 9 -
2.2.3. 进程到程序.....	- 9 -
2.3. 数据在进程空间的存储.....	- 10 -
2.3.1. 示意图.....	- 10 -
2.3.2. 数据在进程空间.....	- 11 -
2.4. 函数的压栈与出栈.....	- 12 -
2.4.1. 普通函数.....	- 12 -
2.4.2. 递归函数.....	- 13 -
2.5. 课堂实战.....	- 13 -
2.5.1. 如下程序中数据存储在哪里.....	- 13 -
2.5.2. 用递归正/逆序打印一个数组.....	- 14 -
3. 数组(Array).....	- 15 -
3.1. 一维数组.....	- 15 -
3.1.1. 本质.....	- 15 -
3.1.2. 初始化.....	- 15 -
3.1.3. 访问.....	- 16 -
3.1.4. 作参数传递.....	- 17 -
3.1.5. 返回堆中一维数组.....	- 18 -
3.1.6. 练习.....	- 18 -

3.2. 二维数组.....	- 22 -
3.2.1. 本质.....	- 22 -
3.2.2. 初始化.....	- 22 -
3.2.3. 访问.....	- 22 -
3.2.4. 线性存储.....	- 24 -
3.2.5. 作参数传递.....	- 25 -
3.3. 数组指针.....	- 25 -
3.3.1. 引入.....	- 25 -
3.3.2. 定义.....	- 25 -
3.3.3. 别名.....	- 26 -
3.3.4. 数组指针与数组名.....	- 26 -
3.3.5. 应用.....	- 26 -
3.4. 多维数组.....	- 27 -
3.4.1. 本质分析.....	- 27 -
3.4.2. 形像描述.....	- 28 -
3.5. 课堂实战.....	- 28 -
3.5.1. 求值?.....	- 28 -
3.5.2. 下面代码的值为多少?	- 28 -
4. 指针(Pointer).....	- 29 -
4.1. 内存编址与变量地址.....	- 29 -
4.1.1. 编址.....	- 29 -
4.1.2. 变量地址.....	- 29 -
4.2. 指针与指针变量.....	- 30 -
4.2.1. 指针的本质.....	- 30 -
4.2.2. 指针变量.....	- 30 -
4.2.3. 运算.....	- 32 -
4.2.4. 课堂实战.....	- 33 -
4.3. 二级指针.....	- 33 -
4.3.1. 定义与初始化.....	- 33 -
4.3.2. 间接数据访问.....	- 34 -
4.3.3. 初始化一级指针.....	- 35 -
4.3.4. 二级指针的步长.....	- 37 -
4.4. 指针数组(字符指针数组).....	- 38 -
4.4.1. 定义.....	- 38 -
4.4.2. 使用.....	- 38 -
4.4.3. 二级指针访问指针数组.....	- 39 -
4.4.4. 常见指针数组.....	- 40 -
4.5. 指针的输入与输出.....	- 41 -
4.6. 堆上一维空间.....	- 41 -
4.6.1. 返回值返回(一级指针).....	- 41 -
4.6.2. 参数返回(二级指针).....	- 41 -
4.7. 堆上二维空间.....	- 42 -

4.7.1. 一级指针作返回值输出.....	- 42 -
4.7.2. 二级指针作返回值输出.....	- 43 -
4.7.3. 三级指针作参数输出.....	- 44 -
4.7.4. 课堂练习.....	- 44 -
4.8. const 修饰指针.....	- 47 -
4.8.1. const 修饰变量.....	- 47 -
4.8.2. const 修饰符.....	- 48 -
4.8.3. const 修饰指针指向.....	- 48 -
4.8.4. 应用(修饰函数参数).....	- 49 -
4.9. 练习.....	- 49 -
4.9.1. 请手写下面代码的输出结果.....	- 49 -
4.9.2. 天生棋局.....	- 50 -
5. 函数(Fucntion).....	- 51 -
5.1. 函数多参返回.....	- 51 -
5.1.1. 引列.....	- 51 -
5.1.2. 正解.....	- 51 -
5.2. 函数指针.....	- 51 -
5.2.1. 函数的本质.....	- 51 -
5.2.2. 函数指针变量定义与赋值.....	- 52 -
5.2.3. 函数指针类型定义.....	- 52 -
5.2.4. 函数类型别名.....	- 53 -
5.2.5. 函数指针调用.....	- 53 -
5.2.6. 函数指针数组.....	- 53 -
5.3. 回调函数.....	- 54 -
5.3.1. 问题引出.....	- 54 -
5.3.2. 回调(函数作参数).....	- 55 -
5.3.3. 本质论.....	- 56 -
5.3.4. qsort.....	- 56 -
5.4. 递归函数.....	- 59 -
5.4.1. 迭代变递归引入.....	- 59 -
5.4.2. 递归公式.....	- 60 -
5.4.3. 书写公式.....	- 60 -
5.4.4. 递向处理.....	- 61 -
5.4.5. 归和处理.....	- 61 -
5.4.6. 返回值递归.....	- 61 -
5.5. 练习.....	- 62 -
5.5.1. (*(void(*) ()) 0)();.....	- 62 -
6. 再论指针与数组.....	- 63 -
6.1. 一级指针与一维数组名.....	- 63 -
6.1.1. 等价条件.....	- 63 -
6.1.2. 便利访问.....	- 63 -

6.2. 二级指针与指针数组名.....	- 63 -
6.2.1. 等价条件.....	- 63 -
6.2.2. 便利访问.....	- 63 -
6.2.3. 通过二级指针申请二维空间.....	- 63 -
6.3. 数组指针与二维数组名.....	- 63 -
6.3.1. 等价条件.....	- 63 -
6.3.2. 便利访问.....	- 63 -
6.4. 关于二级指针与二维数组名.....	- 63 -
6.5. 对数组名的引用.....	- 64 -
6.5.1. 一维数组.....	- 64 -
6.5.2. 二维数组.....	- 64 -
6.6. 小结与练习.....	- 64 -
6.6.1. 指针-数组-函数.....	- 64 -
6.6.2. 请写出右边的示意.....	- 64 -
6.6.3. 传参汇总.....	- 65 -
6.6.4. 写出程序的运行结果.....	- 65 -
7. 基于数组排序(Sort)提高篇.....	- 67 -
7.1. 选择优化.....	- 67 -
7.1.1. 选择原理与步骤.....	- 67 -
7.1.2. 代码实现.....	- 67 -
7.1.3. 优化升级.....	- 67 -
7.2. 冒泡优化.....	- 68 -
7.2.1. 冒泡原理与步骤.....	- 68 -
7.2.2. 代码实现.....	- 68 -
7.2.3. 优化升级.....	- 69 -
7.3. 快速排序.....	- 69 -
7.3.1. 排序原理与步骤.....	- 69 -
7.3.2. 代码实现.....	- 70 -
7.3.3. 逻辑质问.....	- 70 -
7.4. 练习.....	- 71 -
7.4.1. 实现二级排序.....	- 71 -
8. 基于数组的查找(Search).....	- 72 -
8.1. 线性查找.....	- 72 -
8.2. 二分查找.....	- 72 -
8.2.1. 原理步骤.....	- 72 -
8.2.2. 迭代版(Iterative).....	- 72 -
8.2.3. 递归版(Recursive).....	- 73 -
9. 字符串(String).....	- 75 -
9.1. 字符串常量.....	- 75 -
9.1.1. 定义相关.....	- 75 -

9.1.2. c 语言处理字符串.....	- 75 -
9.2. 字符数组.....	- 75 -
9.2.1. 字符数组与字符串.....	- 76 -
9.2.2. 不等价条件.....	- 76 -
9.2.3. 等价条件.....	- 77 -
9.2.4. N 系列字符串函数.....	- 77 -
9.3. 基础字符串操作函数及自实现.....	- 77 -
9.3.1. strlen.....	- 77 -
9.3.2. strcpy/strncpy.....	- 78 -
9.3.3. strcat/strncat.....	- 78 -
9.3.4. strcmp/strncmp.....	- 78 -
9.3.5. sprintf.....	- 79 -
9.3.6. atoi/itoa.....	- 79 -
9.3.7. strchr/strchr.....	- 81 -
9.3.8. strcspn/strspn.....	- 82 -
9.3.9. strstr/strpbrk.....	- 82 -
9.3.10. strtok.....	- 84 -
9.4. 字符串 trim 系列.....	- 87 -
9.4.1. 引例.....	- 87 -
9.4.2. 去除右空格.....	- 87 -
9.4.3. 去除左空格.....	- 87 -
9.4.4. 去除全空格.....	- 88 -
9.4.5. 去除任意字符.....	- 88 -
9.5. 递归逆置字符串.....	- 88 -
9.5.1. 非递归实现.....	- 88 -
9.5.2. 递归逆置.....	- 89 -
9.6. 内存操作函数.....	- 90 -
9.6.1. memset.....	- 91 -
9.6.2. memcpy.....	- 92 -
9.6.3. memmove.....	- 92 -
9.6.4. memcmp.....	- 93 -
9.6.5. memchr.....	- 93 -
9.7. 练习.....	- 93 -
9.7.1. 返回 s2 在 s1 中最后一次出现的位置.....	- 93 -
9.7.2. 清空字符串中的制表符.....	- 93 -
9.7.3. 格式化文件.....	- 93 -
9.7.4. 判断字符串 s1 是否是 s2 循环移位所得.....	- 93 -
9.7.5. 求字符频度.....	- 93 -
9.7.6. 压缩字符串.....	- 93 -
9.7.7. 实现 atoi 功能.....	- 93 -
9.7.8. 实现 itoa 功能.....	- 93 -
10. 数据结构(双向循环链表).....	- 95 -

10.1. 双向循环链表存在的意义.....	- 95 -
10.2. 链表构成.....	- 95 -
10.2.1. 节点图示.....	- 95 -
10.2.2. 节点码示.....	- 95 -
10.2.3. 空链表示意.....	- 96 -
10.2.4. 非空链表示意.....	- 96 -
10.3. 双向循环链表静态形式.....	- 96 -
10.3.1. 节点代码.....	- 96 -
10.3.2. 静态模拟.....	- 96 -
10.4. 双向循环链表的操作.....	- 97 -
10.4.1. 创建.....	- 97 -
10.4.2. 插入.....	- 98 -
10.4.3. 遍历.....	- 99 -
10.4.4. 查找.....	- 99 -
10.4.5. 删除节点.....	- 100 -
10.4.6. 排序.....	- 101 -
10.4.7. 销毁.....	- 102 -
10.5. 链表应用-贪吃蛇.....	- 103 -
10.5.2. MVC 架构分析.....	- 103 -
10.5.3. 其它.....	- 103 -
10.6. 练习.....	- 104 -
10.6.1. 用双向循环链表实现读文件到内存.....	- 104 -
11. 动态库与静态库(Library).....	- 105 -
11.1. 函数库.....	- 105 -
11.1.1. 库存在的意义.....	- 105 -
11.1.2. 库的划分.....	- 105 -
11.1.3. 库的命名规则.....	- 105 -
11.1.4. 静/动态库特点.....	- 105 -
11.2. linux 库存放的路径.....	- 106 -
11.2.1. 使用特点.....	- 106 -
11.2.2. 查看一个文件的库的使用情况.....	- 106 -
11.3. 标准库的使用.....	- 107 -
11.3.1. libm.a.....	- 107 -
11.3.2. lib.so.....	- 107 -
11.3.3. 比较.....	- 108 -
11.4. 自定义库的生成与使用.....	- 110 -
11.4.1. 多文件编程.....	- 110 -
11.4.2. 静态库的生成与使用.....	- 111 -
11.4.3. 动态库的生成与使用.....	- 112 -
11.5. gcc 的编译选项.....	- 113 -
11.6. Linux 平台编译 sqlite3.....	- 114 -
11.6.1. 编译动态库.....	- 114 -

11.6.2. 编译成静态库：	- 115 -
12. 配置文件读写(File).....	- 116 -
12.1. 文件回顾.....	- 117 -
12.1.1. 文件基础.....	- 117 -
12.1.2. 文件操作.....	- 117 -
12.2. 链表回顾.....	- 117 -
12.2.1. 节点.....	- 117 -
12.2.2. 创建.....	- 117 -
12.2.3. 插入.....	- 117 -
12.2.4. 求长.....	- 117 -
12.2.5. 查找.....	- 117 -
12.2.6. 删除.....	- 117 -
12.2.7. 排序.....	- 117 -
12.2.8. 销毁.....	- 117 -
12.3. 字符串处理回顾.....	- 117 -
12.3.1. 常规处理.....	- 117 -
12.3.2. 去空处理.....	- 117 -
12.4. 实现登录系统.....	- 117 -
12.4.1. 链表节点设计.....	- 117 -
12.4.2. 读文件并处理字符串.....	- 118 -
12.4.3. 登录.....	- 118 -
13. 本地数据库(SQLite).....	- 119 -
13.1. 本章综旨.....	- 119 -
13.2. SQLite 简介.....	- 119 -
13.3. 数据库基本概念.....	- 119 -
13.4. sqlite 官网.....	- 120 -
13.4.1. 下载.....	- 120 -
13.4.2. Navicat for SQLite.....	- 120 -
13.5. SQLite 管理操作.....	- 120 -
13.5.1. 进入 sqlite3 交互模式.....	- 120 -
13.5.2. 创建销毁表.....	- 123 -
13.5.3. 列类型.....	- 123 -
13.5.4. 列约束.....	- 123 -
13.5.5. 插入与查询.....	- 125 -
13.5.6. 排序.....	- 126 -
13.5.7. 修改与删除记录.....	- 126 -
13.5.8. 常用函数.....	- 127 -
13.5.9. 备份与恢复.....	- 127 -
13.6. Vs/Qt+SQLite 配置.....	- 127 -
13.6.1. window 中 lib /dll 的关系.....	- 127 -
13.6.2. 生成 lib 文件.....	- 128 -

13.6.3. 建立三方库相关的文件夹.....	- 129 -
13.6.4. 配置当前工程的 pro 文件.....	- 129 -
13.6.5. 测试环境.....	- 129 -
13.7. C/C++操作 SQLite.....	- 130 -
13.7.1. 常用 API 详解.....	- 130 -
13.7.2. 实战增/删/改查.....	- 132 -
13.7.3. 获得表 sqlite3_get_table.....	- 134 -
13.7.4. 防注入.....	- 135 -
14. C 语言中的范型(Generic).....	- 136 -
14.1. void 系列.....	- 136 -
14.2. macro 系列.....	- 136 -
14.3. union.....	- 136 -
15. 附录(Append).....	- 137 -
15.1. ascii 码表.....	- 137 -
15.2. c 运算符优先级.....	- 138 -
15.3. SVN 配置与启动.....	- 139 -
15.3.1. 官方下载地址.....	- 139 -
15.3.2. 创建服务配置文件.....	- 139 -
15.3.3. 配置配置文件.....	- 139 -
15.3.4. 启动服务.....	- 139 -
15.3.5. 创建与删除服务.....	- 140 -
15.3.6. 更新与提交.....	- 140 -
15.4. netplan.....	- 140 -
15.4.1. 配置.....	- 140 -
15.4.2. 启动.....	- 141 -
15.4.3. 其它.....	- 141 -
15.5. TestSql.....	- 141 -

负数	取反+1	负数的计算机表示	符号位为 1
----	------	----------	--------

1.2.2.补码特点

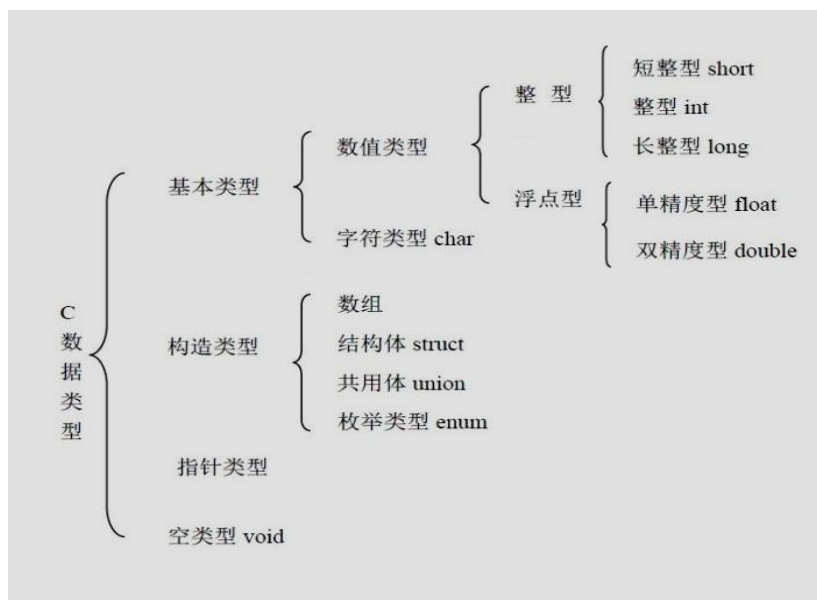
- 0 是补码是 0
- 符号位同普通位一并参与运算
- 补码可以使加减乘除，全部变加法

1.2.3.char(8 位)补码的展示

无符号数	8 位数据的排列组合	有符号数
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
	• • • •	
	0111 1111	127
	1000 0000	-128
	• • • •	
253	1111 1100	-3
254	1111 1110	-2
255	1111 1111	-1

1.3.数据类型

1.3.1.数据类型



1.3.2.范围计算

1.3.2.1. 整型

类型	位数(32 位机)	符号	范围	科学计数
char	8	unsigned	0-255	$0-2^8-1$
		[signed]	-128-+127	-2^7-+2^7-1
short	16	unsigned	0-65535	$0-2^{16}-1$
		[signed]	-32768-+32767	$-2^{15}-+2^{15}-1$
int	32	unsigned		$0-2^{32}-1$
		[signed]		$-2^{31}-+2^{31}-1$
long	32	unsigned		$0-2^{32}-1$
		[signed]		$-2^{31}-+2^{31}-1$
long long	64	unsigned		$0-2^{64}-1$
		[signed]		$-2^{63}-+2^{63}-1$

1.3.2.2. 实型

类型	位数(32 位机)	范围简算	精算	有效位数
float	32	$[2^{-128}, 2^{127}]$	$[10^{-38}, 10^{38}]$	6-7
		$[-2^{127}, -2^{-128}]$	$[-10^{38}, -10^{-38}]$	6-7
double	64	$[2^{-2048}, 2^{2047}]$	$[10^{-308}, 10^{308}]$	15-16
		$[-2^{2047}, -2^{-2048}]$	$[-10^{308}, -10^{-308}]$	15-16

1.3.3.数据类型是对内存的格式化

数据类型提供了，申请内存单元的大小和访问规则。数据类型是架设在线性内存上的一种逻辑关系。

```
#include <stdio.h>
int main()
{
    char a = 0xff;
    printf("%d\n",a);

    unsigned b = 0xff;
    printf("%u\n",b);
    return 0;
}
```

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

```
typedef struct _stu
{
    char sex;
    short age;
    int score;
}Stu;

typedef union _un
{
    char name[8];
    char sex;
    double score;
}Un;

int main(int argc, char* argv[])
{
    void *p = malloc(8);

    // 以 char * + 8 的方式理解
    char *cp = (char*)p;
    for(int i=0; i<8; i++)
    {
        cp[i]= 'a'+i;
    }
    for(int i=0; i<8; i++)
    {
        printf("cp[%d] = %c\n",i,cp[i]);
    }
    // 以 short * + 4 的方式理解
    short *sp = (short*)p;
    for(int i=0; i<4; i++)
    {
        sp[i]= 100 +i;
    }
    for(int i=0; i<4; i++)
    {
        printf("sp[%d] = %d\n",i,sp[i]);
    }
}
```

```
// 以 int * + 2 的方式理解
int *ip = (int*)p;
for(int i=0; i<2; i++)
{
    ip[i]= 100 +i;
}
for(int i=0; i<2; i++)
{
    printf("ip[%d] = %d\n",i,ip[i]);
}
// 以 float * + 2 的方式理解
float *fp = (float*)p;
for(int i=0; i<2; i++)
{
    fp[i]= 100.12344 +i;
}
for(int i=0; i<2; i++)
{
    printf("fp[%d] = %.4f\n",i,fp[i]);
}

// 以 double * 的方式理解
double *dp = (double*)p;
*dp = 0.12345;

printf("double * fp = %.6f\n",*dp);

Stu *stp = (Stu*)p;
stp->sex = 'x';
stp->age = 24;
stp->score = 100;
printf(" sex = %c age = %d score = %d\n",stp->sex,stp->age,stp->score);

Un *up = (Un*)p;

strcpy(up->name,"china");
printf("name = %s\n",up->name);

up->sex = 'x';
printf("sex = %c\n",up->sex);

up->score = 1234.1234;
```

```
printf("score = %f\n",up->score);

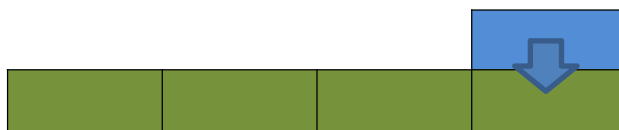
return 0;
}
```

1.4.类型转化

1.4.1.类型转化的原理

1.4.1.1. 小数据赋给大变量

不会造成数据的丢失，系统为了保证数据的完整性，还提供了符号扩充行为。

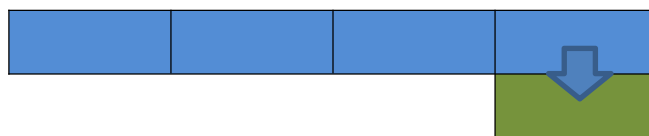


```
#include <stdio.h>

int main(void)
{
    char a = 0xFF; //a = 0x7f    16进制数据是不能表示正负的
    int b = a;
    printf("%d\n",b);
    return 0;
}
```

1.4.1.2. 大数据赋给小变量

会发生 Truncate(截断行为)，有可能会造成数据丢失。



```
#include <stdio.h>

int main(void)
{
    int a = 0xff; //a = 256;
}
```



```
char b = a;
printf("%d\n",b);
return 0;
}
```

1.4.2.隐式转化

1.4.2.1. 整型提升

在 32 位机中，所有位为低于 32 的整型数据，在运算过程中先要转化为 32 位的整型数据，然后才参与运算。

1.4.2.2. 混合提升

- First, if either operand is **long double**, the other is converted to **long double**.
- Otherwise, if either operand is **double**, the other is converted to **double**.
- Otherwise, if either operand is **float**, the other is converted to **float**.
- Otherwise, the **integral promotions** are performed on both operands;
- Otherwise, if either operand is **unsigned int**, the other is converted to **unsigned int**.
- Otherwise, both operands have type **int**

```
int main(void)
{
    unsigned int a = 1;
    int b = -100;
    printf("a +b = %u\n",a+b);
    return 0;
}
```

1.4.3.显示(强制类型)转化

```
int a = 7;
int b = 3;
float c = (float)a/b
```

1.5.练习

1.5.1.下面的代码输出什么？

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

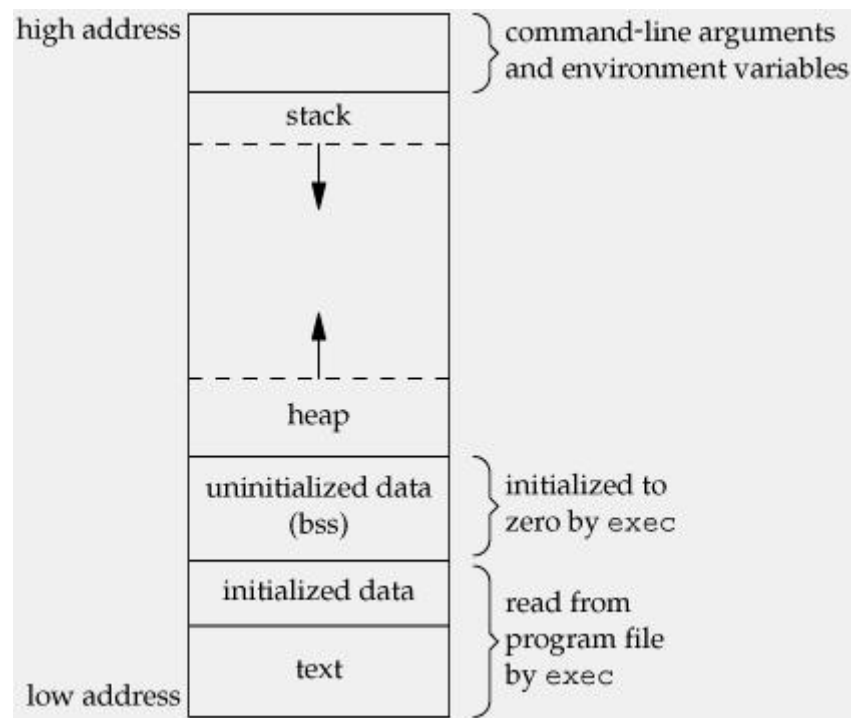
```
void foo(void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b>6)? puts(">6"):puts("<=6");
}
int main()
{
    foo();
    return 0;
}
```

1.5.2. 以下程序输出什么？

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[1000];
    int i;
    for(i=0; i<1000; i++)
    {
        a[i] = -1 -i;
    }
    printf("%d\n",strlen(a));
    return 0;
}
```

2.进程空间(Program Space)

2.1.进程空间



2.2.进程/程序

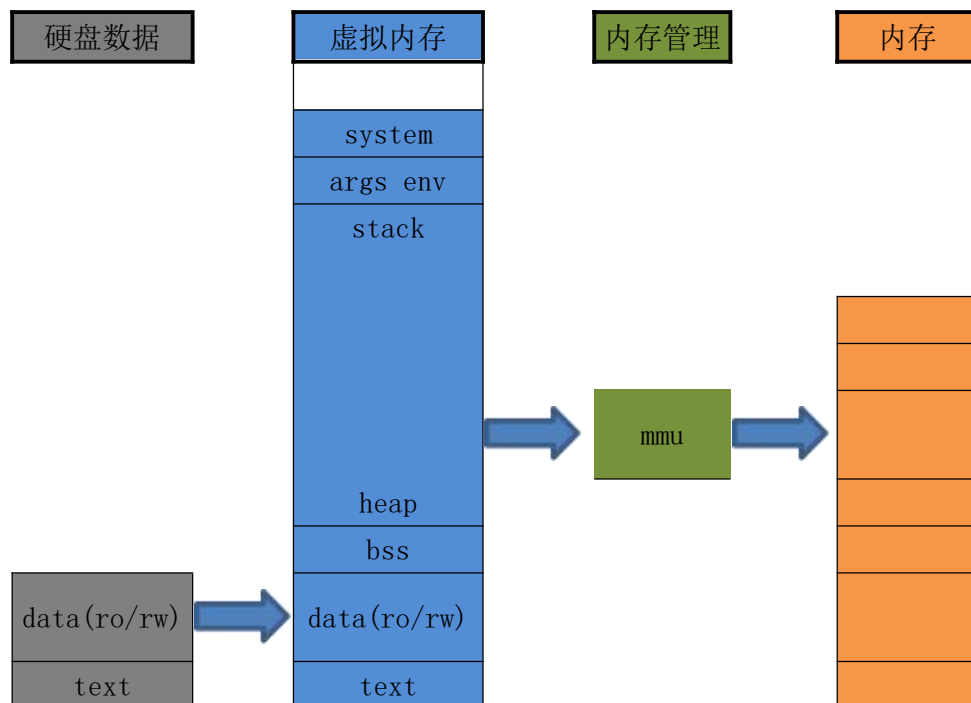
2.2.1.程序

源文件经编译器，编译后的可执行文件。程序是一个静态的概念。程序中包含 2 个区域，分别是：text /initial data

2.2.2.进程

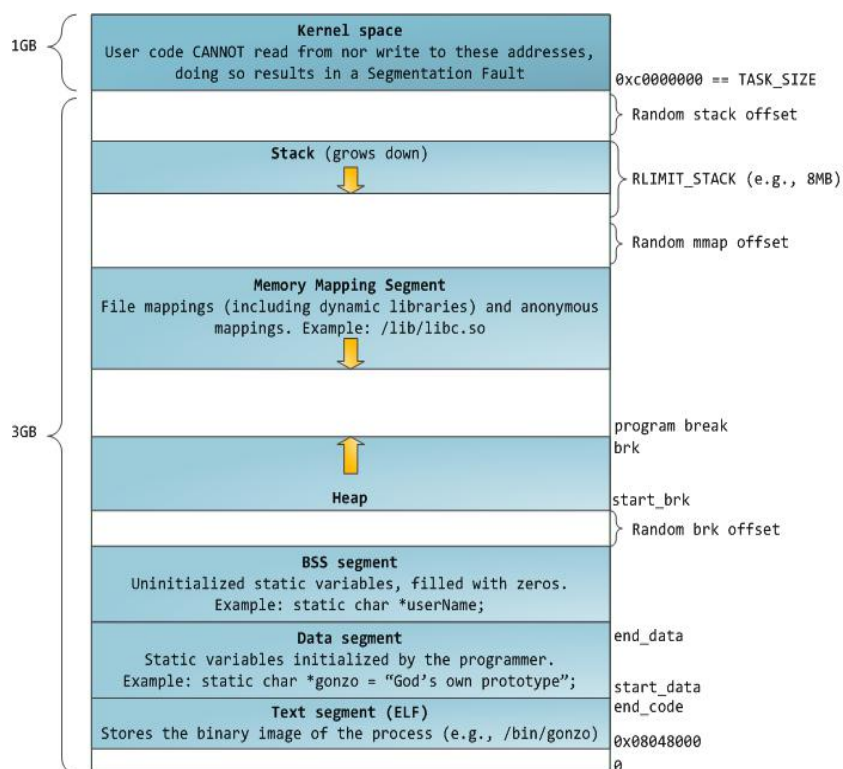
进程，被操作系统加载至运行结束的过程。进程是一个动态的概念。进程包含 5 个区域，分别是：text /initial data /uninitial data /heap /stack

2.2.3.进程到程序





2.3.数据在进程空间的存储

2.3.1.示意图



2.3.2.数据在进程空间

数据量分布		进程空间			注释
		command args			命令行参数
普通局部变量		stack			栈
					
		dynamic lib			加载动态库区
					
malloc 动态申请空间		heap			堆
普通全局 静态(全局或 局部)	未初始化	data	uninit	bss	未初始数据段
	初始化		init	rw	初始化数据段的读写段
				ro	初始化数据段的只读段
常量		text			代码段

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

```
int a;
static int b;

int c = 4;
static int d = 6;
int main(void)
{
    int array[10] = {0,1,2,3,4,5,6,7,8,9,10};

    char * p = "china";

    int var = 5;

    static int s = 6;

    func();
    return 0;
}
void func()
{
    static int si;
    int var;
}
```

2.4.函数的压栈与出栈

2.4.1.普通函数

auto 类型的大将军变量，随用随消时怎么会事呢？今天就通过，函数压栈与出栈的操作，来给大家演示。

```
#include <stdio.h>

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
```

```
{
    int m = 5;
    int n = 6;
    swap(m,n);
    return 0;
}
```

2.4.2.递归函数

```
#include <stdio.h>

int getAge(int n)
{
    if(n == 5)
        return 2;
    else
        return getAge(n+1) +2;
}

int main(void)
{
    int age = getAge(1);
    printf("age = %d\n",age);
    return 0;
}
```

2.5.课堂实战

2.5.1.如下程序中数据存储在哪里

读程序，分别说出数据在进程空间的中存储方式。

```
#include <stdio.h>

char*fa()
{
    char *pa = "123456";
    // pa 指针在栈区，"123456"在常量区，该函数调用完后指针变量 pa 就被释放了
    char *p = NULL;
    // 指针变量 p 在栈中分配 4 字节
    p = (char*)malloc(100);
    // 本函数在这里开辟了一块堆区的内存空间，并把地址赋值给 p
    strcpy(p, "wudunxiong 1234566");
}
```

```
//把常量区的字符串拷贝到堆区
return p;
//返回给主调函数 fb(), 相对 fa 来说 fb 是主调函数, 相对 main 来说,
//fa(),fb()都是被调用函数
}
char*fb()
{
    char *pstr = NULL;
    pstr = fa();
    return pstr;//指针变量 pstr 在这就结束
}
void main()
{
    char *str = NULL;
    str = fb();
    printf("str = %s\n",str);
    free(str);
    //防止内存泄露, 被调函数 fa()分配的内存存的值通过返回值传给主调函数,
    // 然后主调函数释放内存
    str = NULL;//防止产生野指针
    return 0;
}
```

2.5.2.用递归正/逆序打印一个数组

```
void reverseArr(int *parr,int i,int len)
{
    if(i != len-1)
        reverseArr(parr,i+1,len);
    printf("%d\n",parr[i]);
}

int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    reverseArr(arr,0,10);
    return 0;
}
```

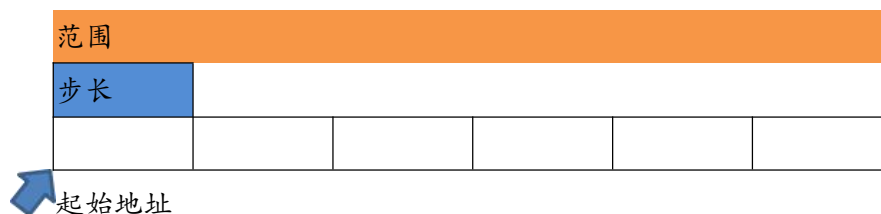

3.数组(Array)

3.1.一维数组



3.1.1.本质

数组是用于存储相同数据类型数据，且在内存空间连续的一种数据结构类型。数组三要素。



```
类型: type [N]
定义: type name[N]
大小: sizeof(type [N]) 或 sizeof(name)
```

3.1.2.初始化

两个凡是。省略初始化

```
int array[10] = {1,2,3}; //部分初始化
int array2[10] = {[3] = 10};
int array1[10] = {0}; //清零
```

```
int array3[10] = {1,2,3,4,5,6,7,8,9,0,1,2}; //越界不检
```

3.1.3.访问

数组名是数组的唯一标识符，数组的每一个元素都是没有名字的，必须依数组名来访问。数据名有两重含义：

3.1.3.1. 数组名作整体访问

数组名作整体访问，就两种常见的形式，一是对数组 `sizeof` 求大小，二是对数组取地址得到数组指针。

```
#include <stdio.h>

int main(void)
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,0};
    //int[10] arr;
    printf("sizeof(arr) = %d\n",sizeof(arr));
    printf("&arr    = %p\n",&arr);
    printf("&arr+1  = %p\n",&arr+1);

    int(*pa)[10] = &arr;
    printf("pa    = %p\n",pa);
    printf("pa+1  = %p\n",pa+1);
    return 0;
}
```

3.1.3.2. 数组名作起始地址访问成员

```
name[i] == *(name+i) == i[name]
```

```
#include <stdio.h>

int main(void)
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,0};

    printf("arr[0]   = %d\n",arr[0]);
    printf("*(arr+0) = %d\n",*(arr+0));
    printf("arr     = %p\n",arr);
    printf("arr +1  = %p\n",arr+1);
}
```

```
printf("&arr[0]    = %p\n",&arr[0]);  
printf("&arr[0] +1 = %p\n",&arr[0]+1);  
  
return 0;  
}
```

3.1.4.作参数传递

作参数传递，旨在传递三要素。

```
#include <stdio.h>  
void selectSort(int *p,int n)  
{  
    for(int i=0; i<n-1; i++)  
    {  
        for(int j=i+1; j<n; j++)  
        {  
            if(p[i]>p[j])  
            {  
                p[i] ^= p[j];  
                p[j] ^= p[i];  
                p[i] ^= p[j];  
            }  
        }  
    }  
}  
  
int main()  
{  
    int array[10] = {9,8,7,6,5,4,3,2,1,0};  
    selectSort(array,10);  
    for(int i=0; i<10 ;i++)  
    {  
        printf("%d\n",array[i]);  
    }  
    return 0;  
}
```

3.1.5.返回堆中一维数组

3.1.5.1. 返回值返回(一级指针)

```
char * allocMem(int n)
{
    char *p = (char*)malloc(n);
    return p;
}
```

3.1.5.2. 参数返回(二级指针)

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

int allocMem(char **p,int n)
{
    *p = (char*)malloc(n);
    return *p== NULL?-1:1;
}

int main(void)
{
    char *p;
    if(allocMem(&p,100)<0)
        return -1;
    strcpy(p,"china");
    printf("%s\n",p);
    free(p);
    return 0;
}
```

3.1.6.练习

3.1.6.1. 合并有序数组使其依然有序

现在数组 `int a[3] = {1,3,5};` `int b[5] = {2,4,6,8,10};` 合并到 `int c[8]` 中去,使其依然有序。

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

int main(void)
```

```
{
    int a[3] = {1,3,5}; int b[5] = {2,5,6,8,10};
    int c[8];
    int i=0,j=0,k=0;
    while(i<3&& j<5)
    {
        if(a[i] < b[j])
            c[k++] = a[i++];
        else (a[i] > b[j])
            c[k++] = b[j++];

        if(i==3)
        {
            while(j<5)
                c[k++] = b[j++];
        }
        if(j==5)
        {
            while(i<3)
                c[k++] = a[i++];
        }

        for(int i=0; i<8; i++)
        {
            printf("%d\n",c[i]);
        }

        return 0;
    }
}
```

3.1.6.2. 求最值

```
#include <stdio.h>

int main(void)
{
    int arr[10] = {1,2,3,10,9,6,7};
    int max = arr[0];
    for(int i=0; i<10; i++)
    {
        if(max<arr[i])
            max = arr[i];
    }
}
```

```
}
printf("max = %d\n",max);
return 0;
}
```

3.1.6.3. 求次最值

```
int main(int argc, char *argv[])
{
    int arr[10] = {94,73,6,77,94,65,54,94,58,71};
    for(int i=0; i<10; i++)
    {
        printf("%d\t",arr[i]);
    }
    putchar(10);

    int max,subm;
    arr[0]>arr[1]? (max = arr[0],subm = arr[1]):(max = arr[1],subm=arr[0]);

    for(int i=2; i<10; i++)
    {
        if(arr[i]>max)
        {
            subm = max;
            max = arr[i];
        }
        else if(arr[i]>subm && arr[i]!=max) // 10 5 10
        {
            subm = arr[i];
        }
    }
    printf("max = %d subm = %d\n",max,subm);
    return 0;
}
```

这样作可以了吗？ 能不能举出反例？ 10 10 5

3.1.6.4. 求次最值(改进)

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

```
int secondMaxOfArray(int *p, int n)
{
    int max, subm;
    max = subm = *p;
    for (int i=1;i<n;i++) {

        if(p[i] > max){
            subm = max;
            max = p[i];
        }
        else {
            if(p[i] < max && p[i]>subm || max == subm)
            {
                subm = p[i];
            }
        }
    }
    return subm;
}

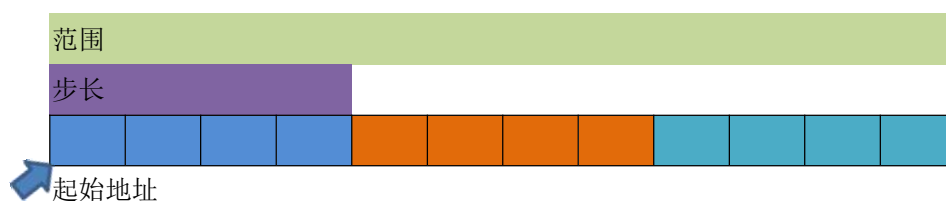
int main()
{
    int arr[] = {10,10,1,4};
    int subm = secondMaxOfArray(arr,4);
    printf("subm = %d\n",subm);
    return 0;
}
```

3.2.二维数组



3.2.1.本质

二维数组的本质是一维数组，只不过，一维数组的成员又是一个一维数组而已。一维数组三要素同样适用：



```
type name[M][N] == type[N] name[M]
int arr[3][4] == int[4] arr[3]
```

3.2.2.初始化

"两个凡是"，省略初始化。

行可以省，列不可以省，部分初始化和清零依然适用

```
int array[2][3] = {[1][2]=3}; //c99
```

3.2.3.访问

数组名，是数组的唯一标识符。

3.2.3.1. 数组名作为整体访问

```
#include <stdio.h>
int main(void)
{
```



```

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

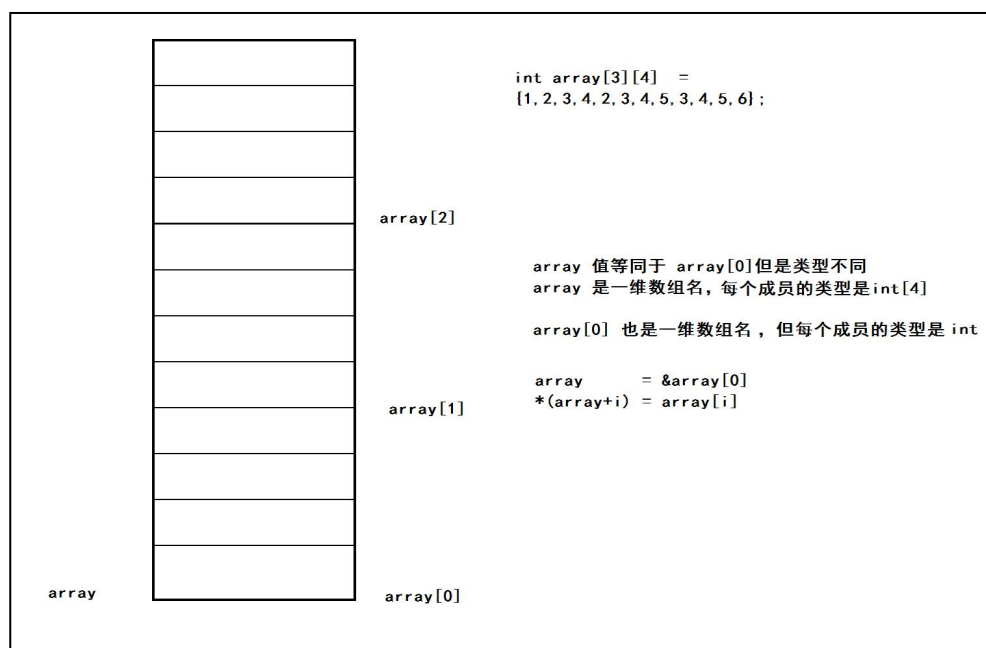
int array[3][4]; //int[4] array[3]    type array[3];    type[3] array;

printf("sizeof(array) = %d sizeof(int[3][4]) = %d\n",
sizeof(array),sizeof(int[3][4]));
printf("&array = %p\n",&array);
printf("&array +1 = %p\n",&array+1);
return 0;
}

```

3.2.3.2. 数组名作起始地址访问成员

如何从 `a` 到 `a[i]` 到 `a[i][j]` 的。



```

#include <stdio.h>
int main(void)
{
    int array[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
    printf("array = %p\n",array);
    printf("array+1 = %p\n",array+1);
}

```

```

printf("array[0]  = %p\n",array[0]);
printf("array[0]+1 = %p\n",array[0]+1);

printf("(array[0]+1) = %d\n",*(array[0]+1)); // *(a+1) a[1]
printf("(array[0]+1) = %d\n",array[0][1]);

//如何 array 过度到 array[0]
printf("*array      = %p\n",*array);
printf("*array + 1  = %p\n",*array + 1);
printf("array[0] + 1 = %p\n",array[0] + 1);

return 0;
}

```

3.2.3.3. 结论

a 表示第 0 行首地址，a+i 表示第 i 行首地址

*(a+i),	a[i],	&a[i][0]表示第 i 行第 0 个元素地址
*(a+i)+j,	a[i]+j,	&a[i][j]表示第 i 行第 j 个元素地址
((a+i)+j)	*(a[i]+j),	a[i][j]表示第 i 行第 j 个元素

3.2.4.线性存储

二维数组在逻辑上是二维的，但是在存储上却是一维的。正是这个特点，也可以用一维数组的方式去访问二维数组的。

```

#include <stdio.h>
int main()
{
    int array[2][3] = {9,8,7,6,5,4};
    for(int i=0;i<2; i++)
    {
        for(int j=0;j<3;j++)
        {
            printf("%d ",array[i][j]);
        }
        putchar(10);
    }
    int *p = (int *)array;
    for(int i=0; i<6;i++)
    {
        printf("%d ",p[i]);
    }
}

```

```
    return 0;
}
```

3.2.5.作参数传递

二维数组本质是常量数组指针，所以跟其对应的形参也应该是数组指针类型的变量。

```
void displayArray(int(*p)[4],int n)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%d ",*(*(p+i)+j));
        }
        putchar(10);
    }
}

int main()
{
    int array[3][4] = {1,2,3,4,2,3,4,5,3,4,5,6};
    displayArray(array,3);
    return 0;
}
```

3.3.数组指针

3.3.1.引入

一次可以移动一个字节指针，称为 `char *` 类型的指针，一次可以移动二个字节的指针，称为 `short*` 类型的指针。

一次可以移动一个数组大小的指针，是什么类型的指针的，又该如何称谓呢？比如二维数组名：`arr[3][4]`；`arr+1` 一次加 1 的大小，就是 `int[4]` 类型的大小，即一个数组的大小。

3.3.2.定义

```
int [N] *pName; => int (*pName)[N];
```

语法解析：“()” 的优先级比“[]” 高，“*” 号和 `pName` 构成一个指针的定义,指针的

类型为 int [N]。

```
#include <stdio.h>

int main(void)
{
    int (*p)[10] = NULL;

    printf("sizeof(p) = %d\n",sizeof(p));
    printf("p   = %p\n",p);
    printf("p+1 = %p\n",p+1);

    return 0;
}
```

3.3.3.别名

```
typedef int (*TYPE)[N];
```

3.3.4.数组指针与数组名

解析	一维数组名	二维数组名
示例	int arr[4];	int arr[3][4];
本质	一级指针	数组指针
引用	&arr 数组指针	&arr 数组指针

3.3.5.应用

3.3.5.1. 二维数组传参

```
void displayArray(int(*p)[4],int n)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%d ",*(p+i)+j));
        }
        putchar(10);
    }
}
```

```
int main()
{
    int array[3][4] = {1,2,3,4,2,3,4,5,3,4,5,6};
    displayArray(array,3); //sizeof(array)/sizeof(int[4])
    return 0;
}
```

3.3.5.2. 一维空间的二维访问

```
int main()
{
    int arr[12] = {1,2,3,4,5,6,7,8,9,10,11,12};

    int (*p)[2] = (int(*)[2])arr;

    for(int i=0; i<sizeof(arr)/sizeof(int[2]);i++)
    {
        for(int j=0; j<2; j++)
        {
            printf("%d\t",p[i][j]);
        }
        putchar(10);
    }
}
```

3.4.多维数组

3.4.1.本质分析

type name[x][y][z] == type [y][z] name[x];

3.4.2.形像描述



3.5.课堂实战

3.5.1.求值？

```
#include <stdio.h>

int main()
{
    int a[5][5];
    int (*p) [4];
    p = a;
    printf("%d \n",&p [4][2] - &a [4][2]);
    return 0;
}
```

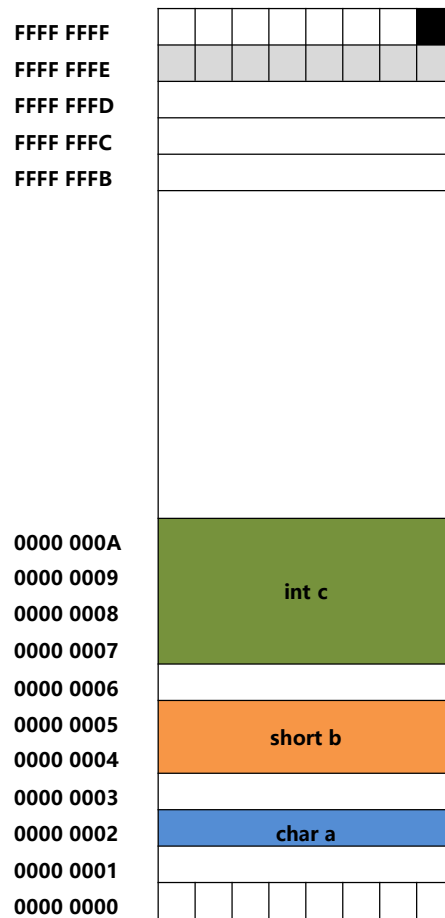
3.5.2.下面代码的值为多少？

```
#include <stdio.h>int
main()
{
    int a[5] = {1,2,3,4,5};
    int *ptr = (int*)&a + 1;
    printf("%d,%d", *(a+1),*(ptr-1)); //2 5
    return 0;
}
```

4. 指针(Pointer)

4.1.内存编址与变量地址

4.1.1.编址



4.1.2.变量地址

对变量取地址，取得是最低位字节的地址。32 位机下，大小均为 4。

```
#include <stdio.h>

int main(void)
{
    char a;
    short b;
```

```
int c;

printf("&a = %p\n",&a);
printf("&b = %p\n",&b);
printf("&c = %p\n",&c);

printf("sizeof(&a) = %d\n",sizeof(&a));
printf("sizeof(&b) = %d\n",sizeof(&b));
printf("sizeof(&c) = %d\n",sizeof(&c));
return 0;
}
```

4.2.指针与指针变量

4.2.1.指针的本质

指针常量的本质，就是一个有类型的地址。指针变量的本质就是，可以存放 4 字节地址，并依据声明类型大小寻址的变量。

```
#include <stdio.h>

int main(void)
{
    int a = 0x12345678;
    printf("%p\n",&a);
    printf("%d\n",*(&a));
    printf("%x\n",*((int*)0x0060FEAC));
    return 0;
}
```

4.2.2.指针变量

内存的地址，即指针(常量)，存放该地址的变量就是指针变量。此变量，必须满足 3 个条件，大小为 4，有类型，区别于其它变量。

type * var;

type 决定了类型(步长), * 表示该变量是指针, var 用于存储地址。

4.2.2.1. 变量大小

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

int main(void)
{
    char *pa; int *pi;
    printf("sizeof(pa) = %d sizeof(pb) = %d\n",
           sizeof(pa),sizeof(pi));
    printf("sizeof(char*) = %d,sizeof(int*) = %d\n",
           sizeof(char*),sizeof(int*));

    char *pm = (char*)malloc(100);
    printf("sizeof(pm) = %d\n",sizeof(pm));

    int (*parr)[10];
    printf("sizeof(parr) = %d\n",sizeof(parr));

    int arr[100];
    printf("sizeof(arr) = %d\n",sizeof(arr));
    printf("sizeof(&arr) = %d\n",sizeof(&arr));
    return 0;
}
```

4.2.2.2. 变量类型

类型, 决定了, 从 var 存放的地址开始的寻址能力。

```
int main(void)
{
    int a = 0x12345678;
    char *pa = &a;
    printf("%x\n",*pa);
    short *ps = &a;
    printf("%x\n",*ps);
    int *pi = &a;
    printf("%x\n",*pi);
}
```

```
return 0;
}
```

4.2.2.3. 引用与解引用

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

int main(void)
{
    int a = 0x12345678;
    printf("%x\n", *a);

    int arr[3];
    printf("arr = %p\n", arr);
    printf("&arr = %p\n", &arr);

    printf("arr+1 = %p\n", arr+1);
    printf("&arr+1 = %p\n", &arr+1);

    printf("*&arr = %p\n", *&arr);
    printf("*&arr + 1 = %p\n", *&arr+1);

    return 0;
}
```

4.2.3.运算

4.2.3.1. 指针大小

内存编址有高低，所以是可能比较大小的。

4.2.3.2. +/-/++

指针常见运算有，加法，减法，本质是地址值+类型的运算，即加 1，为加 sizeof(T) 的大小。

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

int main(void)
{
    int a[10];
```

```
printf("a  = %p\n",a);
printf("a+1 = %p\n",a+1);

printf("&a[9] - &a[4] = %d\n",&a[9] - &a[4]);
printf("(int)&a[9]-(int)&a[4] = %d\n",(int)&a[9] -(int)&a[4]);
return 0;
}
```

4.2.4.课堂实战

如下程序输出什么？

```
#include <stdio.h>
int main()
{
    int a[5] = {1,2,3,4,5};
    int *ptr1 = (int *)(&a + 1);
    int *ptr2 = (int *)((int)a + 1);
    printf("%x, %x",ptr1[-1], *ptr2);
    return 0;
}

#include <stdio.h>
int main(int argc, char **argv)
{
    int a;
    int *p = &a;
    printf("%x, %x\n",p,p+1);
    printf("%x, %x",(int)p,(int)p+1);
    //printf("%x, %x",(void*)p,(void*)p+1);
    return 0;
}
```

4.3.二级指针

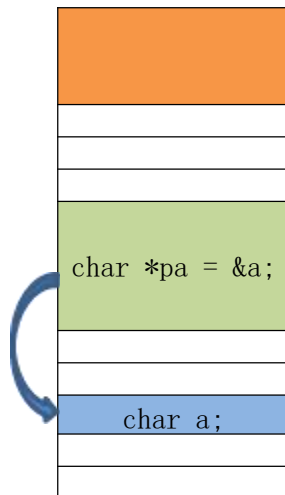
二级指针，是一种指向指针的指针。我们可以通过它实现间接访问数据，和改变一级指针的指向问题。

4.3.1.定义与初始化


```
char* *p= &pa;
```

- 33 -

更多视频：<http://edu.nzhsoft.cn>



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

int main(void)
{
    char c = 'm';
    char *pc = &c;
    char * *pp = &pc;

    printf("c  = %c\n",c);
    printf("*pc = %c\n",*pc);

    printf("&c = %p\n",&c);
    printf("pc  = %p\n",pc);
    printf("**pp = %p\n",*pp);

    printf("***pp = %c\n",**pp);
    return 0;
}
```

4.3.2.间接数据访问

4.3.2.1. 改变一级指针指向的内容

4.3.2.2. 改变一级指针指向

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

```
int main(void)
{
    int a = 10;
    printf("a = %d\n",a);

    int *pa = &a;
    *pa = 200;
    printf("a = %d\n",a);

    int b = 300;
    // pa = &b;
    // printf("*pb = %d\n",b);

    int **pp = &pa;
    *pp = &b;
    printf("*pa = %d\n",*pa);

    return 0;
}
```

4.3.2.3. 改变 N-1 级指针的指向

- 可以通过一级指针，修改 0 级指针（变量）的内容。
- 可以通过二级指针，修改一级指针的指向。
- 可以通过三级指针，修改二级指针的指向。
-
- 可以通过 n 级指针，修改 n-1 级指针的指向。

4.3.3.初始化一级指针

4.3.3.1. 从 FILE *到 sqlite3*

FILE 和 sqlite3 均是描述资源的句柄(即一个结构体)。获取文件句柄的方式是通过返回值，而获取数据库句柄的方式是参数。

FILE *的设计方式，并不是很好。原因是，返回值中要容错出错码。而最合理的方式，是将出错码和返回值分开。

4.3.3.2. 原理剖析

既然，可以改变一级指针的指向问题，也可以初始化一级指针。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
enum
{
    Success, NameErr, SexErr, StrNumErr, ScoreErr
};

struct Stu
{
    char *name;
    char *sex;
    char *strNum;
    float *score;
};

int init(struct Stu **ps)
{
    *ps = (struct Stu*)malloc(sizeof(struct Stu));

    (*ps)->name = (char*)malloc(40);
    if((*ps)->name == NULL)
        return NameErr;
    (*ps)->sex = (char*)malloc(1);
    if((*ps)->name == NULL)
        return SexErr;
    (*ps)->strNum = (char*)malloc(40);
    if((*ps)->name == NULL)
        return StrNumErr;
    (*ps)->score = (float*)malloc(4);
    if((*ps)->name == NULL)
        return ScoreErr;
    return Success;
}

int main(void)
{
    struct Stu * ps = NULL;
    int ret = init(&ps);
    if(ret != Success)
        return -1;
```

```
strcpy(ps->name,"wgl");
*(ps->sex) = 's';
strcpy(ps->strNum,"1001");
*(ps->score)= 300;

printf("name = %s,sex = %c,strNum = %s,score = %.2f\n",
       ps->name,*(ps->sex),ps->strNum,*(ps->score));

return 0;
}
```

4.3.4.二级指针的步长

所有类型的二级指针，由于均指向一级指针类型，一级指针类型大小是 4，所以二级指针的步长也是 4，这个信息很重要。

```
#include <stdio.h>

struct Stu
{
    char c;
    int i;
};

int main(void)
{
    char c;
    char *pc = &c;
    char **ppc = &pc;
    printf("ppc  = %p\n",ppc);
    printf("ppc+1 = %p\n",ppc+1);

    int **ppi = NULL;
    printf("ppi  = %p\n",ppi);
    printf("ppi+1 = %p\n",ppi+1);

    double **ppd = NULL;
    printf("ppd  = %p\n",ppd);
    printf("ppd+1 = %p\n",ppd+1);

    struct Stu ** pps = NULL;
```

```
printf("pps  = %p\n",pps);
printf("pps+1 = %p\n",pps+1);

return 0;
}
```

4.4.指针数组(字符指针数组)

4.4.1.定义

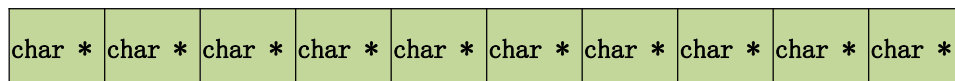
指针数组的本质是数组，数组中每一个成员是一个指针。定义形式如下：

```
char * pArray[10];
```

语法解析：

pArray 先与"[]" 结合，构成一个数组的定义,char *修饰的是数组的内容，即数组的每个元素。

图示：

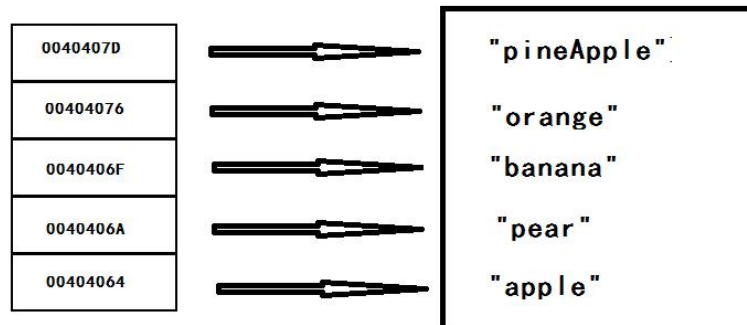


4.4.2.使用

```
char * pArray[] = {"apple","pear","banana","orange","pineApple"};
```

char * pArray[]

data area



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

```
int main(int argc ,char * argv[])
{
```

```
    char * pArray[] =
    {"apple","pear","banana","orange","pineApple"};
```



```
for(int i=0; i<sizeof(pArray)/sizeof(*pArray); i++)
{
    printf("%s\n",pArray[i]);
}
return 0;
}
```

4.4.3.二级指针访问指针数组

4.4.3.1. 指针数组名赋给二级指针的合理性

二级指针与指针数组名等价的原因：

char **p 是二级指针；

char* array[N]; array = &array[0]; array[0] 本身是 char*型；

char **p = array;

```
#include <stdio.h>
int main(int argc ,char * argv[])
{
    char * pArray[] =
    {"apple","pear","banana","orange","pineApple"};

    for(int i=0; i<sizeof(pArray)/ sizeof(*pArray); i++)
        printf("%s\n",pArray[i]);

    char **pArr = pArray;
    for(int i=0; i<sizeof(pArray)/ sizeof(*pArray); i++)
        printf("%s\n",pArr[i]);
    return 0;
}
```

4.4.3.2. 完美匹配的前提(小尾巴 NULL)

数组名，赋给指针以后，就少了维度这个概念，所以用二级指针访问指针数组，需要维度，当然了，也可以不用需要。

```
#include <stdio.h>

int main(int argc ,char * argv[])
{
    //演绎1
    int arr[10] = {1};
```

```
for(int i=0; i<10; i++)
    printf("%d\n",arr[i]);
int *parr = arr;
for(int i=0; i<10; i++)
    printf("%d\n",*parr++);

//演绎 2
char *str = "china";
while(*str)
    printf("%c\n",*str++);
char * pArray[] =
{"apple","pear","banana","orange","pineApple",NULL};

char **pa = pArray;
while(*pa != NULL)
{
    printf("%s\n",*pa++);
}

return 0;
}
```

4.4.4.常见指针数组

4.4.4.1. char**argv 与 char*argv[]

```
#include <stdio.h>

int main(int argc, char **argv /*char *argv[]*/)
{
    // for(int i=0; i<argc; i++)
    // {
    //     printf("%s\n",argv[i]);
    // }
    while(*argv != NULL)
    {
        printf("%s\n",*argv++);
    }
    return 0;
}
```

4.4.4.2. char **env 与 char*env[]

```
#include <stdio.h>

int main(int argc, char **argv, char*env[])
{
    while(*env)
    {
        printf("%s\n", *env++);
    }
    return 0;
}
```

4.5.指针的输入与输出

指针作输出与输入，是一种常见的称谓，输入指的是，指针指向的内容，作为函数处理的原始数据的一部分。

```
char * strcpy(const char * dest, const char* src);
```

src 作输入的。

输出指的是，指针指向空间，可以作为函数返回的情形。比如上题中的 **dest** 就是用作输出的。

还比如，前面我们讲的通过二级指针，初始化一级指针的情形。

```
struct Stu * ps = NULL;
int ret = init(&ps);
```

4.6.堆上一维空间

4.6.1.返回值返回(一级指针)

```
char * allocSpace(int n)
{
    char *p = (char*)malloc(n);
    return p;
}
```

4.6.2.参数返回(二级指针)

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

int allocSpace(char **p,int n)
{
    *p = (char*)malloc(n);
    return *p== NULL?-1:1;
}

int main(void)
{
    char *p;
    if(allocMem(&p,100)<0)
        return -1;
    strcpy(p,"china");
    printf("%s\n",p);
    free(p);
    return 0;
}
```

4.7.堆上二维空间

二维数组，是一种二维空间，但是不代表，二维空间就是二维数组。二维空间，并不一定就是二维数组，但具有数组的访问形式。但已经远远不是数组的定义了。

4.7.1.一级指针作返回值输出

```
void * alloc2dSpace(int base,int row,int line)
{
    void *p = malloc(base*row*line);
    return p;
}

int main()
{
    int (*p)[5] = alloc2dSpace(sizeof(int),3,5);
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<5; j++)
```

```
        {
            p[i][j] = i+j;
        }
    }

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<5; j++)
        {
            printf("%d ",*(p+i)+j));
        }
        putchar(10);
    }
    free(p);
    return 0;
}
```

4.7.2.二级指针作返回值输出

4.7.2.1. 空间申请

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

void ** alloc2dSpace(int base, int row,int line)
{
    void **p = malloc(row*sizeof(void*));
    for(int i=0; i<row;i++)
    {
        p[i] = malloc(base * line);
    }
    return p;
}

int main(void)
{
    int **p = alloc2dSpace(sizeof(int),3,4);
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<4; j++)
```

```
    {
        p[i][j] = i+j;
    }
}

for(int i=0; i<3; i++)
{
    for(int j=0; j<4; j++)
    {
        printf("%d ",p[i][j]);
    }
    putchar('\n');
}

return 0;
}
```

4.7.2.2. 空间释放

4.7.3.三级指针作参数输出

```
void alloc2dSpace(void ***p,int base,int row,int line)
{
    *p = malloc(row*sizeof(void*));
    for(int i=0; i<row;i++)
    {
        (*p)[i] = malloc(base * line);
    }
}
```

4.7.4.课堂练习

4.7.4.1. 序列加密

现有字符序列 `char buf[] = "hello everyone"` ；

明文	密文	明文
h	h	h
e	o	e
l	e	l
l	r	l
o	y	o
e	-	e
r	-	r
y	-	y
o	-	o
n	-	n
e	-	e
-	-	-
-	-	-

按行读的话，肯定可以读出数据，如果按列来读的话，则会出再乱码的现像。正是这种现像可作为一种加密手段，称为序列加密。

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

//chi
//na
//is
//gre
//at

//cniga
//hasrt
//i e

char * encode(char *buf, int line)
{
    int len = strlen(buf);
    int nlen;
    if(len%line!=0)
        nlen = len + (line - len%line);
    else
        nlen = len;
    char * tmp = (char *)malloc(nlen+1);
    char * secret = (char*)malloc(nlen+1);
    char *psecret = secret;
    strcpy(tmp,buf);
    for(int i=strlen(tmp); i<nlen; i++)
```

```
    tmp[i] = ' ';
    tmp[nlen] = '\\0';

    int row = nlen/line;
    char (*ptmp)[line] =tmp;
    for(int i=0; i<line; i++)
    {
        for(int j=0; j<row; j++)
        {
            *psecret++ = ptmp[j][i];
        }
    }
    *psecret = '\\0';
    free(tmp);
    return secret;
}

char * decode(char* buf, int line)
{
    int len = strlen(buf);
    int nline = len/line;
    int row = line;
    char * desecret = (char*)malloc(len+1);
    char *pd = desecret;
    char (*p)[nline] = buf;
    for(int i=0; i<nline; i++)
    {
        for(int j=0; j<row; j++)
        {
            *pd++ = p[j][i];
        }
    }
    *pd= '\\0';
    while(*(--pd) == 32)
        *pd= '\\0';
    return desecret;
}

int main()
{
    char buf[] = "china is great";
```



```
char * secret = encode(buf,3);
printf("%s\n",secret);

char * desecret = decode(secret,3);
printf("%s\n",desecret);

free(secret);
free(desecret);
}
```

4.7.4.2. 读文件到内存

请将文件中的内容读到堆中，并打印。

```
bin:x:1:1/sbin/nologin
daemon:x:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
```

4.7.4.3. 读 HTTP 之 Get 请求

4.8.const 修饰指针

use const whatever possible，增强程序的健壮性。

4.8.1.const 修饰变量

const 修饰变量，此时称为常变量。常变量，有常量的属性，比用宏定义的常量有了类型的属性。

```
#include <stdio.h>
#define N 400a

int main(void)
{
```

```
//    const int a = 400a;

    const int a = 200; //定义的时候必须初始化
    printf("a = %d\n",a);
//    a = 200;
    printf("a = %d\n",a);

    int *p = &a;    //
    *p = 300;
    printf("a = %d\n",a);

    return 0;
}
```

4.8.2.const 修饰符

const 修饰指针，表示指针的指向是恒定的，不可更改。

```
#include <stdio.h>

int main(void)
{
    int a;
    int * const p = &a;
    int b ;
    p = &b;
    return 0;
}
```

4.8.3.const 修饰指针指向

```
#include <stdio.h>

int main(void)
{
    int a;
    const int * p = &a;

    printf("p = %p\n",p);
    int b ;
    p = &b;
    printf("p = %p\n",p);

    *p = 200;
```

```
const int *const q = &a;
q = &b;
*q = 200;

return 0;
}
```

4.8.4.应用(修饰函数参数)

常用于修饰入参指针，表示其指向的内容不可以修改。这样，可以增加程序的健壮性。如果用户不小心发生了修改行为，则会用编译警告来提示，而不是用运行的错误来提示。

```
char * strcpy ( char * destination, const char * source );
char * strcat ( char * destination, const char * source );
```

4.9.练习

4.9.1.请手写下面代码的输出结果

```
#include<stdio.h>
void func(int b[100])
{
    printf("%d\n",sizeof(b));
}
int main()
{
    int *p = NULL;
    printf("%d\n",sizeof(p));
    printf("%d\n",sizeof(*p));

    int a[100];
    printf("%d\n",sizeof(a));
    printf("%d\n",sizeof(a[100]));
    printf("%d\n",sizeof(&a[0]));

    int b[100];
    func(b);
    int c[2][3];
    printf("%d\n",sizeof(c));
```

```
printf("%d\n",sizeof(c[0]));  
int (*d)[3];  
printf("%d\n",sizeof(d));  
char *q = malloc(100);  
printf("%d\n",sizeof(q));  
//printf("%d\n",sizeof(&a)); //window vs linux  
return 0;  
}
```

4.9.2.天生棋局

有两个函数原型的题目可任意实现一个

1.传入一个n 在堆空间中产生n*n 方格的棋盘

```
int ** createBoard(int n);  
int createBoard1(int ***p,int n);
```

2.N 颗棋子随机落在棋盘上<需要防止落在同一位置>

```
int initBoard(int **p,int n)
```

3. 打印棋盘

```
int printBoard(int **p,int n);
```

如果有两颗棋子落同一行或者同一列 输出好棋，否则 输出 不是好棋。将棋局中的棋子打印出来 空位用 O 有棋子的用 X 表示。

4.销毁棋盘

```
int destroyBoard(int **p,int n);  
int destroyBoard1(int ***p,int n);
```

5. 函数(Fucntion)

5.1.函数多参返回

5.1.1.引列

请写一个函数，同时返回两个正整型数据的和与差。但是我们知道函数只有一个返回值的，该如何实现呢？

```
int foo(int * sum ,int *diff, int a,int b);
```

5.1.2.正解

当我们既需要，通过函数返回值来判断函数调用是否成功，又需要把数据传递出来，此时，我们就要用到多参返回，多参返回，都是通过传递调用空间中的空间地址来实现的。

比如前面我们讲到的，通过参数，返回堆上的一维空间，二维空间和初始化指针就是如此。

5.2.函数指针

5.2.1.函数的本质

函数的本质是一段可执行性代码段的。函数名，则是指向这段代码段的首地址，这段代码总是以 **return** 结尾。

```
#include <stdio.h>
void print()
{
    printf("china\n");
}

int main()
{
    print();
    printf("%p\n",&print);
    printf("%p\n",print);
    int a;
    int *p = &a;

    //函数也是一个指针，应当存在一个什么样的指针变量中呢？
    return 0;
}
```

5.2.2.函数指针变量定义与赋值

```
#include <stdio.h>
void print()
{
    printf("china\n");
}
void dis()
{
    printf("china\n");
}

int main()
{
    void (*pf)() = print; //void (*pf)() = &print;
    pf();                //(*pf)();
    pf = dis;
    pf();
    return 0;
}
```

5.2.3.函数指针类型定义

```
#include <stdio.h>
void print()
{
    printf("china\n");
}
void dis()
{
    printf("china\n");
}

typedef void (*PFUNC)();

int main()
{
    PFUNC pf= print;
    pf();
    pf = dis;
    pf();
    return 0;
}
```

5.2.4.函数类型别名

5.2.5.函数指针调用

5.2.6.函数指针数组

可不可以对函数进行数组化管理呢？即，将同参同返回的函数，纳入数组中管理，答案是肯定的。

函数指针数组的本质，也是数组，也是指针数组，只不过每个成员都是一个函数指针而已。

5.2.6.1. 定义

一个返回值和参数皆为 `void`，函数指针数组，定义如下，`funcArray` 先跟 `[]` 结合构成数组，然后跟 `*` 结合构成指针数组，前后的 `void` 修饰，用于表示数组成的成员是一个函数指针。

```
void (*funcArray[N])(void);
```

5.2.6.2. 应用场景

函数指针的一个用法出现在菜单驱动系统中。例如程序可以提示用户输入一个整数值来选择菜单中的一个选项。用户的选择可以做函数指针数组的下标，而数组中的指针可以用来调用函数。

```
#include <stdio.h>
void function0(int);
void function1(int);
void function2(int);

int main()
{
    void (*f[3])(int) = {function0,function1,function2};
    //将这3个函数指针保存在数组 f 中

    int choice;

    printf("Enter a number between 0 and 2, 3 to end: ");
    scanf("%d",&choice);

    while ((choice >= 0) && (choice <3))
    {

        (*f[choice])(choice);
        //f[choice]选择在数组中位置为 choice 的指针。
    }
}
```

```
//指针被解除引用，以调用函数，并且 choice 作为实参传递给这个函数。
printf("Enter a number between 0 and 2,3 to end: ");
scanf("%d",&choice);
}

printf("Program execution completed.");
return 0;
}

void function0(int a)
{
    printf("You entered %d so function0 was called\n",a);
}

void function1(int b)
{
    printf("You entered %d so function1 was called\n",b);
}

void function2(int c)
{
    printf("You entered %d so function2 was called\n",c);
}
```

5.3.回调函数

5.3.1.问题引出

当我们要实现排序的时候，升序和降序，都是写死在程序中的，如果要改只能改动源代码，那么如果程序是以库的形式给出的呢？那又如何呢？

```
#include <stdio.h>

void selectSort(int *p, int n)
{
    for(int i=0; i<n-1 ;i ++){
        for(int j=i+1; j<n; j++){
            if(p[i] < p[j])
            {
                p[i] = p[i]^p[j];
                p[j] = p[i]^p[j];
                p[i] = p[i]^p[j];
            }
        }
    }
}
```



```
        p[j] = p[i]^p[j];
        p[i] = p[i]^p[j];
    }

    }

}

int main(void)
{
    int arr[10] = {6,5,4,3,2,1,7,8,9,0};
    selectSort(arr,10);
    for(int i=0; i<10; i++)
    {
        printf("%d\n",arr[i]);
    }
    return 0;
}
```

5.3.2.回调(函数作参数)

```
#include <stdio.h>

int callBackCompare(int a,int b)
{
    return a<b?1:0;
}

void selectSort(int *p, int n,int(*pf)(int,int))
{
    for(int i=0; i<n-1 ;i ++ )
    {
        for(int j=i+1; j<n; j++)
        {
            if(pf(p[i],p[j]))
            {
                p[i] = p[i]^p[j];
                p[j] = p[i]^p[j];
                p[i] = p[i]^p[j];
            }
        }
    }
}
```

```

    }
}

int main(void)
{
    int arr[10] = {6,5,4,3,2,1,7,8,9,0};
    selectSort(arr,10,callBackCompare);
    for(int i=0; i<10; i++)
    {
        printf("%d\n",arr[i]);
    }
    return 0;
}

```

5.3.3.本质论

回调函数，本质也是一种函数调用，先将函数以指针的方式传入，然后，调用。这种写法的好处是，对外提供函数类型，而不是函数定义。这样我们只需要依据函数类型和函数功能提供函数就可以了。给程序的书写带来了很大的自由。

5.3.4.qsort

标准库为我们封装的基于数组快速排序的**万能**接口。

5.3.4.1. 函数接口

函数声明	<pre>void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));</pre>	
所在文件	stdlib.h	
函数功能	快速排序由 base 指向的，num 个成员大小为 size 的的数组，compar 决定是升序，还是降序。	
参数及返回解析		
参数	void* size_t size_t int (*)(void*,void*)	指向数组的指针 数组中成员的个数 每个成员的大小 决定升序降序的回调函数
返回值	void	无返回值

5.3.4.2. 回调函数例举

以 int 类型数组为例，实现升序排列，qsort 只对 compare 返回的正值感兴趣，也就是

当返回正值才发生交换行为。

```
int cmp ( const void *a , const void *b )
{
    type va = *(type*)a; type vb = *(type*)b;
    if(va > vb)
        return 1;
    else if (va < vb)
        return -1;
    else
        return 0;
}
```

■ 整型类示范

```
int cmp ( const void *a , const void *b )
{
    return *(int *)a - *(int *)b; //return *(int*)a >*(int*)b?1:-1;
}
```

■ 字符型示范

```
int cmp( const void *a , const void *b )
{
    return *(char *)a - *(char*)b;
}
```

■ 浮点型示范

```
int cmp( const void *a , const void *b )
{
    return *(double *)a > *(double *)b ? 1 : -1;
}
```

■ 对结构体一级排序

```
struct In
{
    double data;
    int other;
} s[100];
//按照 data 的值从小到大将结构体排序
int cmp( const void *a ,const void *b)
{

```

```
return (*(struct In *)a).data > (*(struct In *)b).data ? 1 : -1;
}
```

■ 对结构体二级排序

```
struct In { int x; int y; }s[100];
//按照 x 从小到大排序，当 x 相等时按照 y 从大到小排序
int cmp( const void *a , const void *b )
{
    struct In *c = (struct In *)a; struct In *d = (struct In *)b;
    if(c->x != d->x)
        return c->x - d->x;
    else
        return d->y - c->y;
}
```

■ 对字符串进行排序

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int cmp(const void *a, const void *b)
{
    return strcmp((char*)a,(char*)b) > 0? 1:-1;
    return strcmp(*(char**)a,*(char**)b) > 0? 1:-1;
}

int main(void)
{
    char * pa[4] ={"Huawei","MicroSoft","Apple","Oracle"};
    qsort(pa,4,4,cmp);

    for(int i=0; i<4; i++)
    {
        printf("%s\n",pa[i]);
    }
    return 0;
}
```

5.3.4.3. 课堂练习

请将下面的数组排序：要求，先按名字排序，如果名字相同，则按成绩排序。Stu stu[] = {"aaa",23.5},{"xxx",45.6},{"bbb",89},{"xxx",23.4},{"yyy",100};

```
typedef struct
{
    char name[30];
    float score;
}Stu;

Stu stu[] =
{"aaa",23.5},{"xxx",45.6},{"bbb",89},{"xxx",23.4},{"yyy",100};

int myCmp(const void * a,const void * b)
{
    if(strcmp(((Stu*)a)->name,((Stu*)b)->name) != 0)
        return strcmp(((Stu*)a)->name,((Stu*)b)->name);
    else
        return ((Stu*)a)->score > ((Stu*)b)->score?1:-1;
}

int main()
{
    qsort(stu,sizeof(stu)/sizeof(stu[0]),sizeof(stu[0]),myCmp);
    for(int i=0; i< 5 ;i++)
    {
        printf("%s %f\n",stu[i].name,stu[i].score);
    }
    return 0;
}
```

5.4.递归函数

5.4.1.迭代变递归引入

5.4.1.1. 迭代

```
int main()
{
    for (int i=0;i<5;i++) {
```

```
        printf("%d\t",i);  
    }  
}
```

5.4.1.2. 递归

```
void print(int i){  
    if (i == 5)  
        return;  
    printf("%d\t",i);  
    print(i+1);  
}  
  
int main(){  
    print(0);  
}
```

5.4.2.递归公式

5.4.2.1. 累和

```
//f(n) = n + f(n-1)  n>1  
//f(1) = 1
```

5.4.2.2. 阶乘

```
//f(n) = n *f(n-1)  n>0  
//f(0) = 1          n=0
```

5.4.3.书写公式

```
void print(int i)  
{  
    if 终止条件  
        return;  
    print(递向终止);  
    printf("%d\t",i);  
}  
  
void print(int i)  
{  
    if (终止条件)  
    {  
        printf("%d\t",i);  
    }
```

```
    print(递归终止);  
}  
}
```

5.4.4.递归处理

```
void print(int i)  
{  
    if (i == 5)  
        return;  
    printf("%d\t",i);  
    print(i+1);  
}
```

5.4.5.归和处理

```
void print(int i)  
{  
    if (i == 5)  
        return;  
    print(i+1);  
    printf("%d\t",i);  
}
```

5.4.6.返回值递归

5.4.6.1. 求和

```
//f(n) = n + f(n-1)  n>=2  
//f(1) = 1  
  
int sum(int n)  
{  
    if(n >=2)  
        return n + sum(n-1);  
    return 1;  
}  
  
int main()  
{  
    int s = sum(100);  
    printf("s = %d\n",s);  
}
```

5.4.6.2. 求 n!

```
//f(n) = n *f(n-1)  n>0
//f(0) = 1          n=0

int fact(int n)
{
    if(n > 1)
        return n * fact(n-1);
    return 1;
}

int main()
{
    int f = fact(5);
    printf("f = %d\n",f);
}
```

5.5.练习

5.5.1. `*(void(*)())0();`

请阐述此标题的语义？

6. 再论指针与数组

由于数组名，是常量，指针对数组的引用可以说是对数组名访问方式的解放。使其更自由，方便。指针实现了数组名不能实现的增量运算。

6.1. 一级指针与一维数组名

6.1.1. 等价条件

```
int arr[N] <=> int *p = arr + N
```

6.1.2. 便利访问

```
*p++
```

6.2. 二级指针与指针数组名

6.2.1. 等价条件

```
char *arr[N] <=> char **p = arr + N  
char *arr[N] = {, NULL} <=> char **p
```

6.2.2. 便利访问

```
while(*p)  
    *p++;
```

6.2.3. 通过二级指针申请二维空间

6.3. 数组指针与二维数组名

6.3.1. 等价条件

```
int arr[M][N] <=> int (*p)[N] + M
```

6.3.2. 便利访问

可以任意的一维空间，当作二维的方式来访问。比如序列加密。

6.4. 关于二级指针与二维数组名

两者没有关系，一个指针**数组**，而另一个是数组**指针**。

```
int **p    != int a[][]
```

6.5.对数组名的引用

6.5.1.一维数组

```
//int a[3]; int[3] a;  &a; int[3] *p; int (*p)[3];
int (*p)[3] = &a;
```

6.5.2.二维数组

```
//int a[3][4]; A      a[3]; A[3]  a; &a; A[3] *p; A (*p)[3];
//int[4] (*p)[3]; int(*p)[3][4];
int(*p)[3][4] == &a;
```

6.6.小结与练习

6.6.1.指针-数组-函数

声明修饰符	含义	优先级	示例
()	表示函数	高	char (* func[3])()
[]	表示数组		char (*p)[N]
*	表示指针	低	char * p[N]

6.6.2.请写出右边的示意

类型	含义
int array[8][8]	
int *ptr	
int * array[10]	
int * p[3][4]	
int (*p)[3][4]	
int (*p[3])[4]	
int(**P)[3]	
char * func()	
char (*func)()	
char (* func[3])()	

6.6.3.传参汇总

原数据类型	应用	传递实参类型	匹配形参类型
int arr[4];	等价传递	foo(arr, 4)	foo(int *p, int n)
int arr[3][4];	等价传递	foo(arr, 3)	foo(int (*p)[4], int n)
char *argv[3];	等价传递	foo(argv, 3)	foo(char **p, int n)
char * p;	等价传递	foo(p)	foo(char *p)
char * p=NULL;	初始化传递	foo(&p)	foo(char **p)
char **p	等价传递	foo(p)	foo(char **p)
char **p =NULL;	初始化传递	foo(&p)	foo(char ***p)
int (*p)[4];	等价传递	foo(p)	foo(int (*p)[4])
void func(int,int);	回调传递	foo(func)	foo(void (*p)(int, int))

详见：http://blog.csdn.net/linux_wgl/article/details/51713540

6.6.4.写出程序的运行结果

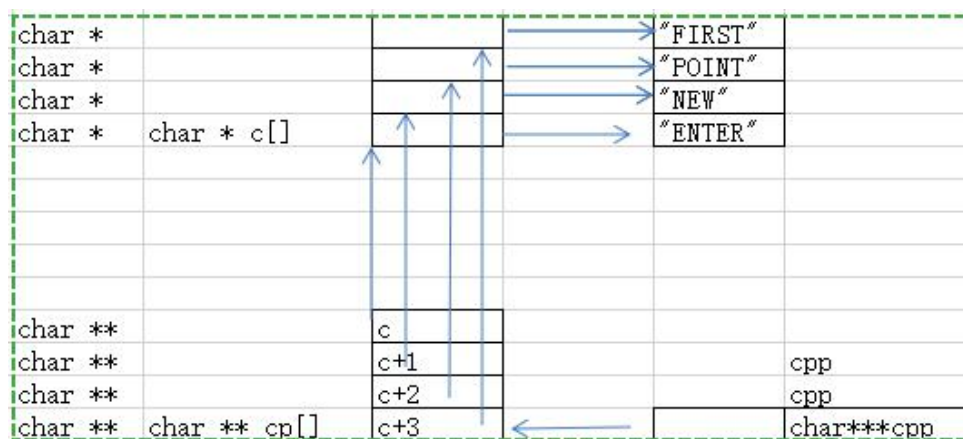
6.6.4.1. 题目

```

7.写出程序的运行结果
char *c[] = {"ENTER","NEW","POINT","FIRST"};
char **cp[] = { c + 3 , c + 2 , c + 1 , c};
char ***cpp = cp;
int main(void)
{
    printf("%s",**++cpp);
    printf("%s",*--*++cpp+3);
    printf("%s",*cpp[-2]+3);
    printf("%s\n",cpp[-1][-1]+1);
    return 0;
}

```

6.6.4.2. 内存图示



7. 基于数组排序(Sort)提高篇

7.1.选择优化

7.1.1.选择原理与步骤

- ① 在未排序的序列中取首元素，同其后的所有元素一一比较，若大，则交换。
- ② 经上轮排序，首元素有序。
- ③ 重复上面操作，直至未排序元素个数为 1。

7.1.2.代码实现

```
void selectSort(int *data, int n)
{
    for (int i = 0; i < n-1; i++)
    {
        for (int j = i + 1; j < n ;j++)
        {
            if (data[i]>data[j])
            {
                swap(&data[i], &data[j]);
            }
        }
    }
}
```

7.1.3.优化升级

7.1.3.1. 原则

比而不换，记录当轮最小下标。 在没有减少比较次数的情况下，减少交换次数。

7.1.3.2. 逻辑

记录开始下标 i 下标为最小下标 minIdx ，用此下标来比较数据，若有更小下标 j ，则更新此下标 minIdx 。本轮结束若 $\text{minIdx}==i$ ，说明最初下标处的值为最小，若不等，则将 minIdx 和 i 处的值交换。

7.1.3.3. 实现

```
void selectSort(int *p, int n)
{
    int minIdx;
```

```
for(int i=0; i<n-1; i++)
{
    minIdx= i;
    for(int j=i+1; j<n; j++)
    {
        if(p[minIdx]>p[j])
        {
            minIdx= j;
        }
    }
    if(minIdx!= i)
    {
        p[i]    = p[i]^p[minIdx];
        p[idx]   = p[i]^p[minIdx];
        p[i]    = p[i]^p[minIdx];
    }
}
```

7.2.冒泡优化

7.2.1.冒泡原理与步骤

- ① 从头开始，比较相邻的两个元素，如果第一个比第二个大则交换，直至结束。
- ② 上一轮比较结束，最大的数沉底。不再参与下轮比较。
- ③ 重复以上操作，直到待排序元素个数为 1。

7.2.2.代码实现

```
void popSort(int *data,int n)
{
    for (int i = 0; i < n-1; i++)
    {
        for (int j = 0; j < n -i- 1; j++)
        {
            if (data[j]>data[j + 1])
            {
                swap(&data[j], &data[j + 1]);
            }
        }
    }
}
```

7.2.3.优化升级

7.2.3.1. 原则

序而不排，如果在一轮比较中没有发生交换行为，说明已经有序，无需再排。

7.2.3.2. 逻辑

先假设,标志位为有序 **ordered = 1**，内重循环中若没有将其清零 **ordered = 0**，则认为本轮已有序，无需下轮。

7.2.3.3. 实现

```
void popSort(int *data,int n)
{
    int ordered;
    for (int i = 0; i < n-1; i++)
    {
        ordered = 1; // 每一轮比较前都要设置标志位,假设有序
        for (int j = 0; j < n - i - 1; j++)
        {
            if (data[j] > data[j + 1])
            {
                data[j] = data[j]^data[j+1];
                data[j+1] = data[j]^data[j+1];
                data[j] = data[j]^data[j+1];
                flag = 0; //若有发生比较，则无序，还需要继续比较
            }
        }
        if(flag)
            break;
    }
}
```

7.3.快速排序

7.3.1.排序原理与步骤

- ① 从数组中挑出一个元素作为基准(原则上随机，为了简便我们取第一个元素)，
- ② 重新排列数组，所有比基准小的位于基准的左侧，所有比基准大于或等于的位于基准的右侧。此过程称为分组。
- ③ 以**递归**的方式分别对基准两侧的子序列进行排序。

7.3.2.代码实现

```
void quickSort(int *p, int low ,int high)
{
    if(low < high)
    {
        int pivot = p[low];
        int l = low; int h = high;
        while(l < h)
        {
            while(p[h] >= pivot && l < h)
                h--;
            p[l] = p[h];
            while(p[l] <= pivot && l < h)
                l++;
            p[h] = p[l];
        }
        p[l] = pivot;
        quickSort(p, low, l-1);
        quickSort(p, l+1, high);
    }
}
```

7.3.3.逻辑质问

7.3.3.1. 不处理内对外循环影响

```
void quickSort(int *p, int left, int right)
{
    if(left < right)
    {
        int low = left, high = right;
        int pivot = p[low];
        while(low < high)
        {
            while(p[high] > pivot )
                high--;
            p[low] = p[high];

            while(p[low] < pivot)
                low++;
            p[high] = p[low];
        }
    }
}
```



```

    p[high] = p[low] = pivot;    //中心节点

    quickSort(p,left,low-1);
    quickSort(p,high+1,right);
}
}

```

反例 arr[10] = {1,2,3,4,5,6,7,8,9,10}

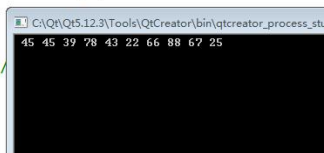
7.3.3.2. 不处理相等逻辑

```

if(left < right)
{
    int low = left, high = right;
    int pivot = p[low];
    while(low < high)
    {
        while(p[high] > pivot && low < high)
            high--;
        p[low] = p[high];

        while(p[low] < pivot && low < high)
            low++;
        p[high] = p[low];
    }
    p[high] = p[low] = pivot;    //
    quickSort(p,left,low-1);
    quickSort(p,high+1,right);
}

```



此情形会导致死循环。

7.3.3.3. 终结版

7.4.练习

7.4.1.实现二级排序

现在有一个学生的结构体数组，实现按成绩排序，如果成绩相同，则按学号排序。结构体节点如下：

```

struct Stu
{
    char name[30];
    int num;
    float score;
};

```

8.基于数组的查找(Search)

8.1.线性查找

```
int linearSearch(int *data, int size, int find)
{
    for (int i = 0; i < size; i++)
    {
        if (find == data[i])
        {
            return i;
        }
    }
    return -1;
}
```

8.2.二分查找

8.2.1.原理步骤

二分查找的前提是原始数据有序。

- ① 假设集合中数据是升序排列，将集合中间的元素同目标比较。若相等，则查找成功。否则利用中间位置将集合分成两个子集。
- ② 若中间元素大于目标元素，则在左子集中查找，否则则在右子集中查找。
- ③ 重复以上操作，直至找到要查找的元素，或是直到子集不存在。

8.2.2.迭代版(Iterative)

```
int binarySearch(int *data, int low, int high, int find)
{
    while (low <= high) // low 和 high 相等的时候，还要比较一次
    {
        int mid = (low + high) / 2;
        if (data[mid] == find)
            return mid;
        else if (data[mid] > find)
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

```
}  
    return -1;  
}
```

8.2.3.递归版(Recursive)

8.2.3.1. ???

凡是有返回的函数，即便是没有变量接收，系统也会产生一个匿名空间，作为返回值的空间。

若是将 `else` 拿掉，则不会产生正确结果，即为例证。

```
int binarySearch(int * data, int low ,int high, int find)  
{  
    // 循环的中止条件，就是递归的中止条件。  
    if (low <= high)  
    {  
        int mid = (low + high) / 2;  
        if (find == data[mid])  
            return mid;  
        else if (find > data[mid])  
            binarySearch(data, mid + 1, high, find); //或是 return  
        else  
            binarySearch(data, low, mid-1, find);    //或是 return  
    }  
    else  
        return -1;  
}
```

8.2.3.2. 正确版

```
int binarySearch(int * data, int low ,int high, int find)  
{  
    // 循环的中止条件，就是递归的中止条件。  
    if (low <= high)  
    {  
        int mid = (low + high) / 2;  
        if (find == data[mid])  
            return mid;  
        else if (find > data[mid])  
            return binarySearch(data, mid + 1, high, find);  
        else  
            return binarySearch(data, low, mid-1, find);  
    }  
}
```

```
return -1;  
}
```

9. 字符串(String)

9.1.字符串常量

9.1.1.定义相关

C 语言并没有显示的提供字符串类型，所以字符串常以常量的形式存在。字符串是一串用双引号引起的一串字符(零个或多个)，形如"china"，系统默认在其后添加了 NUL 字符，用于表示结束，NUL 字节不存在与其它可打印字符的关联，这也是 NUL 被选为中止符的原因，但是字符串的长度不包含其在内。

9.1.2.c 语言处理字符串

c 语言中它是以常量的形式存在，编译的时候处理成一个指向常量字符串的指针。指针的类型是 `const char * const` 。

```
#include <stdio.h>
int main()
{
    printf("%s\n %p\n %c\n %c\n",
           "china","china","china","china"[3]);
    return 0;
}
```

```
int main()
{
    printf("sizeof(\"china\") = %d\n",sizeof("china"));
    printf("sizeof(\"\") = %d\n",sizeof("")); //空串的表示方法
}
```

空指针与指针指向空串

```
char *p = NULL;
char *pp = "";
```

9.2.字符数组

字符串常量很适用于那些不会对其进行修改的情形。要对其修改，则要存储于字符数组去。

字符数组是一种可以存在于栈区或是堆区的一种数据类型，它可以用于存储字符串，实现对字符串的修改等一系列的操作。

9.2.1. 字符数组与字符串

```
int main()
{
    char buf [] = "abcdefg";
    printf("%p %p\n",buf, buf+1);
    printf("%p %p\n","abcdefg", "abcdefg"+1);
    printf("%d %d\n",sizeof(buf),sizeof("abcdefg")); //大小，步长匹配
    return 0;
}
```

9.2.2. 不等价条件

读越界，常常带来写越界，读越界，往往并不会造成内存崩溃。而写越界，则会造成数据破坏，严重会造成内存崩溃。

9.2.2.1. 读越界

```
int main()
{
    char src[5] = "china is great";
    printf("%s\n",src);
    return 0;
}
```

9.2.2.2. 写越界

```
#include <stdio.h>
#include <string.h>
void foo(void)
{
    char string[10],str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy(string,str1); //读越界往往会伴随着写越界的发生
    printf("%s",string);
}
int main()
{
    foo();
    return 0;
}
```

```
}
```

9.2.3.等价条件

字符串，存于字符数组，不存在读越界与写越界，那么就可以称为等价了。

9.2.4.N 系列字符串函数

N 系列函数存在的意义，就是在避免写越界。

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

int main(void)
{
    char * p = "123456";
    char str[10];
    strncpy(str,p,10);          //将读到的'\0',写入 str
    printf("str = %s\n",str);

    char * pp = "12345678911234";
    strncpy(str,pp,10);
    printf("str = %s\n",str); //通常会有乱码打出

    char * ppp = "123456789aaaaa";
    strncpy(str,ppp,9);          //只读到了 9 个字符，然后在尾后加'\0'
    str[9] = '\0';
    printf("str = %s\n",str);

    return 0;
}
```

9.3.基础字符串操作函数及自实现

9.3.1.strlen

9.3.1.1. 使用

9.3.1.2. 自实现

```
int myStrlen(const char * str)
{
    int len = 0;
```

```
while(*str++)
    len++;
return len;
}
```

9.3.2.strcpy/strncpy

9.3.2.1. 使用

9.3.2.2. 自实现

```
char *myStrcpy(char *dst, const char * src)
{
    char *d = dst;
    while(*dst++ = *src++);
    return d;
}
```

9.3.3.strcat/strncat

9.3.3.1. 使用

9.3.3.2. 自实现

```
char *myStrcat(char *dst, const char *src)
{
    char * d = dst;
    while(*dst) dst++;
    while(*dst++ = *src++);
    return d;
}
```

9.3.4.strcmp/strncmp

9.3.4.1. 使用

9.3.4.2. 自实现

```
int myStrcmp(const char * s1, const char * s2)
{
    for(;;*s1 && *s2; s1++,s2++) // 再简洁,该如何写呢?
    {
        if(*s1 == *s2)
            continue;
        else
            return *s1 - *s2;
    }
    return *s1 - *s2;
}
```


}

9.3.5.sprintf

函数声明	int sprintf (char * str, const char * format, ...);	
所在文件	string.h	
函数功能	将数据按 format 的格式，写入某个字符串缓冲区。	
参数及返回解析		
参数	char *	
	char *	
	...	
返回值	int	

```
int sprintf ( char * str, const char * format, ... );
int main()
{
    int a,b,c,d;
    printf("pls input ip:");
    scanf("%d.%d.%d.%d",&a,&b,&c,&d);
    char buf[16];
    sprintf(buf,"%d.%d.%d.%d",a,b,c,d);
    printf("%s\n",buf);
    return 0;
}
```

9.3.6.atoi/itoa

函数声明	int atoi(const char *nptr);	
所在文件	string.h	
函数功能	将字符串转换成整型数; atoi()会扫描参数 nptr 字符串, 跳过前面的空格字符, 直到遇上数字或正负号才开始做转换, 而再遇到非数字或字符 ('0') 时才结束转化, 并将结果返回 (返回转换后的整型数)。	
参数及返回解析		
参数	char *	待转化的字符串
返回值	int	转化后的整型数据

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

int main()
{
```

```

char buf[100] = " 123abc";
int data = atoi(buf);
printf("data = %d\n",data);

char a[] = "-100";
char b[] = "99";
printf("a + b = %d\n",atoi(a)+atoi(b));
return 0;
}

```

函数声明	char * itoa (int value, char * str, int base); (非标库函数)	
所在文件	string.h	
函数功能	Converts an integer value to a null-terminated string using the specified base and stores the result in the array given by str parameter. 译： 根据指定的进制(base), 将整型数据转化为以'\0'结尾的字符串，保存到 str 指向的字符数组中。	
参数及返回解析		
参数	int char * int	待转化的整数 存放字符串的空间 进制
返回值	char *	转化后的字符串首地址

```

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

int main()
{
    char buf[100];
    int a = 123;
    itoa(a,buf,10);
    printf("base 10 %s\n",buf);
    itoa(a,buf,16);
    printf("base 16 %s\n",buf);
    itoa(a,buf,2);
    printf("base 2 %s\n",buf);
    return 0;
}

```

9.3.7.strchr/strchr

9.3.7.1. 函数介绍

函数声明	char *strchr(char* str,int ch)	
所在文件	string.h	
函数功能	返回字符串 str 中 首次出现 字符 c 的位置指针，找不到返回 NULL。	
参数及返回解析		
参数	char * int	待查找字符串 要查找的字符
返回值	char*	待查打字符的地址或找不到返回 NULL

9.3.7.2. 使用

```
#include <stdio.h>
#include <string.h>
int main()
{
    char buf[100] = "china";
    char *p = strchr(buf,'n');
    printf("%s\n",p);
    return 0;
}
```

9.3.7.3. 统计一个字符在字符串内出现的次数

```
int calcCharCountOfString(char * str, char ch)
{
    int count = 0;
    while(*str != '\0')
    {
        if(ch == *str)
            count++;
        str++;
    }
    return count;
}
```

```
int calcCharCountOfString(char * str, char ch)
{
    int count = 0;
    while(str = strchr(str,ch))
```

```

{
    count++;
    str++;
}
return count;
}

```

9.3.7.4. 自实现

```

char * myStrchr(char *str, int ch)
{
    while(*str != ch&& *str != '\0')
        str++;
    if(*str == '\0')
        return NULL;
    else
        return str;
}

```

9.3.8.strcspn/strspn

9.3.9.strstr/strpbrk

9.3.9.1. 函数接口

函数声明	char * strstr (char * str1, const char* str2);	
所在文件	string.h	
函数功能	strstr() 函数搜索字符串 str2 在字符串 str1 中是否出现。找到所搜索的字符串，则该函数返回 第一次匹配 的字符串的地址；如果未找到所搜索的字符串，则返回 NULL。	
参数及返回解析		
参数	char *str1 char *str2	待查找字符串 要查找的字符中
返回值	char *	待查找字符串的地址或 NULL

9.3.9.2. 使用

```

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

```

```
int main()
{
    char str1[] = "abcd123456efg";
    char str2[] = "1234";
    char *pf = strstr(str1,str2);
    if(pf != NULL)
        printf("%s\n",pf);
    else
        printf("find none\n");
    return 0;
}
```

9.3.9.3. 统计一个字符串在另一个字符串内出现的次数

```
int calcStrCountOfString(char * string, char* str)
{
    int count = 0;
    int len = strlen(str);
    while(string = myStrstr(string,str))
    {
        count++;
        string += len;
    }
    return count;
}
```

9.3.9.4. 自实现(借助库函数)

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

char * myStrstr(char *s1,char *s2)
{
    int n = strlen(s2);
    for(;;(s1 = strchr(s1,*s2))!=NULL;s1++)
    {
        if(strncmp(s1,s2,n)==0)
            return s1;
    }
}
```

```
return NULL;
}
```

9.3.9.5. 自实现(不借助任何库函数)

```
char * myStrstr(char *s1,char *s2)
{
    if(*s2) //查找的数据为空
    {
        while(*s1)
        {
            for(int n=0;*(s1+n) == *(s2+n); n++)
            {
                if(*(s2+n+1) == '\0')
                    return s1;
            }
            s1++;
        }
        return NULL;
    }
    else
        return NULL; //return s1;??
}
```

9.3.10.strtok

9.3.10.1.函数接口

函数声明	char *strtok(char *s, <i>char *delim</i>);	
所在文件	string.h	
函数功能	分解字符串为一组字符串。s 为要分解的字符串，delim 为分隔符字符串。首次调用时，s 指向要分解的字符串，之后再次调用要把 s 设成 NULL。 strtok 在 s 中查找包含在 delim 中的字符并用 NULL('\0') 来替换，直到找遍整个字符串。	
参数及返回解析		
参数	char *	待分割的字符串
	char *	分割所依据的字符串
返回值	char *	从 s 开头开始的一个个被分割的串。当没有被分割的串时

	则返回 NULL。所有 delim 中包含的字符都会被滤掉，并将被滤掉的地方设为一处分割的节点。
--	--

9.3.10.2.分析入门

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char buf[] = "aaaaaaaaa@bbbbbbbbbbbbb@ccccccccccc@dddddddddd";
    char delim[] = "@"; // "@#"
    int size = sizeof(buf);
    for(int i=0; i<size; i++)
        printf("%c",buf[i]);
    putchar(10);

    strtok(buf,delim);
    for(int i=0; i<size; i++)
        printf("%c",buf[i]);
    putchar(10);
    strtok(NULL,delim); // strtok(buf,delim);
    for(int i=0; i<size; i++)
        printf("%c",buf[i]);
    putchar(10);
    strtok(NULL,delim); // strtok(buf,delim);
    for(int i=0; i<size; i++)
        printf("%c",buf[i]);
    putchar(10);
    // 第一个或是最后一个是分隔符，如何
    return 0;
}
```

9.3.10.3.使用

解析 linux 密码文件

```
# cat /etc/passwd
root:x:0:0:Superuser:/:
daemon:x:1:1:Systemdaemons:/etc:
bin:x:2:2:Ownerofsystemcommands:/bin:
sys:x:3:3:Ownerofsystemfiles:/usr/sys:
```

程序实现

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

int main()
{
    char buf[] = "sys:x:3:3:Ownerofsystemfiles:/usr/sys: ";

    char *p = strtok(buf, ":");
    while(p != NULL)
    {
        printf("%s\n", p);
        p = strtok(NULL, ":");
    }
    return 0;
}
```

9.3.10.4. 分割字符串到二维空间

```
char ** readStrbyTok(char *str, char *delim){}
```

9.3.10.5. 思考实现 `strtok`

```
static char *olds;
char *STRTOK (char *s, const char *delim)
{
    char *token;

    if (s == NULL)
        s = olds;

    /* Scan leading delimiters. */
    s += strspn (s, delim);    //s 中不包含 delim 中任一字符的首位置
    if (*s == '\0')
```



```
{
    olds = s;
    return NULL;
}

/* Find the end of the token. */
token = s;
s = strpbrk (token, delim); // token 中找 delim 中的字符，首次匹配的指针
if (s == NULL)
    /* This token finishes the string. */
    olds = __rawmemchr (token, '\0');
else
{
    /* Terminate the token and make OLDS point past it. */
    *s = '\0';
    olds = s + 1;
}
return token;
}
```

9.4.字符串 trim 系列

9.4.1.引例

读文件登录的问题。

```
web = edu.nzhsoft.cn
```

9.4.2.去除右空格

```
void trimStrRightSpace(char * str)
{
    while(*str) //区分指向结束标志，还是结束标志的下一个。
        str++;
    while*(--str) == ' ')
        *str = '\0';
}
```

9.4.3.去除左空格

```
void trimStrLeftSpace(char * str)
{

```

```
char *t= str;
if(*t != ' ' )
    return;
while(*t == ' ')
    t++;
while(*str++ = *t++)
    ;
}
```

9.4.4.去除全空格

```
void trimStrSpace(char *str)
{
    char *t = str;
    while(*str)
    {
        if(*str != ' ')
        {
            *t++ = *str;
        }
        str++;
    }
    *t='\0';
}
```

9.4.5.去除任意字符

自实现。

9.5.递归逆置字符串

9.5.1.非递归实现

```
void strReverse(char * str)
{
    char * st = str;
    char * en = str +strlen(str)-1;
    while(st < en)
    {
        *st ^= *en;
        *en ^= *st;
        *st ^= *en;
    }
}
```

```
        st++; en--;  
    }  
}
```

9.5.2.递归逆置

9.5.2.1. 逆置

要求，在不提供任何多余变量的情况下。

```
#include <stdio.h>  
void strReverse(char * str)  
{  
    if(*str)  
    {  
        strReverse(str+1);    //++str 或是 str++均会改变当前层的值  
        printf("%c",*str);  
    }  
    //printf("%c",*str);    printf 在上，在下，会打印不同的结果。  
}  
int main(void)  
{  
    char arr[] = "china";  
    strReverse(arr);  
    return 0;  
}
```

9.5.2.2. 将逆置的串保存(全局)

```
#include <stdio.h>  
#include <string.h>  
  
char buf[1024] = {0};  
void strReverse(char * str)  
{  
    if(*str)  
    {  
        strReverse(str+1);  
        //    printf("%c",*str);  
        strncat(buf,str,1);  
    }  
}
```

```
int main(void)
{
    char arr[] = "china";
    strReverse(arr);

    printf("%s\n",buf);
    return 0;
}
```

9.5.2.3. 将逆置的串保存(局部)

```
#include <stdio.h>
#include <string.h>
void strReverse(char * str,char *buf)
{
    if(*str)
    {
        strReverse(str+1,buf);
        //    printf("%c",*str);
        strncat(buf,str,1);
    }
}

int main(void)
{
    char buf[1024] = {0};
    char arr[] = "china";
    strReverse(arr,buf);

    printf("%s\n",buf);
    return 0;
}
```

9.6.内存操作函数

字符串所有的操作，都是依托了字符串重要的结束标志。但是如果我们要操作某一段内存，比如拷贝一个数组，就要用到内存操作函数了。它的一个重要特点就是，并不关心内容中的标志。

```
void * memcpy (void * dst, void const * src, size_t length);
void * memmove(void * dst, void const * src, size_t length);
int  memcmp (void const * a, void const * b,size_t length);
void * memchr (void const * a, int ch, size_t length);
void * memset (void * a, int ch ,size_t length);
```

每一个函数原型都包含一个显示参数说明需要处理的字节数，但仍然同 **strn** 系列的函数不同，他们在遇到 **NUL** 字符时不会停止。

9.6.1.memset

此函数常用于初始化一段内存空间。比如，初始化一个数组，注意它初始化的**最小单位是字节**。

```
#include <stdio.h>
#include <string.h>
int main()
{
    char buf[1024];
    memset(buf,0,1024);
    //  memset(buf,'a',1024); //这样作很危险
    //  printf("buf = %s",buf);
    strcpy(buf,"china is great\n");
    printf("buf = %s",buf);

    int array[10];

    memset(array,1,10*sizeof(int)); //hex 01010101 -> dec 16843009
    for(int i=0; i<10; i++)
    {
        printf("%d\n",array[i]);
    }
    return 0;
}
```

9.6.2.memcpy

实现**两段空间**的拷贝，比如实现数组间的拷贝，此时，不可以采用 `strcpy` 或是 `strncpy`。因为 `strcpy` 或是 `strncpy` 对 '\0' 敏感。

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

int main(void)
{
    int a[10] = {1,2,3,4,5,0,6,7,8,9}; //整型数组间的拷贝
    int b[10];
    memcpy(b,a,10*sizeof(a[0]));
    for(int i=0; i<10; i++)
    {
        printf("%d\n",b[i]);
    }

    char c[10] = {'a','b','c','d','\0','\n','e','f','g','h'};
    char d[10]; //字符数组间的拷贝
    memcpy(d,c,10*sizeof(a[0]));
    for(int i=0; i<10; i++)
    {
        printf("%d\n",d[i]);
    }
    puts(d);
    return 0;
}
```

9.6.3.memmove

如果目标区域和源区域有重叠的话，`memmove` 能够保证**源串**在被覆盖之前，将重叠区域的字节拷贝到目标区域中，但复制后源内容会被更改。

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

int main ()
{
    //15    20
    char str[] = "memmove can be very useful.....";
    //strncpy(str+20,str+15,11);
    memmove (str+20,str+15,11);
    puts (str);
    return 0;
}
```

```
}  
运行结果：  
memmove can be very very very v.  
memmove can be very very useful.
```

9.6.4.memcmp

比较两段空间的前 n 个字节，其它逻辑等同于 strcmp。

9.6.5.memchr

查找一段空间中的一个字符，若存在则返回，所查找到字符的指针，若无，返回 NULL。

9.7.练习

9.7.1.返回 s2 在 s1 中最后一次出现的位置

函数原型 `char * strLastStr(const char * s1,const char *s2);`

9.7.2.清空字符串中的制表符

重写去除字符串中空格的三个函数，去除的内容，除了空格，还有 '\t'。

9.7.3. 格式化文件

编写一程序，读一个文件，删除其中的空行和非空行中的空格和 '\t'，存入另外一个文件中。

9.7.4.判断字符串 s1 是否是 s2 循环移位所得

比如 `char s1[] = "abcd"` `char s2[] = "bcda"`；

9.7.5.求字符频度

求一个字符串中出现频率最高的那个字符及其出现次数,要求时间复杂度为 $O(N)$

提示：空间换时间：使用一个额外的数组统计每个字符出现的次数，再扫描一次得到查找的字符，这是 $O(N)$ 的时间复杂度。

9.7.6.压缩字符串

删除字符串中的数字并压缩字符串。如字符串 "abc123de4fg56" 处理后变为 "abcdefg"。注意空间和效率。（算法只需要一次遍历，不需要开辟新空间，时间复杂度为 $O(N)$ ）

9.7.7.实现 atoi 功能

编码实现字符串转整型的函数(实现函数 atoi 的功能)。如将字符串 "123" 转化为 123, "-0123" 转化为 -123

9.7.8.实现 itoa 功能

编码实现整型转字符串 itoa 的函数。

原创作者： 王桂林

技术交流：QQ/Wx: 329973169

10.数据结构(双向循环链表)



10.1.双向循环链表存在的意义

数组这样的结构提供了连续内存的访问和使用，链表是对内存零碎空间的有效组织和使用，双向循环链表增大了访问的自由度。

10.2.链表构成

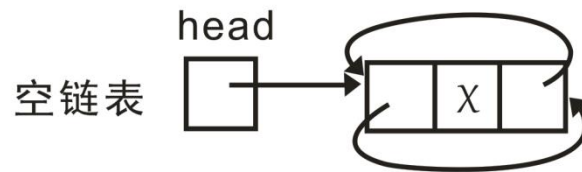
10.2.1.节点图示

12
NULL
NULL

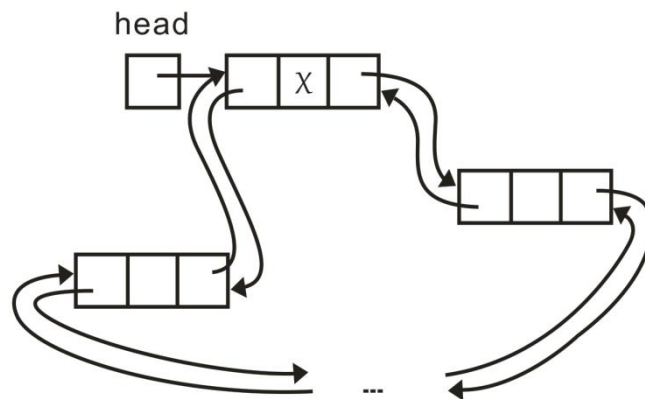
10.2.2.节点码示

```
typedef struct dnode
{
    int data;
    struct dnode *pre;
    struct dnode *next;
}DNode;
```

10.2.3.空链表示意



10.2.4.非空链表示意



10.3.双向循环链表静态形式

10.3.1.节点代码

```
typedef struct node
{
    int data;
    struct node *pre;
    struct node *next;
}DNode;
```

10.3.2.静态模拟

10.4.双向循环链表的操作

10.4.1.创建

10.4.1.1.尾插法

```
DNode * creatDList()
{
    DNode * head = (DNode*)malloc(sizeof(DNode));
    DNode * tail= head; //尾插法的一个特点是，必须有指针，指向最后一个节点
    DNode * cur = NULL;
    head->next = NULL;
    head->pre = NULL;
    int data;
    scanf("%d",&data);
    while(data)
    {
        cur = (DNode*)malloc(sizeof(DNode));
        cur->data = data;

        tail->next = cur;
        cur->pre = tail;
        tail= cur;

        scanf("%d",&data);
    }
    //完成回环
    cur->next = head;
    head->pre = cur;
    return head;
}
```

10.4.1.2.头插法

```
DNode * creatDList()
{
    DNode * head = (DNode*)malloc(sizeof(DNode));
    DNode * cur = NULL;
    head->next = head;
    head->pre = head;
    int data;
    scanf("%d",&data);
```

```
while(data)
{
    cur = (DNode*)malloc(sizeof(DNode));
    cur->data = data;

    //*****
    //此为添加第一个节点的逻辑，但是第二次循环不再适用。
    // cur->next = head;
    // cur->pre = head;
    // head->next = cur;
    // head->pre = cur;
    //*****

    //故将逻辑改为如下，结果兼容添加第一个结点的逻辑
    cur->next = head->next;
    cur->pre = head;

    head->next = cur;
    cur->next->pre = cur;

    scanf("%d",&data);
}
return head;
}
```

10.4.2.插入

```
void insertDList(DNode * head,int insertData)
{
    DNode * cur;
    cur = (DNode*)malloc(sizeof(DNode));
    cur->data = insertData;

    cur->next = head->next;
    cur->pre = head;

    head->next = cur;
    cur->next->pre = cur;
}
```

10.4.3.遍历

10.4.3.1.走 next

```
void traverseDList(DNode *head)
{
    DNode *phead = head->next;
    while (phead != head)
    {
        printf("%d ",phead->data);
        phead = phead->next;
    }
}
```

10.4.3.2.走 pre

10.4.3.3.求长度

```
int lenDList(DNode *head)
{
    int len = 0;
    DNode *phead = head->next;
    while(phead != head)
    {
        len++;
        phead = head->next;
    }
    return len;
}
```

10.4.4.查找

10.4.4.1.单向

```
DNode * searchDList(DNode * head,int find)
{
    DNode *phead = head->next;

    while(phead != head)
    {
        if(phead->data == find)
            return phead;
    }
}
```

```
    phead = head->next;
}
return NULL;
}
```

10.4.4.2.双向

```
DNode * searchList(DNode * head,int find)
{
    DNode * pClock = head->next;
    DNode * pAntiClock = head->pre;

    while (pAntiClock != pClock->pre)
    {
        if(pClock->data == find)
            return pClock;
        if(pAntiClock->data == find)
            return pAntiClock;

        if(pClock == pAntiClock) //交错而过 一定包含相等的逻辑在里面
            return NULL;

        pClock = pClock->next;
        pAntiClock = pAntiClock->pre;
    }
    return NULL;
}
```

10.4.5.删除节点

```
void deleteNodeDList(DNode * pfind)
{
    if(pfind == NULL)
        return;
    else
    {
        pfind->pre->next = pfind->next;
        pfind->next->pre = pfind->pre;

        free(pfind);
    }
}
```

```
}
```

10.4.6.排序

10.4.6.1.交换数据

```
void sortDList(DNode * head,int n)
{
    DNode * p,*q;
    for(int i=0; i<n-1; i++)
    {
        p = head->next;
        q = p->next;
        for(int j=0; j<n-1-i; j++)
        {
            if(p->data > q->data)
            {
                p->data = p->data ^ q->data;
                q->data = p->data ^ q->data;
                p->data = p->data ^ q->data;

            }
            p = p->next;
            q = q->next;
        }
    }
}
```

10.4.6.2.交换指针

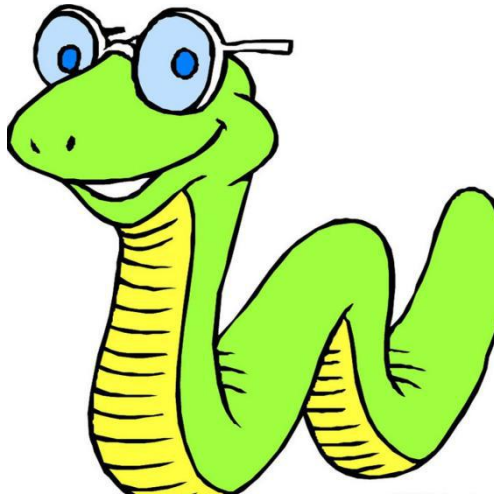
```
void sortDList(DNode * head,int n)
{
    DNode *p,*q,*tmp;
    for(int i=0; i<n-1; i++)
    {
        p = head->next;
        q = p->next;
        for(int j=0; j<n-1-i; j++)
        {
            if(p->data > q->data)
            {
```

```
    }  
    p = p->next;  
    q = q->next;  
}  
  
}  
}
```

10.4.7.销毁

```
void destroyDList(DNode * head)  
{  
    head->pre->next = NULL;  
    DNode * pre = head;  
    while(head != NULL)  
    {  
        head = head->next;  
        free(pre);  
        pre = head;  
    }  
}
```


10.5.链表应用-贪吃蛇



10.5.1.1.业务逻辑

10.5.1.2.图示

10.5.1.3.细化分析

10.5.2.MVC 架构分析

10.5.2.1.Model

10.5.2.2.View

10.5.2.3.Control

10.5.3.其它

10.5.3.1.非阻塞输入

10.5.3.2.精细延时

10.6.练习

10.6.1.用双向循环链表实现读文件到内存

用双向链表实现基础班中，读文件到链表，用界面实现其功能（显示，添加，删除，保存，退出）。

11.动态库与静态库(Library)

11.1.函数库

11.1.1.库存在的意义



"Stop Trying to Reinvent the Wheel", 不要重复造轮子。库存在的意义，就是避免重复造轮子。像我们使用的 `printf` 函数，没有必要每个使用此功能的函数，都要重写来实现。重复现成的、可用的、已经证明很好用的东西就是造轮子。

11.1.2.库的划分

函数库，大体上划分为两大类，一类是动态库，一类是静态库。

11.1.3.库的命名规则

静态库的名字一般是 `libxxx.a` 动态库的名字一般是 `libxxx.so` 有时候也是 `libxxx.so.major.minor`, `xxx` 是该 `lib` 的名称, `major` 是主版本号, `minor` 是副版本号。

比如在 `linux` 系统中常用的数学库的动静态版本: `libm.a libm.so`

11.1.4.静/动态库特点

静态库，会在编译阶段**完全被整合进代码段**中，所以生成的可执行性文件也比较大。这样一来，它的优点也显而易见了，即编译后的执行程序不再需要函数库的支持，因为所有要使用的函数已经被编译进去了。当然这也是他的一个缺点，第一，生成可执行性文件体积大，第二，静态库如果发生了改变那么你的程序必须要重新编译。

相对于静态函数库，动态函数库在编译的时候**并没有被编译进目标代码**中，你的程序执行到相关函数时才调用该函数库里的相应函数，因此动态函数库所产生的可执行文件比较小。由于函数库没有被整合进你的程序，而是程序运行时动态的申请并调用，所以程序的运行环境中必须提供相应的库。动态函数库的改变并不影响你的程序，所以动态函数库的升级比较方便。

11.2.linux 库存放的路径

`/lib /usr/lib`

```
wgl@NengZhong-vm:/opt$ sudo find / -name "*libm.a*" |xargs ls -lh
-rw-r--r-- 1 root root 700K 2月 17 03:19 /usr/lib/i386-linux-gnu/libm.a
wgl@NengZhong-vm:/opt$ sudo find / -name "*libm.so*" |xargs ls -lh
lrwxrwxrwx 1 root root 12 6月 12 2016 /lib/i386-linux-gnu/libm.so.6 ->
libm-2.19.so
lrwxrwxrwx 1 root root 29 6月 12 2016 /usr/lib/i386-linux-gnu/libm.so ->
/lib/i386-linux-gnu/libm.so.6
wgl@NengZhong-vm:/opt$ ls -lh /lib/i
i386-linux-gnu/ ifupdown/      init/
wgl@NengZhong-vm:/opt$ ls -lh /lib/i386-linux-gnu/libm-2.19.so
-rw-r--r-- 1 root root 274K 2月 17 03:19 /lib/i386-linux-gnu/libm-2.19.so
wgl@NengZhong-vm:/opt$
```

11.2.1.使用特点

当要使用静态的程序库时，连接器会找出程序所需的函数，然后将它们拷贝到执行文件，由于这种拷贝是完整的，所以一旦连接成功，静态程序库也就不再需要了。

然而，对动态库而言，就不是这样。动态库会在执行程序内留下一个标记指明当程序执行时，首先必须载入这个库。

当静态库和动态库同名时，`gcc` 命令将优先使用动态库。

显然动态库的优点更明显。

11.2.2.查看一个文件的库的使用情况

当我们拿到一个可执行文件，可以查看其执行过程中需要用到的库。

```
[root@localhost opt]# file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
[root@localhost opt]# ldd a.out
linux-gate.so.1 => (0x00e2f000)
libm.so.6 => /lib/libm.so.6 (0x00c39000)
libc.so.6 => /lib/libc.so.6 (0x00aa0000)
/lib/ld-linux.so.2 (0x00a7a000)
[root@localhost opt]#
```

11.3.标准库的使用

下面以使用标准库中的数学库为例：libm.a libm.so 原程序如下：

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6
7     double x = 3.14;
8     printf("sin(x) = %f\n",sin(x));
9     return 0;
10 }
```

11.3.1.libm.a

```
wgl@NengZhong-vm:/opt$ vim math.c
wgl@NengZhong-vm:/opt$ gcc math.c //缺少 math.h
math.c: In function 'main':
math.c:5:25: warning: incompatible implicit declaration of built-in function
'sin' [enabled by default]
printf("sin(x) = %f\n",sin(x));
                        ^
/tmp/ccooXXRf.o: In function `main':
math.c:(.text+0x1b): undefined reference to `sin'
collect2: error: ld returned 1 exit status
wgl@NengZhong-vm:/opt$ vim math.c
wgl@NengZhong-vm:/opt$ gcc math.c //缺少链接库
/tmp/ccFVtZWu.o: In function `main':
math.c:(.text+0x1b): undefined reference to `sin'
collect2: error: ld returned 1 exit status
wgl@NengZhong-vm:/opt$ gcc math.c -lm -static -o static //链接静态库
wgl@NengZhong-vm:/opt$ ls -lh
total 760K
-rw-rw-r-- 1 xmg xmg 115 6月 12 13:17 math.c
-rwxrwxr-x 1 xmg xmg 754K 6月 12 13:18 static
wgl@NengZhong-vm:/opt$ ./static //运行
sin(x) = 0.001593
```

11.3.2.lib.so

```
wgl@NengZhong-vm:/opt$ gcc math.c -lm -o dynamic //链接动态库
```

```
wgl@NengZhong-vm:/opt$ ls -lh //比较大小
total 768K
-rwxrwxr-x 1 xmg xmg 7.2K 6月 12 13:23 dynamic
-rw-rw-r-- 1 xmg xmg 115 6月 12 13:17 math.c
-rwxrwxr-x 1 xmg xmg 754K 6月 12 13:18 static
wgl@NengZhong-vm:/opt$ ./dynamic //运行
sin(x) = 0.001593
```

11.3.3.比较

11.3.3.1.libm.a

```
wgl@NengZhong-vm:/usr/lib/i386-linux-gnu$ sudo find / -name "*libm.a*"
|xargs ls -lh
[sudo] password for xmg:
-rw-r--r-- 1 root root 700K 2月 17 03:19 /usr/lib/i386-linux-gnu/libm.a
wgl@NengZhong-vm:/usr/lib/i386-linux-gnu$ sudo mv libm.a libm.a.bak
wgl@NengZhong-vm:/usr/lib/i386-linux-gnu$ mv libm.a.bak libm.a
```

```
wgl@NengZhong-vm:/opt$ ./static
sin(x) = 0.001593
```

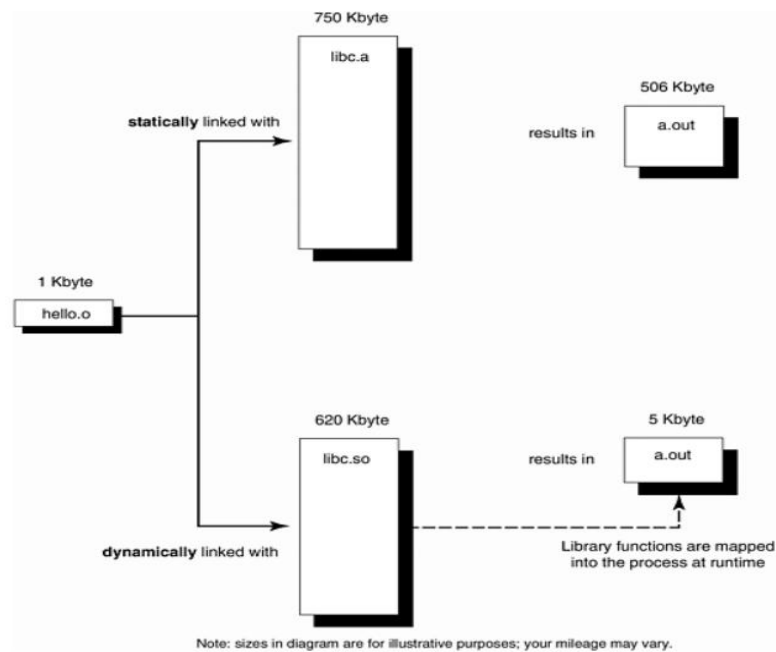
11.3.3.2.libm.so

```
wgl@NengZhong-vm:/usr/lib/i386-linux-gnu$ sudo find / -name "*libm.so*"
|xargs ls -lh
lrwxrwxrwx 1 root root 12 6月 12 2016 /lib/i386-linux-gnu/libm.so.6 ->
libm-2.19.so
lrwxrwxrwx 1 root root 29 6月 12 2016 /usr/lib/i386-linux-gnu/libm.so ->
/lib/i386-linux-gnu/libm.so.6

wgl@NengZhong-vm:/usr/lib/i386-linux-gnu$ cd /lib/i386-linux-gnu/
wgl@NengZhong-vm:/lib/i386-linux-gnu$ sudo mv libm-2.19.so
libm-2.19.so.bak
wgl@NengZhong-vm:/lib/i386-linux-gnu$ sudo mv libm-2.19.so.bak
libm-2.19.so
```

```
wgl@NengZhong-vm:/opt$ ./dynamic
./dynamic: error while loading shared libraries: libm.so.6: cannot open
shared object file: No such file or directory
wgl@NengZhong-vm:/opt$ ./dynamic
sin(x) = 0.001593
wgl@NengZhong-vm:/opt$
```

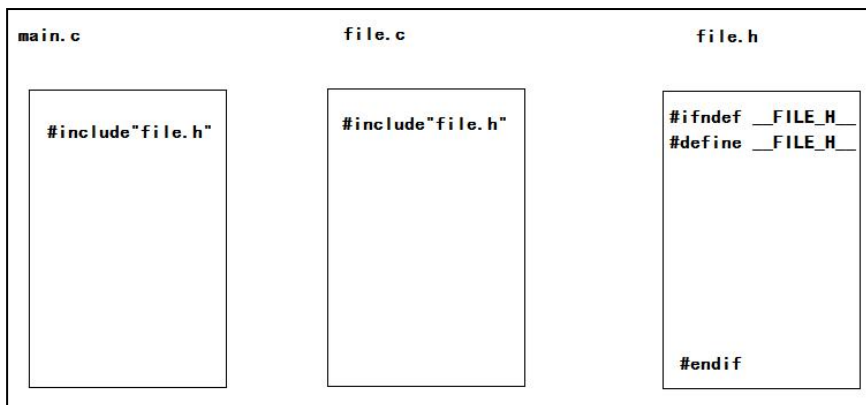
11.3.3.3. 图示



11.4.自定义库的生成与使用

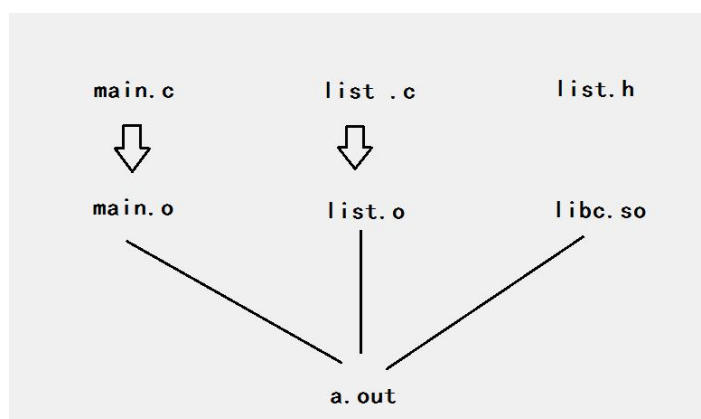
11.4.1.多文件编程

11.4.1.1. 书写方式



11.4.1.2.理论基础

编译期间，只需要函数声明就可以通过编译，在链接阶段提供函数实体就可以。



11.4.1.3. 实战案例

mystr.h

```
#ifndef __MYSTR_H__
#define __MYSTR_H__

int myStrlen(const char *src);

#endif
```

mystr.c

```
#include "mystr.h"
```



```
int myStrlen(const char * src)
{
    int len = 0;
    while(*src++)
    {
        len++;
    }
    return len;
}
```

main.c

```
#include <stdio.h>
#include "mystr.h"
int main(void)
{
    char * p = "china";
    int len = myStrlen(p);

    printf("len = %d\n",len);

    return 0;
}
```

11.4.2.静态库的生成与使用

11.4.2.1.步骤

- ① 将 **xx.c** 文件生成 **xx.o** 文件 `gcc -c xx.c`
- ② 将 **xx.o** 文件打包制作成 **libxx.a** 文件 `ar rc libxx.a xx.o`
- ③ 将 **libxx.a** 放到系统库，或指定链节路径下使用

11.4.2.2.实战

```
wgl@NengZhong-vm:/opt$ gcc -c mystr.c
wgl@NengZhong-vm:/opt$ ls
main.c mystr.c mystr.h mystr.o
wgl@NengZhong-vm:/opt$ ar cr libmystr.a mystr.o
wgl@NengZhong-vm:/opt$ ls
libmystr.a main.c mystr.c mystr.h mystr.o
wgl@NengZhong-vm:/opt$ gcc main.c -L. -lmystr
wgl@NengZhong-vm:/opt$ ./a.out
len = 5
wgl@NengZhong-vm:/opt$ mv libmystr.a libmystr.a.bak
```

- 111 -

```
wgl@NengZhong-vm:/opt$ ./a.out
len = 5
wgl@NengZhong-vm:/opt$ mv libmystr.a.bak libmystr.a
wgl@NengZhong-vm:/opt$ sudo mv libmystr.a /usr/lib/
wgl@NengZhong-vm:/opt$ gcc main.c -lmystr
wgl@NengZhong-vm:/opt$ ./a.out
len = 5
wgl@NengZhong-vm:/opt$ sudo mv /usr/lib/libmystr.a ./
wgl@NengZhong-vm:/opt$
```

11.4.3.动态库的生成与使用

11.4.3.1.步骤

- ① 将 **xx.c** 文件生成 **xx.o** 文件 `gcc -c xx.c`
- ② 将 **xx.o** 打包生成 **libxx.so** `gcc -shared -fPIC -o libxx.so xx.o`
- ③ 将 **libxx.a** 放到系统库，或指定链节路径下使用

11.4.3.2.实战

```
mg@xmg-vm:/opt$ gcc -c mystr.c
wgl@NengZhong-vm:/opt$ ls
main.c mystr.c mystr.h mystr.o temp
wgl@NengZhong-vm:/opt$ gcc -shared -fPIC -o libmystr.so mystr.o
wgl@NengZhong-vm:/opt$ ls
libmystr.so main.c mystr.c mystr.h mystr.o temp
wgl@NengZhong-vm:/opt$ gcc main.c
/tmp/ccSBzMF.o: In function `main':
main.c:(.text+0x19): undefined reference to `myStrlen'
collect2: error: ld returned 1 exit status
wgl@NengZhong-vm:/opt$ sudo mv libmystr.so /usr/lib/
wgl@NengZhong-vm:/opt$ gcc main.c -lmystr
wgl@NengZhong-vm:/opt$ ./a.out
len = 5
```

```
wgl@NengZhong-vm:/opt$ ls
a.out libmystr.so main.c mystr.c mystr.h mystr.o temp
wgl@NengZhong-vm:/opt$ gcc main.c -L. -lmystr
wgl@NengZhong-vm:/opt$ ./a.out
./a.out: error while loading shared libraries: libmystr.so: cannot open
shared object file: No such file or directory
```

```
wgl@NengZhong-vm:/opt$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
wgl@NengZhong-vm:/opt$ export LD_LIBRARY_PATH
wgl@NengZhong-vm:/opt$ ./a.out
len = 5
wgl@NengZhong-vm:/opt$
```

11.4.3.3. 链接路径

链接时找不到某一个函数，出现"xxxxx undefined"，应该就是链接时找不到相应的动态库，连接时动态库的路径如下：

- 1> 默认路径是/lib,/usr/lib/, /usr/local/lib
- 2> -L: 指定连接时动态库的路径
- 3> LD_LIBRARY_PATH: 指定连接路径

11.4.3.4. 运行路径

运行应用程序时出现 "error while loading shared libraries"，则是加载时找不到相应的动态库

- 1> 默认的动态库搜索路径/lib;
- 2> 默认的动态库搜索路径/usr/lib。
- 3> 环境变量 LD_LIBRARY_PATH 指定的动态库搜索路径;
- 4> 修改配置文件/etc/ld.so.conf 中指定的动态库搜索路径来改变,然后运行 ldconfig 更新一下, /etc/ld.so.cache

现在假设我们自定义的库放到了/usr/local/lib 下：解决方案如下

将动态库放到, /lib 或/usr/lib 中。

或 export LD_LIBRARY_PATH=/usr/local/lib:\$LD_LIBRARY_PATH，但是此种令在当前 shell 中有效。

或 vim /etc/ld.so.conf 中添加 /usr/local/lib 这一行，然后 ldconfig 更新缓存

11.5.gcc 的编译选项

参数	意义
-shared	指定生成动态链接库。

-fPIC	表示编译为位置独立的代码，用于编译共享库。目标文件需要创建成 位置无关码 ，概念上就是在可执行程序装载它们的时候，它们可以放在可执行程序的内存里的任何地方。
-L	表示要连接的库所在的目录。
-l	指定链接时需要的动态库。编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上 lib ，后面加上 .a/.so 来确定库的名称。

11.6.Linux 平台编译 sqlite3

11.6.1. 编译动态库

编译命令行管理工具：

```
gcc shell.c sqlite3.c -lpthread -ldl -o sqlite3
```

编译 SQLite 为单独的动态链接库：

```
gcc sqlite3.c -lpthread -ldl -fPIC -shared -o libsqlite3.so
```

-fPIC:

表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

-shared:

表示生成一个共享目标文件（让连接器生成 T 类型的导出符号表，有时候也生成弱连接 W 类型的导出符号），即我们所说的动态链接库。它可以和其他目标文件连接产生可执行文件。只有部分系统支持该选项。

pthread 系统库用于确保 SQLite 是线程安全的。但因为命令行工具是单线程的，对命令行工具则可编译成非线程安全的，以忽略 pthread 库。命令为 `gcc -DSQLITE_THREADSAFE=0 shell.c sqlite3.c -ldl -o sqlite3`。dl 系统库用于支持动态装载，sqlite3_load_extension()接口和 SQL 函数 load_extension()需要用到它。如果不需要这些特性，可以使用 SQLITE_OMIT_LOAD_EXTENSION 编译选项来忽略，如 `gcc -DSQLITE_THREADSAFE=0 -DSQLITE_OMIT_LOAD_EXTENSION shell.c sqlite3.c -o sqlite3`。

使用动态库 libsqlite3.so：在你的程序中（例如 test.c）通过包含头文件 sqlite3.h 来使用库中的函数，编译程序的命令为 `gcc test.c -L. -lsqlite3 -o test`。其中 -L. 表示让链接库的搜索路径包含当前目录，-lsqlite3 指明编译器查找动态库 libsqlite3.so，编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上 **lib**，后面加上 **.so** 来确定库的名称。通过 `ldd test` 可查看 test 程序是如何调用动态库中的函数的。

调用动态库时有几个问题会经常碰到。有时明明已经将库的头文件所在目录通过 "-I" include 进来了，库所在文件通过 "-L" 参数引导，并指定了 "-l" 的库名，但通过 ldd 命令察看时，就是死活找不到你指定链接的 so 文件，这时你要作的就是修改 LD_LIBRARY_PATH，这个环境变量指示动态连接器可以装载动态库的路径。或者修改 /etc/ld.so.conf 文件，然后调用/sbin/ldconfig 来达到同样的目的。通常这样做就可以解决库无法链接的问题了。

11.6.2. 编译成静态库：

```
gcc -c sqlite3.c -lpthread -ldl -o sqlite3.o
//编译成目标文件
ar -r libsqlite3.a sqlite3.o
//将列出的各个目标文件一起打包成一个静态库 libsqlite3.a
链接静态库：
gcc test.c -L. -lsqlite3 -static -o test    //也可不加-static 选项
```

12.配置文件读写(File)

目录为止，大家已经学习文件的读写操作和链表的操作。现在我们要求读写用户配置文件生成链表，完成登录的功能。

我们所要读写的文件是，著名的版本管理控制软件 SVN 的配置文件，文件样式如下：

```
### This file is an example password file for svnserve.  
### Its format is similar to that of svnserve.conf. As shown in the  
### example below it contains one section labelled [users].  
### The name and password for each user follow, one account per line.  
  
[users]  
harry =   harryssecret  
sally  =  sallyssecret
```

12.1.文件回顾

12.1.1.文件基础

12.1.1.1.流

12.1.1.2.文本文件与二进制文件

12.1.1.3.缓冲

12.1.2.文件操作

12.1.2.1.一次读写一个字符

12.1.2.2.一次读写一行字符

12.1.2.3.一次读写一块

12.1.2.4.格式化读写

12.2.链表回顾

12.2.1.节点

12.2.2.创建

12.2.3.插入

12.2.4.求长

12.2.5.查找

12.2.6.删除

12.2.7.排序

12.2.8.销毁

12.3.字符串处理回顾

12.3.1.常规处理

12.3.2.去空处理

12.4.实现登录系统

12.4.1.链表节点设计

```
typedef struct node
{
    char *name;
    char *passwd;
```

```
}LoginNode;
```

12.4.2.读文件并处理字符串

12.4.3.登录

13.本地数据库(SQLite)

13.1.本章综旨

- 建立数据库的基本概念
- 简单数据库的命令行操作及图形化管理
- QT/VS 中第三方库的配置
- 简单 C 语言操作数据库及对多级指针的深化认知

13.2.SQLite 简介

SQLite 是一个开源的嵌入式关系数据库，实现自包容、零配置、支持事务的 SQL 数据库引擎。其特点是高度便携、使用方便、结构紧凑、高效、可靠。与其他数据库管理系统不同，SQLite 的安装和运行非常简单，在大多数情况下 - 只要确保 SQLite 的二进制文件存在即可开始创建、连接和使用数据库。如果您正在寻找一个嵌入式数据库项目或解决方案，SQLite 是绝对值得考虑。

13.3.数据库基本概念

数据库管理软件	excel	sqlite
数据库	xx.xls	mydb
表	sheet	table

开发接口	交互语言	数据库文件	交互软件	人机交互
FILE*		xx.txt	nodepad++ sublime UE	
VBA		xx.excel	excel	
Sqlite3*	SQL	my.db	Navicat	

13.4.sqlite 官网

13.4.1.下载

官网下载：<http://www.sqlite.org/download.html>

官网文档：<http://www.sqlite.org/docs.html>

Source Code

[sqlite-amalgamation-3130000.zip](#) (1.88 MiB) C source code as an [amalgamation](#), version 3.13.0.
(sha1: b46e199f06aa6f644989076da40227da68db7b6a)

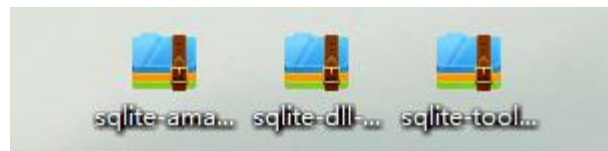
Precompiled Binaries for Windows

[sqlite-dll-win32-x86-3130000.zip](#) (422.71 KiB) 32-bit DLL (x86) for SQLite version 3.13.0.
(sha1: b1796bd0c2e7eaed79d1cd8d5dd90e02d491b43d)

[sqlite-dll-win64-x64-3130000.zip](#) (700.46 KiB) 64-bit DLL (x64) for SQLite version 3.13.0.
(sha1: e1254b108e884e09552f8e57104724b5f756a6ce)

Documentation

[sqlite-doc-3130000.zip](#) (5.29 MiB) Documentation as a bundle of static HTML files.
(sha1: 1f10c30709c67bdb62d0d86f2b36c0148f64ed21)



13.4.2.Navicat for SQLite

13.5.SQLite 管理操作

13.5.1.进入 sqlite3 交互模式

```
C:\Users\wgl\Desktop\sqlite>sqlite3.exe mydb
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> .databases
seq  name          file
-----
0     main           C:\Users\wgl\Desktop\sqlite\mydb
sqlite> .exit

C:\Users\wgl\Desktop\sqlite>sqlite3.exe
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open yourdb
sqlite> .databases
seq  name          file
-----
0     main           C:\Users\wgl\Desktop\sqlite\yourdb
sqlite> .exit

C:\Users\wgl\Desktop\sqlite>
```

```

sqlite> .help
.auth ON|OFF          Show authorizer callbacks
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.binary on|off         Turn binary output on or off.  Default OFF
.changes on|off        Show number of rows changed by SQL
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dbinfo ?DB?          Show status information about the database
.dump ?TABLE? ...      Dump the database in an SQL text format
                        If TABLE specified, only dump tables matching
                        LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.eqp on|off|full       Enable or disable automatic EXPLAIN QUERY PLAN
.exit                 Exit this program
.explain ?on|off|auto? Turn EXPLAIN output mode on or off or to automatic
.fullschema ?--indent? Show schema and the content of sqlite_stat tables
.headers on|off        Turn display of headers on or off
.help                 Show this message
.import FILE TABLE    Import data from FILE into TABLE
.indexes ?TABLE?       Show names of all indexes
                        If TABLE specified, only show indexes for tables
                        matching LIKE pattern TABLE.
.limit ?LIMIT? ?VAL?   Display or change the value of an SQLITE_LIMIT
.load FILE ?ENTRY?     Load an extension library
.log FILE|off          Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                        ascii  Columns/rows delimited by 0x1F and 0x1E
                        csv    Comma-separated values
                        column Left-aligned columns.  (See .width)
                        html   HTML <table> code
                        insert  SQL insert statements for TABLE
                        line   One value per line
                        list   Values delimited by .separator strings
                        tabs   Tab-separated values
                        tcl    TCL list elements
.nullvalue STRING      Use STRING in place of NULL values
.once FILENAME         Output for the next SQL command only to FILENAME
.open ?FILENAME?       Close existing database and reopen FILENAME
.output ?FILENAME?     Send output to FILENAME or stdout
.print STRING...       Print literal STRING
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                 Exit this program
.read FILENAME         Execute SQL in FILENAME
.restore ?DB? FILE     Restore content of DB (default "main") from FILE
.save FILE             Write in-memory database into FILE
.scanstats on|off      Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?PATTERN?     Show the CREATE statements matching PATTERN
                        Add --indent for pretty-printing
.separator COL ?ROW?   Change the column separator and optionally the row
                        separator for both the output mode and .import

```

解析：

- ① **sqlite3.exe [dbname]** 打开 **sqlite3** 的交互模式，[并创建数据库]。
- ② **.open dbname** 若未创建数据，可以创建数据库，若已创建则打开。
- ③ **.databases** 显示当前打开的数据库文件
- ④ **.exit** 退出交互模式。
- ⑤ **.help**, 列出命令的提示信息，可供于查阅。

13.5.2.创建销毁表

13.5.2.1.示例

```
sqlite> .open mydb
sqlite> create table student(idx integer,name text,age integer);
sqlite> .tables
student
sqlite> .schema
CREATE TABLE student(idx integer,name text,age integer);
sqlite> create table teacher(num integer,name text, age integer);
sqlite> .tables
student  teacher
sqlite> .schema
CREATE TABLE student(idx integer,name text,age integer);
CREATE TABLE teacher(num integer,name text, age integer);
sqlite> drop table teacher;
sqlite> .tables
student
sqlite>
```

13.5.2.2.解析：

- ① `.open mydb` 打开要创建表的那个数据库。
- ② `create table <table_name> (field1 type1, field2 type2,...);` 建表语句。
- ③ `.tables` 查看当前数据库下所有的表。
- ④ `.schema [tbname]` 查看表结构(主要看列信息)。
- ⑤ `drop table <table_name>;` 销毁表的语句。

13.5.3.列类型

SQLite 支持的类型相对比较简单，每个存储在 SQLite 数据库中的值都具有以下存储类之一。

存储类	描述
NULL	值是一个 NULL 值。
INTEGER	值是一个带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。
REAL	值是一个浮点值，存储为 8 字节的 IEEE 浮点数字。
TEXT	值是一个文本字符串，使用数据库编码（UTF-8、 UTF-16BE 或 UTF-16LE）存储。
BLOB	值是一个 blob 数据，完全根据它的输入存储。

13.5.4.列约束

建表语句中，可以在 `type` 的后面添加，`column-constraint` 称为列约束，常见的列约束。

约束	语义	详解
PRIMARY KEY	主键	主键必须包含唯一的值, 主键列不能包含 NULL 值。
ASC (AUTOINCREMENT)	自增	默认地, AUTOINCREMENT 的开始值是 1, 每条新记录递增 1
NOT NULL	不为空	
DEFAULT <i>value</i>	默认值	
UNIQUE	值唯一	

```
sqlite> CREATE TABLE COMPANY(  
    ID INTEGER PRIMARY KEY AUTOINCREMENT,  
    NAME      TEXT    NOT NULL,  
    AGE       INT     NOT NULL,  
    ADDRESS   CHAR(50),  
    SALARY    REAL  
);
```

13.5.5.插入与查询

```
sqlite> .tables
student
sqlite> insert into student values(1001,"zhangsan",23);
sqlite> insert into student values(1002,"lisi",23);
sqlite> insert into student values(1003,"wangwu",56);
sqlite> select idx,name,age from student;
1001|zhangsan|23
1002|lisi|23
1003|wangwu|56
sqlite> select idx,age from student;
1001|23
1002|23
1003|56
sqlite> .header on
sqlite> select * from student;
idx|name|age
1001|zhangsan|23
1002|lisi|23
1003|wangwu|56
sqlite> .mode column
sqlite> select * from student;
idx      name      age
-----
1001      zhangsan  23
1002      lisi      23
1003      wangwu    56
sqlite>
```

解析：

- ① **.tables** 插入数据之前先确定，所要插入表的存在。
- ② **insert into <table_name> values (value1, value2,...);**向表中添加新记录
insert into <table_name>(key1, key2) values(value1, value2); 值和名字对应即可
- ③ **select filed1,field2 ... from <table_name>;**查询表中某些字段的记录。
- ④ **select * from <table_name>;**查询表中所有字段的记录。
- ⑤ **.header on** 显示字段名称。
- ⑥ **.mode column** 以列模式显示字段的记录。默认是 list 模式

13.5.6.排序

```
sqlite> select * from student;
idx      name      age
-----
1001     zhangsan   23
1002     lisi       23
1003     wangwu     56
sqlite> select * from student order by idx desc;
idx      name      age
-----
1003     wangwu     56
1002     lisi       23
1001     zhangsan   23
sqlite> select * from student order by idx asc;
idx      name      age
-----
1001     zhangsan   23
1002     lisi       23
1003     wangwu     56
sqlite>
```

解析：

- ① **order by field desc|asc** 此子句，可以用于表达排序，**desc** 表示降序，**asc** 表示升序。

13.5.7.修改与删除记录

```
sqlite> select * from student;
idx      name      age
-----
1001     zhangsan   23
1002     lisi       23
1003     wangwu     56
sqlite> update student set name="bob";
sqlite> select * from student;
idx      name      age
-----
1001     bob        23
1002     bob        23
1003     bob        56
sqlite> update student set name="jim" where idx=1001;
sqlite> select * from student;
idx      name      age
-----
1001     jim        23
1002     bob        23
1003     bob        56
sqlite> delete from student where idx=1001;
sqlite> select * from student;
idx      name      age
-----
1002     bob        23
1003     bob        56
sqlite>
```


解析：

- ① **update**
`<table_name> set <field1=value1>,<field2=value2>...where <expression>;`
更新表中记录。若没有 **where** 子句，则会全部修改。
- ② **delete from** `<table_name> where <expression>;` 若没有 **where** 子句，则会删全表内容，但不同于 **drop**。

13.5.8.常用函数

13.5.9. 备份与恢复

13.5.9.1.备份数据库

```
---通过在命令行窗口下执行 sqlite3.exe 以重新建立和 SQLite 的连接。  
sqlite3 test.db  
sqlite> .backup 'c:/test.db'  
sqlite> .exit
```

13.5.9.2.恢复数据库

```
sqlite3 test.db  
sqlite> .restore 'c:/test.db'
```

13.5.9.3.复制数据库

```
sqlite3 test.db  
sqlite> .clone 'c:/newtest.db'  
sqlite> .exit
```

13.6.Vs/Qt+SQLite 配置

13.6.1.window 中 lib /dll 的关系

首先介绍一下静态库（静态链接库）、动态库（动态链接库）的概念，首先两者都是代码共享的方式。

静态库：

在链接步骤中，连接器将从库文件取得所需的代码，复制到生成的可执行文件中，这种库称为静态库，其特点是可执行文件中包含了库代码的一份完整拷贝；缺点就是被多次使用就会有多份冗余拷贝。即静态库中的指令都全部被直接包含在最终生成的 EXE 文件中了。在 vs 中新建生成静态库的工程，编译生成成功后，只产生一个.lib 文件

动态库：

动态链接库是一个包含可由多个程序同时使用的代码和数据的库，DLL 不是可执行文件。动态链接提供了一种方法，使进程可以调用不属于其可执行代码的函数。函数的可执

行代码位于一个 DLL 中，该 DLL 包含一个或多个已被编译、链接并与使用它们的进程分开存储的函数。在 vs 中新建生成动态库的工程，编译成功后，产生一个.lib 文件和一个.dll 文件

那么上述静态库和动态库中的 lib 有什么区别呢？

静态库中的 lib:

该 LIB 包含函数代码本身（即包括函数的索引，也包括实现），在编译时直接将代码加入程序当中

动态库中的 lib:

该 LIB 包含了函数所在的 DLL 文件和文件中函数位置的信息（索引），函数实现代码由运行时加载在进程空间中的 DLL 提供。'

总之，lib 是编译时用到的，dll 是运行时用到的。如果要完成源代码的编译，只需要 lib；如果要使动态链接的程序运行起来，只需要 dll。

13.6.2.生成 lib 文件

13.6.2.1.填加环境变量

C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin

注：win10 需要重启

13.6.2.2.生成 lib 库

```
D:\mySqlite\sqlite-dll-win32-x86-3130000>lib.exe
Microsoft (R) Library Manager Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.
用法: LIB [选项] [文件]

选项:
    /DEF[:文件名]
    /ERRORREPORT:{NONE|PROMPT|QUEUE|SEND}
    /EXPORT:符号
    /EXTRACT:成员名
    /INCLUDE:符号
    /LIBPATH:目录
    /LIST[:文件名]
    /LTCG
    /MACHINE:{ARM|ARM64|EBC|X64|X86}
    /NAME:文件名
    /NODEFAULTLIB[:库]
    /NOLOGO
    /OUT:文件名
    /REMOVE:成员名
    /SUBSYSTEM:{BOOT_APPLICATION|CONSOLE|EFI_APPLICATION|
                EFI_BOOT_SERVICE_DRIVER|EFI_ROM|EFI_RUNTIME_DRIVER|
```

```
NATIVE|POSIX|WINDOWS|WINDOWSCE}[,#[.##]]
/VERBOSE
/WX[:NO]
D:\mySqlite\sqlite-dll-win32-x86-3130000>lib.exe /def:sqlite3.def
/machine:x86
Microsoft (R) Library Manager Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.
正在创建库 sqlite3.lib 和对象 sqlite3.exp
```

13.6.3.建立三方库相关的文件夹

在当前工程路径下，分别建两个文件夹，`sqlh` 和 `sqlLib`，这样作的目的为了避免同其它的三方库混淆。然后将 `sqlite3.h` 和 `sqlite3.lib` 分别放入其中，并将 `sqlite3.dll` 放到可执行文件夹中去。

13.6.4.配置当前工程的 pro 文件

上述工作完成后，配置当前工程的 `pro` 文件。如下：

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += main.c

include(deployment.pri)
qtcAddDeployment()

INCLUDEPATH += sqlh
LIBS += -L $$_PRO_FILE_PWD_/sqlLib -l sqlite3

QMAKE_CFLAGS += -std=c99
```

13.6.5.测试环境

```
#include <stdio.h>
#include "sqlh/sqlite3.h"

int main(int argc, char **argv){
    sqlite3 *db;
```

```

char *zErrMsg = 0;
int rc;

rc = sqlite3_open("mydb", &db);
if( rc ){
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    return -1;
}
sqlite3_close(db);
return 0;
}

```

13.7.C/C++操作 SQLite

13.7.1.常用 API 详解

13.7.1.1.sqlite3_open

函数声明	int sqlite3_open(const char *filename, sqlite3 **ppDb);	
所在文件	sqlite3.h	
函数功能	打开一个数据库，文件名不一定要存在，如果此文件不存在，sqlite 会自动创建	
参数及返回解析		
参数	const char* sqlite3 **	指文件名 结构体指针（关键数据结构）
返回值	int	表示操所是否正确（SQLITE_OK 操作正常）

13.7.1.2.sqlite3_close

函数声明	int sqlite3_close(sqlite3* db);	
所在文件	sqlite3.h	
函数功能	如果用 sqlite3_open 开启了一个数据库，结尾时不要忘了用这个函数关闭数据库。	
参数及返回解析		
参数	sqlite3*	数据库句柄

返回值	int	表示操作是否正确（SQLITE_OK 操作正常）
-----	-----	--------------------------

13.7.1.3.sqlite3_exec

函数声明	int sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *, char **errmsg);	
所在文件	sqlite3.h	
函数功能	这个函数的功能是执行一条或者多条 SQL 语句， SQL 语句之间用";"号隔开。通常 sqlite3_callback 和它后面的 void*这两个位置都可以填 NULL，表示不需要回调。比如做 insert 操作，做 delete 操作，就没有必要使用回调。而当作 select 时，就要使用回调，因为 sqlite3 把数据查出来，得通过回调告诉你查出了什么数据。	
参数及返回解析		
参数	sqlite3* const char* sqlite_callback void *, char **	数据库句柄 待执行的 sql 语句 回调函数 传入参数 输出错误信息
返回值	int	表示操所是否正正确（SQLITE_OK 操作正常）

函数声明	typedef int (*sqlite3_callback)(void* data, int argc, char** argv, char** column);	
所在文件	sqlite3.h	
函数功能	回调函数必须定义为这个函数的类型	
参数及返回解析		
参数	void * int char ** char **	Data provided in the 4th argument of sqlite3_exec() The number of columns in row An array of strings representing fields in the row An array of strings representing column names

返回值	int	在回调函数中可以获得执行 Sql 得详细过程，如果所有 Sql 执行完毕则应该返回 0，否则，则说明这次执行并没有完全成功
-----	-----	---

13.7.1.4.sqlite3_get_table

函数声明	int sqlite3_get_table(sqlite3*, const char *sql, char ***resultp, int *nrow, int *ncolumn, char **errmsg)	
所在文件	sqlite3.h	
函数功能	执行一次查询 Sql 并且返回得到一个记录集。	
参数及返回解析		
参数	sqlite3* const char * char *** int * int * char **	句柄 sql 它是一维数组，内存布局为：第一行是字段名称，后面是紧接着是每个字段的值 行数 列数 错误信息
返回值	int	表示操所是否正确（SQLITE OK 操作正常）

13.7.2.实战增/删/改查

13.7.2.1.代码

```
#include <stdio.h>
#include "sqlite3h/sqlite3.h"

int myCallBack(void*p ,int argc ,char**argv , char**col)
{
    for(int i=0; i<argc; i++)
    {
        printf("%s %s\n",col[i],argv[i]);
    }
    return 0; //调用成功返回 0
}

int main(void)
```

```
{
    sqlite3 *db;
    int rc = sqlite3_open("yourdb",&db);
    if(rc != SQLITE_OK)
    {
        printf("open error\n");
        return -1;
    }

    char *sql =
        "create table tab(id integer primary key, name varchar(20),age integer)";
    sqlite3_exec(db,sql,NULL,NULL,NULL);

    sql = "insert into tab values(1001,\"liudehua\",54)";
    sqlite3_exec(db,sql,NULL,NULL,NULL);

    sql = "insert into tab values(1002,\"liming\",55)";
    sqlite3_exec(db,sql,NULL,NULL,NULL);

    sql = "insert into tab values(1003,\"zhangxueyou\",44)";
    sqlite3_exec(db,sql,NULL,NULL,NULL);

    sql = "insert into tab values(1003,\"guofucheng\",59)";
    sqlite3_exec(db,sql,NULL,NULL,NULL);

    sql = "select * from tab";

    sqlite3_exec(db,sql,myCallBack,NULL,NULL);

    sqlite3_close(db);
    return 0;
}
```

13.7.2.2.表信息格式

```
char ** ColName
    char* id
    char* name
    char* age
    char* addr

char **argv
```

```
char* 1
char* tangseng
char* 18
char* henan

char ** ColName
    char* id
    char* name
    char* age
    char* addr

char **argv
    char* 1
    char* wukong
    char* 18
    char* henan
```

13.7.3.获得表 sqlite3_get_table

13.7.3.1.代码

```
sql = "select * from company";

char * err;
int row, column;

char **table;

sqlite3_get_table(
    db,
    sql,
    &table,
    &row,
    &column,
    &err
);

for(int i=0; i<column*(row+1); i++)
{
    printf("%10s\t",table[i]);
}
```



```
        if(!((i+1)%column))
            putchar(10);
    }
    //按行列的方式访问
    for(int i=0; i<(row+1); i++)
    {
        for(int j=0; j<line; j++)
        {
            printf("%-10s", table[i*row+line]);
        }
        putchar(10);
    }
```

13.7.3.2.表信息格式

```
char **
    char * id
    char * name
    char * age
    char * addr

    char * 1
    char * teacherkang
    char * 18
    char * henan

    char * 2
    char * teacherwang
    char * 18
    char * shangdong
```

13.7.4.防注入

14.C 语言中的范型(Generic)

14.1.void 系列

14.2.macro 系列

14.3.union

15.附录(Append)

15.1.ascii 码表

ASCII							
ASCII	字符	ASCII	字符	ASCII	字符	ASCII	字符
0	NUL	32	space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	TAB	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z

27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	

15.2.c 运算符优先级

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){ list }	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	= = ! =	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	? :	Ternary conditional	Right-to-Left
14	=	Simple assignment	Left-to-right
	+= -=	Assignment by sum and difference	
	*= /= % =	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

15.3.SVN 配置与启动

15.3.1.官方下载地址

<https://tortoisetsvn.net/downloads.html>

for 32-bit OS

TortoiseSVN 1.9.4 - 32-bit

for 64-bit OS

TortoiseSVN 1.9.4 - 64-bit

15.3.2.创建服务配置文件

```
C:\Program Files\TortoiseSVN\bin>svnadmin.exe create d:\svnrepo
```

15.3.3.配置配置文件

15.3.3.1.svnserve.conf

```
auth-access = write           //授权用户可写
password-db = passwd          //开启用户名和密码
authz-db = authz              //设置用权限
```

15.3.3.2.passwd

```
[users]
# harry = harryssecret
# sally = sallyssecret
wgl = wgl
```

15.3.3.3.authz

```
# [/foo/bar]
# harry = rw
# &joe = r
# * =

# [repository:/baz/fuz]
# @harry_and_sally = rw
# * = r
[/]
wgl = rw
```

15.3.4.启动服务

```
C:\Program Files\TortoiseSVN\bin>svnserve.exe -d -r d:\svnrepo
```

15.3.5.创建与删除服务

15.3.5.1.createservice.bat

```
sc create mysvn binPath="C:\Program Files\TortoiseSVN\bin\svnserve.exe  
--service -r d:\svnrepo" displayname="mysvn"
```

15.3.5.2.deleteservice.bat

```
sc delete mysvn
```

15.3.6.更新与提交

```
svn://192.168.xxx.xxx
```

15.4.netplan

15.4.1.配置

15.4.1.1.配置文件

```
root@nzhsoft:/etc/netplan# vim 01-network-manager-all.yaml
```

15.4.1.2.图示与文字

```
1 # Let NetworkManager manage all devices on this system
2 network:
3   version: 2
4   renderer: networkd
5   ethernets:
6     ens33:
7       dhcp4: yes
8       addresses: [192.168.8.111/24]
9       gateway4: 192.168.8.1
10      nameservers:
11        addresses: [114.114.114.114,8.8.8.8]
12
```

```
# Let NetworkManager manage all devices on this system
network:
  version: 2
  renderer: networkd
  ethernets:
    ens33:
      dhcp4: yes
      addresses: [192.168.8.111/24]
      gateway4: 192.168.8.1
```

```
nameservers:  
    addresses: [114.114.114.114,8.8.8.8]
```

15.4.1.3.规则

1. 配置文件里在冒号：号出现的后面一定要空一格，不空格则在运行 netplan apply 时提示出错。
2. 关键之关键是看清配置总共分为五个层次，逐层向后至少空一格。
3. 若需要注释，需要用到#，比如 #addresses: [192.168.8.111/24]
4. yes 的地方，可以填 no

15.4.2.启动

```
# netplan apply
```

15.4.3.其它

15.4.3.1.查看 80 端口

```
root@nzhsoft:/usr/local/openresty/nginx/sbin# netstat -nulp | grep 80
```

15.4.3.2.apache2

```
systemctl disable apache2.service  
systemctl stop apache2.service
```

15.5.TestSql

```
BEGIN TRANSACTION;  
CREATE TABLE Cars(Id integer PRIMARY KEY, Name text, Cost integer);  
INSERT INTO Cars VALUES(1,'Audi',52642);  
INSERT INTO Cars VALUES(2,'Mercedes',57127);  
INSERT INTO Cars VALUES(3,'Skoda',9000);  
INSERT INTO Cars VALUES(4,'Volvo',29000);  
INSERT INTO Cars VALUES(5,'Bentley',350000);  
INSERT INTO Cars VALUES(6,'Citroen',21000);  
INSERT INTO Cars VALUES(7,'Hummer',41400);  
INSERT INTO Cars VALUES(8,'Volkswagen',21600);  
COMMIT;  
BEGIN TRANSACTION;  
CREATE TABLE Orders(Id integer PRIMARY KEY, OrderPrice integer  
CHECK(OrderPrice>0),  
Customer text);  
INSERT INTO Orders(OrderPrice, Customer) VALUES(1200, 'Williamson');  
INSERT INTO Orders(OrderPrice, Customer) VALUES(200, 'Robertson');
```

```
INSERT INTO Orders(OrderPrice, Customer) VALUES(40, 'Robertson');
INSERT INTO Orders(OrderPrice, Customer) VALUES(1640, 'Smith');
INSERT INTO Orders(OrderPrice, Customer) VALUES(100, 'Robertson');
INSERT INTO Orders(OrderPrice, Customer) VALUES(50, 'Williamson');
INSERT INTO Orders(OrderPrice, Customer) VALUES(150, 'Smith');
INSERT INTO Orders(OrderPrice, Customer) VALUES(250, 'Smith');
INSERT INTO Orders(OrderPrice, Customer) VALUES(840, 'Brown');
INSERT INTO Orders(OrderPrice, Customer) VALUES(440, 'Black');
INSERT INTO Orders(OrderPrice, Customer) VALUES(20, 'Brown');
COMMIT;
BEGIN TRANSACTION;
CREATE TABLE Friends(Id integer PRIMARY KEY, Name text UNIQUE NOT NULL,
                      Sex text CHECK(Sex IN ('M', 'F')));
INSERT INTO Friends VALUES(1,'Jane', 'F');
INSERT INTO Friends VALUES(2,'Thomas', 'M');
INSERT INTO Friends VALUES(3,'Franklin', 'M');
INSERT INTO Friends VALUES(4,'Elisabeth', 'F');
INSERT INTO Friends VALUES(5,'Mary', 'F');
INSERT INTO Friends VALUES(6,'Lucy', 'F');
INSERT INTO Friends VALUES(7,'Jack', 'M');
COMMIT;
BEGIN TRANSACTION;
CREATE TABLE Customers(CustomerId integer PRIMARY KEY, Name text);
INSERT INTO Customers(Name) VALUES('Paul Novak');
INSERT INTO Customers(Name) VALUES('Terry Neils');
INSERT INTO Customers(Name) VALUES('Jack Fonda');
INSERT INTO Customers(Name) VALUES('Tom Willis');
CREATE TABLE Reservations(Id integer PRIMARY KEY,
                           CustomerId integer, Day text);
INSERT INTO Reservations(CustomerId, Day) VALUES(1, '2009-22-11');
INSERT INTO Reservations(CustomerId, Day) VALUES(2, '2009-28-11');
INSERT INTO Reservations(CustomerId, Day) VALUES(2, '2009-29-11');
INSERT INTO Reservations(CustomerId, Day) VALUES(1, '2009-29-11');
INSERT INTO Reservations(CustomerId, Day) VALUES(3, '2009-02-12');
COMMIT;
BEGIN TRANSACTION;
CREATE TABLE Names(Id integer, Name text);
INSERT INTO Names VALUES(1,'Tom');
INSERT INTO Names VALUES(2,'Lucy');
INSERT INTO Names VALUES(3,'Frank');
INSERT INTO Names VALUES(4,'Jane');
INSERT INTO Names VALUES(5,'Robert');
```



```
COMMIT;  
BEGIN TRANSACTION;  
CREATE TABLE Books(Id integer PRIMARY KEY, Title text, Author text,  
                     Isbn text default 'not available');  
INSERT INTO Books VALUES(1, 'War and Peace', 'Leo Tolstoy', '978-0345472403');  
INSERT INTO Books VALUES(2, 'The Brothers Karamazov',  
                           'Fyodor Dostoyevsky', '978-0486437910');  
INSERT INTO Books VALUES(3, 'Crime and Punishment',  
                           'Fyodor Dostoyevsky', '978-1840224306');  
COMMIT;
```