# .函数

### 7.1 函数定义

1. 一个简单的函数大致如下：

function ( ) {

something;

}

这段书上翻译太拗口直接摘英文，this function simply returns,it severs a useful purpose as a stub(stub : the small unused part of something).

1. 没有返回值的函数在声明中应该把函数的类型声明为void
2. 调用一个真函数（即返回一个值的函数）/\*好吧，第一次听说这个定义\*/
3. 从表达式内部调用一个过程类型的函数（无返回值的函数）是一个严重错误，因为这样在表达式的求值过程中会使用一个不可预测的值（垃圾）。

### 7.2 函数声明

1. 如果没有关于调用函数的特定信息，编译器便假设在这个函数的调用是参数的类型和数量是正确的。它同时会返回一个整型值。
2. 书上第二遍强调：“记住，值的类型并不是值的内在本质（inherent），这取决于它被使用的方式，比如，编译器认定一个函数会返回一个整型值，它将产生整数指令来操纵这个值。”

### 7.3 函数的参数

1. 下标引用是间接访问的另一种形式，它可以对指针进行间接访问操作，访问指针指向所在内存位置。
2. 在声明数组参数不用声明数组的长度，因为函数不为数组元素分配内存，函数只会分配内存给指向数组首元素的指针变量。

### 7.5 递归

1. 尾递归：递归调用是函数所执行的最后一项任务，它与普通递归区别在与对内存的占用上（和迭代一样，占用恒定的内存）

例（普通递归）：

int fun( int n ) {

if( n <= 0 )

return 1;

else

return n \* fun( n - 1 );

}

当调用fun(5)

调试信息：

fun (5)

5 \* fun(4)

5 \* (4 \* fun(3))

5 \* (4 \* (3 \* fun(2)))

5 \* (4 \* (3 \* (2 \* fun(1)))

5 \* (4 \* (3 \* (2 \* (1 \* fun(0)))

5 \* (4 \* (3 \* (2 \* (1 \* (1)))

5 \* (4 \* (3 \* (2 \* (1))

5 \* (4 \* (3 \* (2))

5 \* (4 \* (6))

5 \* 24

120

普通递归创建stack累积而后计算收缩，我们并不希望出现这种情况

所以我们将上面函数写成尾递归

int fun(int n， int total = 1) {

if( n == 1 )

return total;

else

return fun(n - 1, n \* total);

}

当调用fun(5)

调试信息：

Fun(4, 5 \* 1)

Fun(3, 4 \* 5)

Fun(2, 3 \* 4 \* 5)

Fun(1, 2 \* 3 \* 4 \* 5)

这里我们引用了一个中间变量total，其实我觉得尾递归和用while控制的循环很像，再把上面这个例子改写一下

int fun(int n) {

int total = 1;

while (n > 0) {

total = n \* total;

n--;

}

return total;

}

更加简洁，而且好理解些。

### 7.6 可变参数列表

1. 黑科技，可以接受不同数目的参数
2. 使用方法：

例：

int fun(int n, ...) {

va\_list var\_test;

va\_start(var\_test, n); //初始化后var\_test指向可变参数部分第一个参数

printf(“%d”, va\_arg(var\_test, int)); //va\_arg()中第二个参数代表下一个参数类型

va\_end(var\_test);

}

1. 原理：

<step 1> va\_list是一个字符指针，先声明一个变量var\_test

<step 2>初始化，让var\_test指向可变参数部分第一个参数

<step 3>获取var\_test指向变量，再将var\_test + 1(必须说明下个变量的类型，因为不用 类型占用字节数不一样)

<step 4>将var\_test置为NULL

在这个里面呢，又涉及另一个知识点，函数的参数是通过栈来存储的，参数表最后

一个元素第一个入栈。

## 编程练习：

### 7.11.2

代码:

#include<stdio.h>

int gcd(int M, int N) {

if (M > 0 && N > 0) {

if (M % N == 0) {

return N;

}

else {

return gcd(N, M % N);

}

}

else {

return 0;

}

}

int main() {

printf("65和26的公约数:%d\n",gcd(65, 26));

printf("26和65的公约数:%d\n",gcd(26, 65));

printf("当M小于零时函数返回值：%d", gcd(-1, 2));

}

结果：

1

总结：gcd(26, 65)步骤比gcd(65, 26)多一步，gcd(26,65)第一步之后转换为gcd(65,26)

### 7.11.3

代码：

#include<stdio.h>

int ascii\_to\_integer(char \*string) {

int value = 0;

while (\*string >= '0' && \*string <= '9') {

value \*= 10;

value += \*string - '0';

string++;

}

if (\*string != '\0') {

value = 0;

}

return value;

}

int main() {

char string[10] = "12345";

printf("%d", ascii\_to\_integer(string));

}

测试数据：”12345”

结果：

2

### 7.11.4

代码：

#include<stdio.h>

#include<stdarg.h>

int max\_list(int max, ...) {

va\_list var\_max;

va\_start(var\_max, max);

int temp;

if (max >= 0) {

temp = va\_arg(var\_max, int);

}

else return 0;

while(temp >= 0) {

if (temp > max) {

max = temp;

}

temp = va\_arg(var\_max, int);

}

va\_end(var\_max);

return max;

}

int main() {

printf("%d", max\_list(0, 10, 6, 8, 3, 5, 7, 12, 29, 59, -1)); //必须以一个负值结尾

}

测试数据：0, 10, 6, 8, 3, 5, 7, 12, 29, 59, -1

结果：

3

### 7.11.5

代码(看的答案)：

#include<stdio.h>

#include<stdarg.h>

void print\_integer(int value) {

}

void print\_float(int value) {

}

void print(char\* format, ...) {

va\_list arg;

char ch;

char \*str;

va\_start(arg, format);

while ((ch = \*format++) != '\0') {

if (ch != '%') {

putchar(ch);

continue;

}

switch(\*format!='\0' ? \*format++ : '\0') {

case 'd':

print\_integer(va\_arg(arg, int));

break;

case 'f':

print\_integer(va\_arg(arg, float));

break;

case 'c':

putchar(va\_arg(arg, int));

break;

case 's':

str = va\_arg(arg, char\*);

while (\*str != '\0') {

putchar(\*str);

str++;

}

break;

}

}

}

int main() {

char value = 'a';

char string[12] = "hello world";

print("%c\n", value);

print("%s", string);

}

结果：

4

总结：这道题是真的厉害，其实可以更甚，一个库函数都不用，写汇编调用显卡打印字符， 不过现在来做可能太费时，可以等到操作系统上完后尝试下。

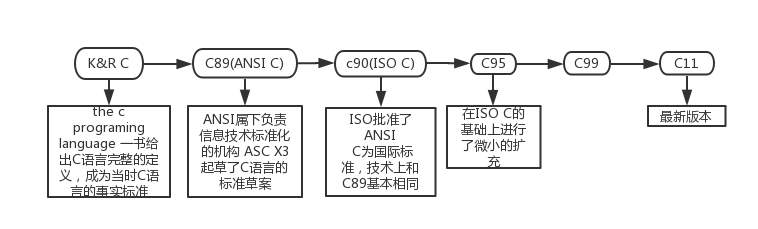
答案中的print()函数有一点我是没考虑到的

形如print(“the max is %d, the min is %d”, max, min);

答案根据print()的第一个参数来控制后边参数的输出，又因为参数的个数是不确定的， 这种情况使用可变参数列表简直完美，虽然简陋，但是思路是没有问题的。

## 问题：

1. ANSI C和 K&R C？

书中多次提起这两种C语言规范，在网上查阅了资料，整理了下C语言标准的发展历史

# .数组

### 8.1 一维数组

1. 几乎在所有使用数组名的表达式中，数组名的值是一个指针常量，也就是数组第一个元素的地址。

有两个例外：

1. 当数组名作为sizeof的操作符，sizeof返回整个数组的长度，而不是指针常量的长度
2. 数组名作为单目操作符&的操作数时，产生的是一个指向数组的指针，形如(\*p)[10]
3. **除了优先级之外，下标引用和间接访问完全相同（**此句在本章出现了三次**）**
4. 作用于指针的下标引用有效性及依赖于该指针当时恰好指向什么内容(不能指向非法的内存空间),也依赖于下标的值(不能越界)。
5. 声明一个数组，编译器给它分配内存空间，然后再创建数组名，数组名是一个常量指向这个空间的起始位置；当声明一个指针变量时，编译器只会为指针变量本身留内存空间，而不会为它分配内存。
6. 对数组的声明也说明了为什么数组形参可以不写它的元素个数，而可以和任意长度数组匹配。因为数组形参传入的只是分配好的内存空间的起始地址。
7. 自动变量在缺省情况下是未初始化的。

### 8.2 多维数组

1. 多维数组作为参数传入函数，有两种形式：

void func( int (\*test)[10] );

void func( int test[][10] );

## 编程练习：

### 8.8.1

代码：

#include<stdio.h>

int main() {

unsigned char char\_value[3][6][4][5] = {

{/\*0\*/

},

{/\*1\*/

{/\*1,0\*/

},

{/\*1,1\*/

{/\*1,1,0\*/

},

{/\*1,1,1\*/

0,'a'

},

},

{/\*1,2\*/

{/\*1,2,0\*/

},

{/\*1,2,1\*/

},

{/\*1,2,2\*/

0, 0, 0,'A'

},

}

},

};

printf("%c\n", char\_value[1][1][1][1]);

printf("%c", char\_value[1][2][2][3]);

}

结果：

1

### 8.8.3

代码：

#include<stdio.h>

#define Row 3

#define False 0

#define True 1

bool identity\_matrix( int (\*matrix)[3] ) {

for ( int i = 0; i < Row; i++ ) {

if ( matrix[i][i] == 0 ) {

return False;

}

}

for ( int i = 0; i < Row; i++) {

for ( int j = 0; j < 3; j++) {

if ( i == j ) {

continue;

} else if ( matrix[i][j] != 0 ) {

return False;

}

}

}

return True;

}

int main() {

int matrix[Row][3] = {

{

1, 0, 0

},

{

0, 1, 0

},

{

0, 0, 1

}

};

if ( identity\_matrix(matrix) ) {

printf("yes");

}

else {

printf("no");

}

}

测试数据：

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

结果：

2

### 8.8.4

代码：

#include <stdio.h>

#include "math.h"

#define False 0

#define True 1

int identity\_matrix( int \*matrix, int size ) {

//检查矩阵是否为正方形

if ( sqrt(size) != size / sqrt(size) ) {

return False;

}

int Row = sqrt(size);

for ( int i = 0; i < Row; i++ ) {

if ( \*(matrix + i \* Row + i) == 0 ) {

return False;

}

}

for ( int i = 0; i < Row; i++) {

for ( int j = 0; j < Row; j++) {

if ( i == j ) {

continue;

} else if ( \*(matrix + i \* Row + j) != 0 ) {

return False;

}

}

}

return True;

}

int main() {

int matrix[3][4] = {

{

1, 0, 0, 0

},

{

0, 1, 0, 0

},

{

0, 0, 1, 0

}

};

float size = 12;

if ( identity\_matrix(\*matrix, size) ) {

printf("yes");

}

else {

printf("no");

}

}

测试数据：

a.

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

size = 12；

结果：2

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |

b.

size = 16；

结果：3

### 8.8.5

代码：

#include<stdio.h>

int main() {

int x = 3, y = 2, z = 4;

int m1[3][2] = {{2, -6},{3, 5},{1, -1}};

int m2[2][4] = {{4, -2, -4, -5}, {-7, -3, 6, 7}};

int r[3][4] = {0};

int sum = 0;

for ( int i = 0; i < x; i++ ) {

for ( int j = 0; j < z; j++) {

for (int k = 0; k < y; k++) {

sum += m1[i][k] \* m2[k][j];

}

r[i][j] = sum;

sum = 0;

}

}

for ( int i = 0; i < x; i++ ) {

for ( int j = 0; j < z; j++) {

printf("%5d", r[i][j]);

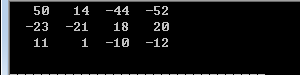
}

printf("\n");

}

}

结果：



总结：我觉得这道题用指针多此一举，给定了x，y，z，直接声明数组更合符常里，书上给的函数原型强行是用指针，给自己添麻烦，如果说是练习目的，那8.8.4那道题更好一点，用指针的确方便一些。

### 8.8.8

代码（没做出来）：

#include <stdlib.h>

#include <stdio.h>

int board[8][8] = {0}; //表示棋盘，初始为0，表示未放置皇后

int num = 0; //解数目

//对于棋盘前row-1行已放置好皇后

//检查在第row行、第column列放置一枚皇后是否可行

bool check(int row,int col) {

if(row == 0) return true;

int i,j;

//纵向只能有一枚皇后

for(i = 0; i <= row - 1; i++) {

if(board[i][col] == 1) return false;

}

//主对角线元素row-col相等

//左上至右下只能有一枚皇后

i = row - 1;

j = i - (row - col);

while(i >= 0 && j >= 0) {

if(board[i][j] == 1) return false;

i--;

j--;

}

//处在副对角线的元素row+col相等

//右上至左下只能有一枚皇后

i = row - 1;

j = row + col - i;

while(i >= 0 && j <= 7) {

if(board[i][j] == 1) return false;

i--;

j++;

}

return true;

}

//当已放置8枚皇后，为可行解时，输出棋盘

void output() {

int i,j;

num++;

printf("answer %d:\n",num);

for(i = 0; i < 8; i++) {

for(j = 0; j < 8; j++) printf("%d ",board[i][j]);

printf("\n");

}

}

//采用递归函数实现八皇后回溯算法

//该函数求解当棋盘前row-1行已放置好皇后，在第row行放置皇后

void solve(int row) {

int j;

//考虑在第row行的各列放置皇后

for (j = 0; j < 8; j++) {

//在其中一列放置皇后

board[row][j] = 1;

//检查在该列放置皇后是否可行

if (check(row, j) == true) {

//若该列可放置皇后，且该列为最后一列，则找到一可行解，输出

if(row == 7) output();

//若该列可放置皇后，则向下一行，继续搜索、求解

else solve(row + 1);

}

//取出该列的皇后，进行回溯，在其他列放置皇后

board[row][j] = 0;

}

}

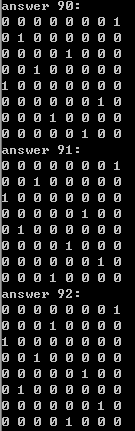
int main() {

solve(0);

return 0;

}

结果：共有92组解，截取最后四组



总结：这道题写了相当久，最开始被书上的图给误导了，以为每次判断对角线是否冲突都需要遍历所有处于主对角线和副对角线上的元素，写出来的程序死循环。不得已只好去网上看看别人的思路，看完发现最开始我的思路没问题，就是回溯这一步要用递归，我用的是一个数组pre\_col[]保存每次成功放置皇后的列数，有点栈的感觉，回溯的时候就恢复到上次成功的下一列，但是涉及的操作太复杂，比如判断是否这行回溯过，又要添加更多的变量保存这些信息，最后把别人的程序理解了，改了改，摒弃了我原来那种。

## 问题：

1. 数组指针和指针数组？

|  |  |  |
| --- | --- | --- |
|  | 数组指针 | 指针数组 |
| 定义 | 指向数组的指针 | 数组成员是指针 |
| 声明方式 | int (\*P)[10] | int \*p[10] |
| 解释 | ()优先级高，所以p是一个指针，指针指向哪里呢？然后执行下标引用，所以p指向某种类型数组。 | []优先级高，先和p结合，结果是一个整型数组，int \*再声明其为一个元素为整型指针的数组 |

再写一点帮助理解：

int (\*p)[10];

\*p; //p是一个指针

int ( )[10]; //指向一个包含10个整型元素的数组

1. 指针的效率？

-----------------------------------------------------2016.11.19-------------------------------------------------------------

# .字符串

### 9.1 字符串基础

1. NUL字节是字符串的终止符，但它本身不是字符串的一部分，所以字符串的长度并不包括NUL字节。

### 9.2 字符串长度

1. 在表达式中使用无符号数将导致不可预料的结果：

例：if ( strlen(x) >= strlen(y) ) 和

if ( strlen(x) - strlen(y) >= 0 )

看起来两句相同，实际第二条语句结果永远为真，因为strlen()返回无符号数，两个无符号数相减还是无符号数，永远大于零。

1. **寻找一种更好的算法比改良一种差劲的算法更有效率，复用已经存在的软件比重新开发一个效率更高。**

### **9.3 不受限制的字符串函数**

1. strcpy( dst, src) 如果参数dst src内存出现重叠，将会出现不可预知的结果

解决方案：使用memmove(dst, src) 此函数先将src拷贝到一个临时位置，这个临时位置不会和源或目标操作数重叠，然后再把它从这个临时位置复制到dst。

1. strcpy和strcat都返回它们第一个参数的拷贝，就是指向目标字符数组的指针。
2. strcmp(a, b) 当a和b相等返回0

### 9.4 长度受限的字符串函数

1. strncpy(dst, src, len),如果src所需空间大于len，那么将只有len个字符复制到dst，它的结果将不会以NUL字节结尾。

2. strncat最多向目标数组赋值len个字节（再加一个NUL字节），它并不关心目标数组空间是否足够

### 9.6 高级字符串查找

1. char \*strtok(char \*src, char const \*sep),

这个函数将src中sep字符串中包含的字符作为分割符，将src分割

sep参数是个字符串，sep是个指针常量。 在第一次调用strtok时向它传递一个指向字符串的指针，然后这个函数被重复调用（此时第一个参数为NULL）,直到它返回NULL为止。

实例：

char line[] = "10.0.20.111";

static char whitespace[] = ".";

char \*token;

for ( token = strtok( line, whitespace);

token != NULL;

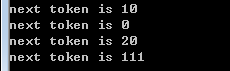
token = strtok( NULL, whitespace ) )

{

printf("next token is %s\n", token);

}

结果：



为什么第二次开始要将NULL作为第一个参数？

char \*strtok(char \*src, char const \*sep)

在网上看了strtok的源码，strtok用一个静态指针olds保存下一次字符串分割的起始位置，函数开头就是if (src == NULL) {src = olds;}

### 9.8 字符操作

1. 直接测试或操纵字符会降低程序的可移植性，比如

if (ch >= ‘A’ && ch < ‘Z’)

这条语句在使用EBCDIC字符集的机器上会失败

应该使用if (isupper(ch)),通用。

## 编程练习：

### 9.14.3

代码：

//my\_strcpy();

#include<stdio.h>

char\* my\_strcpy(char \*buffer, char \*src, int size) {

int length = 0;

while ( length < size && \*(src + length) != '\0') {

\*(buffer + length) = \*(src + length);

length++;

}

if (length == size) {

\*(buffer + size - 1) = '\0';

}

else {

\*(buffer + length) = '\0';

}

return buffer;

}

int main() {

char buffer[5];

char src[10] = "hello";

printf("%s", my\_strcpy(buffer, src, 5));

}

结果：

1

### 9.14.4

代码：

//my\_strcat();

#include<stdio.h>

char\* my\_strcat(char \*dst, char \*src, int size) {

int dst\_index = 0;

while (\*(dst + dst\_index) != '\0') {

dst\_index++;

}

int src\_index = 0;

while (dst\_index < size && \*(src + src\_index) != '\0') {

\*(dst + dst\_index) = \*(src + src\_index);

dst\_index++;

src\_index++;

}

if (dst\_index == size) {

\*(dst + size - 1) = '\0';

}

else {

\*(dst + dst\_index) = '\0';

}

return dst;

}

int main() {

char dst[10] = "hello ";

char src[10] = "world!";

printf("%s", my\_strcat(dst, src, 10));

}

结果：

2

### 9.14.8

代码：

//my\_strnchr();

#include<stdio.h>

int my\_strnchr( char const \*str, int ch, int which ) {

int index = 0;

while ( which > 0 && \*(str + index) != '\0' ) {

if (\*(str + index) == ch) {

which--;

}

index++;

}

if (which > 0) {

printf("have not found");

}

else {

return index - 1;

}

}

int main() {

char str[20] = "hello world!";

printf("%d", my\_strnchr(str, 'l', 2));

}

结果:

3

最开始想返回指针，但是结果不好验证，反正是练习，最后决定返回字符在数组的位置，也就是下标加一。

### 9.14.10

代码：

//判断是否为回文串

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void check( char \*src ) {

int front = 0;

int rear = strlen(src) - 1;

while ( front < rear ) {

if (\*(src + front) != \*(src + rear)) {

break;

}

front++;

rear--;

}

//字符串长度为偶数，匹配时front会比rear大1

if ( front >= rear ) {

printf("yes");

}

else {

printf("no");

}

}

int main() {

char src[20] = "HellOolleh";

for ( int i = 0; src[i] != '\0'; i++ ) {

src[i] = tolower(src[i]);

}

check(src);

}

结果：

4

题目给了个要求：忽略除字母的字符，这点还没想到怎么解决，也许用栈，先记下吧。

### 9.14.11

代码：

//数字符串中the的个数

#include<stdio.h>

#include<string.h>

int scan(char \*line) {

int counter = 0;

char \*olds = line; //指向下一个非the字符

while ( (olds = strstr(olds, "the")) != NULL) {

counter++;

olds += 3;

}

return counter;

}

int main() {

char line[50] = "the old, the young, the lady!";

printf("%d", scan(line));

}

结果：

5

### 9.14.12 && 9.14.13

代码：

#include <stdio.h>

#include <string.h>

#define FALSE 0

#define TRUE 1

char alphabet[26] = {0};

int prepare\_key(char \*key) {

if(strlen(key) > 26) {

return FALSE;

}

char has\_shown\_char[26] = {0};

int len; //记录已经出现字母个数

for (int i = 0; \*key != NULL && i < 26; i++) {

//如果重复就跳过

if (strchr(has\_shown\_char, \*key) == NULL) {

alphabet[i] = \*key;

has\_shown\_char[len++] = \*key++;

continue;

}

else {

i = i - 1;

key++;

}

}

//补全字母表

for (int i = len; i < 26; i++) {

for (int j = 0; j < 26; j++) {

if (strchr(has\_shown\_char, 'A' + j) == NULL) {

alphabet[i] = 'A' + j;

has\_shown\_char[len++] = 'A' + j;

break;

}

}

}

return TRUE;

}

void encrypt(char \*data, char const \*key) {

char encrypted\_data[20] = {0};

int i;

for (i = 0; \*data != '\0'; i++) {

if (\*data == ' ') {

encrypted\_data[i] = ' ';

}

else{

encrypted\_data[i] = alphabet[\*data - 'A'];

}

data++;

}

printf("%s", encrypted\_data);

}

int main() {

char key[27] = "TRAILBLAZERS";

prepare\_key(key);

printf("生成密码表：%s\n", alphabet);

char src\_data[20] = "ATTACK AT DAWN";

printf("元数据：%s\n", src\_data);

printf("加密后数据：");

encrypt(src\_data, key);

}

结果：

1

总结：对黑盒模型理解又进一步，这道题里有一步关键，比较字符是否已经出现，假设以上功能可以由函数A完成，先不管这个函数的实现细节，重要知道它可以完成这个功能就可以了，这样我们就可以使用它了，如

if(该字符未出现过){

do something;

}

然后再通过查询，原来这个函数就是strchr，ok，修改下：

if (strchr(has\_shown\_chars, char) == NULL) {

do something;

}

带来的启示，先把框架完成，设想每个函数的功能，把它们当成一个黑盒，实现的细节可以放到最后。

# .结构和联合

### 10.1 结构基础知识

1. 书上例子

struct SLEF\_REF1 {

int a;

struct SELF\_REF1 b;

int c;

}

这种自引用方式会永无止境地重复下去。

改正：将b声明成结构的指针，编译器在结构的长度确定之前就知道指针的长度，所以合法。

这种在结构内部包括指向结构本身的指针看起来很奇怪，事实上很多高级的数据结构，如链表和树都是用这种方式实现的。

1. 还有一个例子

typedef struct {

int a;

MyType \*b;

char c;

} MyType;

b的声明是错误的，类型名直到声明末尾才定义，所以在结构体内部它尚未定义

改正：

typedef struct Mytype\_Tag{

int a;

MyType \*b;

char c;

}MyType;

定义一个结构标签来声明b；

### 10.2 结构，指针和成员

1. 先声明一个结构体：

typedef struct {

int a;

char b;

} Test;

再声明一个结构体指针；

Test test;

Test \*p = test;

然后再比较一下\*p和p->a的区别

这两个表达式的值p存储的地址都是一样的，看起来这个指针既指向整个结构体又指向结构体第一个元素a，但实际上它们的类型是不同的，p被声明称一个指向结构的指针，所以\*p的结构是整个结构，而不是它的第一个成员。

### 10.3 结构的存储分配

1. 只有当存储成员时需要满足正确的边界对齐要求时，成员之间才可能出现用于填充的额外内存空间。举个例子：

struct ALIGN {  
 char a;

int b;

char c;

};

该结构体在内存中的储存方式如下图所示（该机器int型为4个字节）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a |  |  |  | b | c |  |  |  |

可以看到空间利用率为50%，比较低下

解决方案：改变元素声明顺序，把对边界对齐要求最严格的元素先声明

struct ALIGN {

int b;

char a;

char c;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| b | a | c |  |  |

修改后空间利用率75%

### 10.4 结构体作为函数参数

1. 推荐传入指向结构体的指针，指针比整个结构体小得多，把它直接压到堆栈上效率提高很多。而且如果你需要修改结构体的任意成员，也需要使用指针传值的方式。

### 10.5 位段

1. 和汇编语言的伪指令DW，DB很像，申请内存（以字节为单位），c语言的位段实际储存在一个或多个整型变量中。
2. 位段的可移植性很差。
3. 使用位段的好处：因为有些数据类型存在边界对齐，所以奇数位的数据往往会浪费很多存储空间，这时用位段将数据包装起来，可以节省空间。
4. 位段的操作可以用移位和屏蔽来实现。

### 10.6 联合

1. 联合中成员所引用的位相同，区别在每个成员的类型决定了这些位被如何解释。
2. 如果联合的各个成员具有不同的长度，联合的长度就是它最长成员的长度。
3. 针对2中所诉情况，当储存长度短的成员时，浪费空间。更好的方法是储存指向成员的指针，当决定使用哪个成员时，再分配正确数量的内存来储存它。

# .动态内存分配

### 11.1 为什么使用动态内存分配

1. malloc从内存池中提取一块合适的内存，并向该程序返回一个指向这块内存的指针，这块内存此时并没有以任何方式进行初始化。
2. malloc所分配的是一块连续的内存，而且它分配的内存可能会比你实际需要的内存多一点。
3. 对malloc返回值是否为NULL进行检查很重要，因为内存池是有可能分配完的。

### 11.5 常见的动态内存错误

1. 动态内存分配可能出现的问题
   1. malloc分配内存失败返回NULL没有检查到
   2. 释放并非动态分配的内存
   3. 试图释放动态分配内存的一部分
   4. 一块被释放的动态内存接着使用

## 警示：

1. 当对一个指向动态分配的内存的指针进行了复制，而且这个指针的几份拷贝分布在程序各处，你无法保证你使用其中一个指针时它所指向的内存是不是已经被另一个指针释放。所以在这块内存被释放之前停止对它的使用。

## 编程练习：

### 11.11.3

代码：

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

char\* allocate() {

int len = 1;

char input;

char \*buffer = (char\*)malloc(sizeof(char) \* 1);

while ((input = getchar()) != '\n') {

\*(buffer + len - 1) = input;

len++;

realloc(buffer, len);

}

//在字符串结尾加上'结束标志'

\*(buffer + len) = '\0';

return buffer;

}

int main() {

printf("%s", allocate());

}

思路：每读入一个除换行以外的字符，动态分配的buffer的空间就加1，把读入的字符复制到buffer中。

结果：

1

# .使用结构和指针

### 12.2 单链表

1. 优化单链表的插入函数：

因为c允许我们获得当前对象的地址，所以我们不需要pre指针记住上一个节点位置，我们用一个指针linkp保存当前节点的link字段。代码如下：

while( current != NULL && current->value < value) {

linkp = &current->link;

current = \*linkp;

}

最后跳出循环时，将新节点插在current前面，操作：

new->link = current;

\*linkp = new;

### 12.3 双链表

1. 语句提炼：

例：if (x == 3) {

a = 1;

b = 3;

do something;

}

else {

a = 1;

b = 3;

do other thing;

}

可以发现无论x是否等于3，a = 1;b = 3;都会执行，所以我们把代码精简下：

a = 1;

if (x == 3) {

do something;

}

else {

do other thing;

}

b = 3;

## 编程练习：

12.8.4

代码：

#include <stdio.h>

#include <stdlib.h>

#define FALSE 0

#define TRUE 1

typedef struct NODE {

int value;

struct NODE \*link;

}Node;

//头插法建表

void CreatList(Node \*\*root) {

Node \*p = \*root;

Node \*new\_node;

int new\_value;

while (scanf("%d", &new\_value) != EOF) {

//输入负值作为结束标志

if (new\_value < 0) {

break;

}

new\_node = (Node \*)malloc(sizeof(Node));

new\_node->value = new\_value;

p->link= new\_node;

new\_node->link = NULL;

p = p->link;

}

}

//输出链表

void DispList(Node \*\*root) {

Node \*p = (\*root)->link;

while( p != NULL) {

printf("%d ", p->value);

p = p->link;

}

printf("\n");

}

//反置链表

int ReverseList(Node \*\*root) {

Node \*p = NULL;

Node \*tmp = NULL;

if (root == NULL) {

return FALSE;

}

tmp = (\*root)->link;

while (tmp->link != NULL) {

p = tmp->link;

tmp->link = p->link;

p->link = (\*root)->link;

(\*root)->link = p;

}

return TRUE;

}

int main() {

Node\* root = (Node\*)malloc(sizeof(Node));

root->link = NULL;

CreatList(&root);

printf("原链表：");

DispList(&root);

ReverseList(&root);

printf("反置后链表：");

DispList(&root);

}

结果：

x

-------------------------------------------------------2016.12.8------------------------------------------------------------