# .快速上手

### 1.1 简介

1. gets()函数，从标准输入读取一行文本并将他存储作为参数传递给它的数组，一行输入以一串字符组成，以一个换行符结尾，gets()丢弃换行符，并在改行末尾储存一个NUL字节。

|  |  |
| --- | --- |
| 格式 | 含义 |
| %o | 以八进制形式打印一个整型值 |
| %x | 以十六进制形式打印一个整型值 |
| %g | 打印一个浮点数（不输出无意义的0） |

2.

1. while((ch = getchar() != EOF && ch != ‘\n’)); //清空缓冲池
2. strncpy(char \*dest, const char \*src, int n);

## 编程练习：

### 1.8.3

代码：

#include<stdio.h>

int main() {

char checksum = -1;

char a = getchar();

while(a != '\n') {

int temp = checksum;

checksum = checksum + a;

printf("%c", a);

a = getchar();

}

checksum = checksum + a;

printf("\n");

printf("%d", checksum);

}

运行结果：

1.8.3

### 1.8.4

代码：

#include<stdio.h>

#define MAX\_ROW 100

int main() {

int lenth[MAX\_ROW];

char input[MAX\_ROW][1000];

int r = 0, c = 0;

int counter = 0; //记录输入行行数

while (r < MAX\_ROW) {

//若无输入表示结束

input[r][c] = getchar();

if (input[r][c] == '\n') {

break;

} else {

lenth[r]++;

}

//换行符也计算在字符串长度里

while (input[r][c] != '\n') {

c++;

input[r][c] = getchar();

lenth[r]++;

}

c = 0;

r++;

counter++;

}

//排序

int max\_row = 0;

int max\_row\_index = 0;

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

if (lenth[i] > max\_row) {

max\_row = lenth[i];

max\_row\_index = i;

}

}

//输出

printf("the longest string is:");

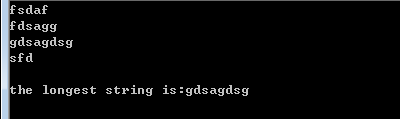
for (int i=0; i<lenth[max\_row\_index]; i++) {

printf("%c",input[max\_row\_index][i]);

}

}

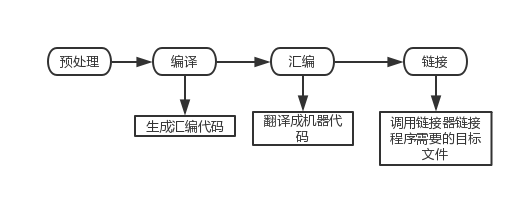
运行结果：



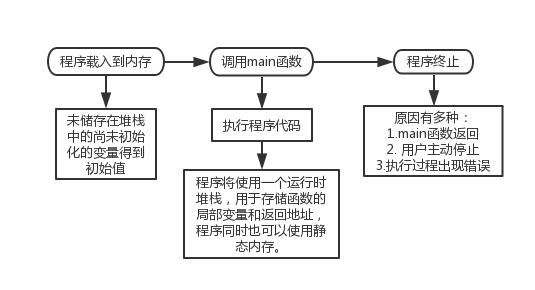
# .基本概念

### 2.1 环境

1. C语言的编译过程：



1. 程序的执行过程：



## 编程练习：

### 2.8.2

代码：

#include<stdio.h>

#define MAX\_NUM 100

//简化成括号匹配问题

char stack[100]; //初始化一个栈

int top = -1;

int main() {

char ch;

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

if (ch == '(') {

top = top + 1;

stack[top] = ch;

}

if (ch == ')') {

if (stack[top] == '(') {

stack[top] = 0;

top = top - 1;

}

}

}

if (top == -1) {

printf("括号是匹配的!\n");

} else {

printf("括号不匹配!\n");

}

}

输出结果：

1. 测试数据:((()))()

测试1

1. 测试数据:()((()()

测试2

1. 测试数据:)))(((

测试3

# .数据

### 3.1 基本数据类型

1. long int至少和int一样长，int至少应该和short int型一样长
2. 枚举类型就是它的值为符号常量而不是字面量的类型，如果某个符号名未显式指定一个值，那么它的值就比前面一个符号名的值大1
3. ANSI标准规定：所有浮点类型至少能够容纳从10^-37到10^37之间的任何值
4. 字符串为空依然存在作为终止符的NUL字节

### 3.2 基本声明

1. char \*message = “Hello world!”;

Char \*message; message = “Hello world!”;

这两条语句是等价的，在书中作者提到“***程序在使用字符串常量会生成一个指向字符的常量指针”，“字符串常量的直接值是一个指针，而不是是这些字符本身***”，这两句就很好地解释了以上两句语句为何等价。

### 3.5 作用域

1. 如果内层代码块中有一个标识符的名字和外层代码块的一个标识符同名，那么内层的标识符将隐藏外层的标识符

### 3.6 链接属性

|  |  |
| --- | --- |
| 链接属性 | 标识符 |
| external | 无论声明几次，位于几个源文件都表示同一个实体 |
| internal | 位于不同源文件的多个声明分属与不同的实体 |
| none | 多个声明被当作独立不同的实体 |

1.

2. extern和static：修改标识符的链接属性，static可以使标识符的链接属性变成internal，需要注意的一点是当***extern用于一个标识符的第二次或以后的声明，extern不会修改标识符第一次声明所指定的链接属性***。

### 3.7 储存类型

1.

|  |  |  |  |
| --- | --- | --- | --- |
| 变量声明位置 | 储存位置 | 储存类型 | 生存期 |
| 代码块外 | 普通内存 | static | 在程序运行之前创建，在程序整个执行期始终存在 |
| 代码块内部 | 运行时堆栈 | auto | 程序执行离开改代码块便自行销毁 |
| 同auto | 寄存器 | register | 同auto，但是它需要一些额外工作。 |

## 问题：

1. **指针常量和常量指针？**

***常量指针：***

声明方式：

int a = 3;

const int \*p = &a ，

指向常量的指针，不能通过常量指针去修改它指向的对象的值，如\*p = 6 ，但是可以修改常量指针的值

***指针常量：***

声明方式：

int a;

int \*const b = &a;

指针常量的值是指针，这个值因为是常量，在声明时需要赋初值，赋了初值后指针的值就不能改变，但是它所指向的对象的值是可以修改的

1. **指针和数组等价？**

指针和数组并不相同，也不能互换，只是说指针可以很方便地访问数组或者模拟数组。等价的基础在于一个关键定义：“ 一个T数组类型的对象如果出现在表达式中会退化为一个指向数组第一个元素的指针，指针的类型是指向T的指针。”一旦数组出现在表达式中，程序就会隐式地生成一个指向数组第一个元素的指针，但是当数组作为sizeof或&操作符的操作数，或者作为字符数组的字符串初始值的时候例外。

1. **怎么输出常量？常量分配空间？整型常量可以做加减吗？**

C语言输出常量和变量是一样的。

字符串常量分配在常量区，基本类型常量会在编译成立即数，占的是代码段的内存。

整型常量可以做加减，但是不能赋值。

1. **int a = 1;和int a; a = 1;有何不同？**

第一种在编译时就在栈上分配了空间，第二种在编译时是一个随机数，在程序运行时在栈分配内存空间，并把1放入内存

## 课后习题：

### 3.13.23

**我的答案**：

char b = 2; //全局变量默认存储类型为static

extern int a = 1;

void x() {

int c = 3;

static float d = 4;

}

void y() {

int a;

}

**在网上查到的答案**：

static char b = 2;

void y(void) {

}

int a = 1;

void x(void) {

int c = 3;

static float d = 4;

}

**最开始我有点不理解一个链接属性为external的标志符怎样让一个函数访问却不允许另一个函数访问，后来我想了个我看来可行的方案，先把a声明在代码外，让它存储类型为static，缺省链接属性为external，再在y中新声明一个a，同名的内层的标识符隐藏外层的标识符。后来查了答案，答案是把a的声明放在y函数后，但是不知道是我用的dev c++在编译时太智能了还是怎么的，y函数其实还是可以访问a。**

**------------------------------------------------分割线(2016.10.29)------------------------------------------------------**

# **.语句**

### 4.8 switch语句

1.

// 其中expression的值必须是整型值。

switch(expression)

Statement

1. Switch语句的执行过程中，执行流将贯穿各个case标签，所以我们用break来划分它，而且有一点要注意，在switch语句中，continue并不起作用。

### 4.9 goto语句

1. goto是一种危险的语句，尽量避免使用它，但是有一种情况：由于break语句只能跳出最内层循环，要想从最深层处循环跳出来就只有用goto语句。

## 编程练习：

### 4.14.1

代码：

#include<stdio.h>

int count = 0; //计数guess了几次

float sqrt(int n) {

float guess = 1;

while (guess != ((guess + n / guess) / 2)) {

guess = (guess + n / guess) / 2;

count++;

}

return guess;

}

int main() {

int n;

printf("请输入n的值\n");

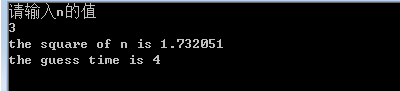
scanf("%d", &n);

printf("the square of n is %f\n", sqrt(n));

printf("the guess time is %d\n", count);

}

结果：



### 4.14.2

代码：

#include<stdio.h>

bool check(int n) {

int count = 0;

for (int i = 2; i <= n; i++) {

if ((n % i) == 0) {

count++;

}

}

if (count > 2) {

return false;

}

else {

return true;

}

}

int main() {

for (int i = 2; i <= 100 ; i++) {

if(check(i)) {

printf("%d ", i);

}

}

}

结果：

modify

### 4.14.4

代码：

#include<stdio.h>

#define MAX\_NUM 100

void copy\_n(char dst[], char src[], int n) {

int src\_lenth = 0;

int count = 0;

//求出源字符串长度

while (src[count] != '\0') {

src\_lenth++;

count++;

}

// 源字符串大于n， dst数组将不是NUL结尾

if (n <= src\_lenth) {

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

dst[i] = src[i];

}

}

//源字符串小于n, dst数组用NUL填满至下标n - 1所指元素

else {

int remain\_lenth = n - src\_lenth;

for (int i = 0; i < src\_lenth; i++) {

dst[i] = src[i];

}

for (int i = src\_lenth; i < remain\_lenth; i++) {

dst[i] = '\0';

}

}

}

int main() {

char src[MAX\_NUM];

char dst[MAX\_NUM];

int n;

printf("请输入字符串:\n");

scanf("%s", src);

printf("请输入n的值:\n");

scanf("%d", &n);

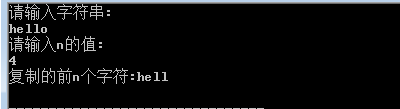
copy\_n(dst, src ,n);

printf("复制的前n个字符:%s\n", dst);

}

结果：

1. src长度大于等于n:





1. src长度小于n：



### 4.14.5

代码(自己没做出来，看了答案)：

#include<stdio.h>

#include<string.h>

int main() {

char input[129];

char str\_prev[129];

bool printed = false;

if (gets(str\_prev) != NULL) {

while (gets(input) != NULL) {

if (strcmp(str\_prev, input) != 0) {

printed = false;

strcpy(str\_prev, input);

}

else if (printed == false) {

printed = true;

printf("the repeat str is:%s\n", str\_prev);

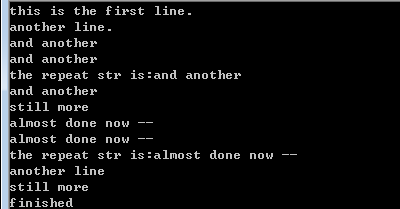
}

}

}

}

结果：



总结：

说实话，这个答案给的并不标准，按常理应该是全部输入行键入后再输出重复行，这种差强人意，勉强达到要求。里面有一步很关键，while里面那个判断句if (strcmp(str\_prev, input) != 0)，else if (printed == false)。进入else if里包含了一个隐含条件：现在输入的这句和上一句是一样的。所以出现多句重复两遍以上的字符串也只会输出一遍。

### 4.14.7

代码：

#include<stdio.h>

void deblank(char string[]) {

int flag[100] = {0}; //标记字符串数组所有元素，string[i]为空格，则flag[i]为1

//先判断第一个数是否为空格

if (string[0] != '\0' && string[0] == ' ') {

flag[0] = 1;

}

int i = 1;

while (string[i] != '\0') {

if (string[i] == ' ') {

flag[i] = 1;

}

//如果前后两个元素均为空格，则数组从下标i + 1处前移，覆盖掉string[i]

if (flag[i] == 1 && flag[i - 1] == 1) {

int count = i;

while (string[count + 1] != '\0') {

string[count] = string[count + 1];

count++;

}

string[count] = '\0';

flag[i] = 0;

}

else {

i++;

}

}

}

int main() {

char string[100];

int i = 0;

printf("请输入字符串：");

while ((string[i] = getchar()) != '\n') {

i++;

}

string[i] = '\0';

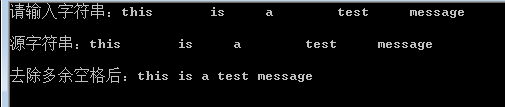
printf("\n源字符串：%s\n", string);

deblank(string);

printf("\n去除多余空格后：%s\n", string);

}

结果：



# .操作符和表达式

### 5.1 操作符

1. 右移操作符：有两种移位方式，第一种逻辑移位：左边移入的位用0填充；第二种算术移位，由原先该值符号决定，符号位为1则移入的位均为1，反之为0
2. 一个程序如果使用了算术右移，那它是不可移植的。还有一种情况，如a << -5，这种移位是有编译器决定的，也是不可移植
3. **位操作符：**这一点想详细说说。

**几个应用**：

a. 在王爽的汇编语言上看到的：

一个字母的大小写二进制表示只有第5位不同（八位二进制排列规则：b7 b6 b5 ... b0），大写是0，小写是1

比如要将一个字母转换成大写(事先并不知道它是大写还是小写)

ch = ch & 0xDF ; //和11011111做与运算

同理全部转换成小写：

ch = ch | 0x20; //和01000000做或运算

b.还有一个：

不使用第三个变量将a和b的值交换

利用异或：

a = a ^ b;

b = b ^ a;

a = a ^ b;

原理也很简单，异或有个性质，对于任何数x，都有x ^ x = 0,x ^ 0 = x

c.再写一个：

1-1000放在含有1001个元素的数组中，只有唯一的一个元素值重复，其它均只出现  
一次。每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空

间，能否设计一个算法实现？

答：把所有数异或，设1^2^...^1000（序列中不包含n）的结果为T，则1^2^...^1000（序列中包含n）的结果就是T^n。T^(T^n)=n。

此题的一个变式（google面试题）：

一个数组存放若干整数，一个数出现奇数次，其余数均出现偶数次，找出这个出现奇数次的数？

答：全部异或，出现偶数次的数全部为零，假如那个出现奇数次的数为x，x ^ 0 = x,得解。

1. 关系操作符返回的结果是整型值，所以可以赋给整型变量
2. **逻辑运算符**：采用“短路求值”，例：&&先对左操作数求值，如果左操作数是假，那么右操作数不再求值

### 5.2 布尔值

1. C并不具备显式的布尔类型，所以用整数来代替，规则：零是假，任何非零值皆为真。

## 问题：

（待解决）

比如书上所讲value & 1 << bit\_number; 测试指定位，如果该位已经被设置为1，则表达式的结果为非零值，这是对一个位操纵的情况，但是我试了下同时对两位及以上的位进行位操作，结果却很奇怪**，**例如：

X = 1010; x = 101;

|  |  |
| --- | --- |
| X & 0111 | false |
| X & 1111 | true |
| X & 1110 | true |
| X & 0110 | false |
| X & 1010 | true |
| X & 0101 | false |

|  |  |
| --- | --- |
| X & 011 | true |
| X & 101 | true |
| X & 110 | true |

真的很奇怪，完全没规律，查也查不到。

## 编程练习：

### 5.9.1

代码：

#include<stdio.h>

int main() {

char ch;

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

if (ch <= 90 && ch >= 65) {

ch = ch | 0x20;

}

printf("%c", ch);

}

}

结果：

1

### 5.9.2

代码：

#include<stdio.h>

void lower\_case\_encrypt(int ch) {

ch = ch + 13 - 96;

if (ch > 26) {

ch = ch % 26;

}

printf("%c", ch + 96);

}

void upper\_case\_encrypt(int ch) {

ch = ch + 13 - 64;

if (ch > 26) {

ch = ch % 26;

}

printf("%c", ch + 64);

}

int main() {

int ch;

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

if(ch >= 65 && ch <= 90) {

upper\_case\_encrypt(ch);

}

else if(ch >= 97 && ch <= 122) {

lower\_case\_encrypt(ch);

}

else {

printf("%c", ch);

}

}

}

结果：

2

### 5.9.3

代码（不会做，看的答案）：

#include<stdio.h>

unsigned reverse(unsigned int value) {

unsigned int answer;

unsigned int i;

answer = 0;

for(i = 1; i != 0; i <<= 1) {

/\*

\*\*将旧的answer左移一位，为下一位留下空间

\*\*如果value的最后一位是1，answer就与1进行or操作

\*\*然后将value右移至下一位

\*/

answer <<= 1;

if (value & 1) {

answer |= 1;

}

value >>= 1;

}

return answer;

}

int main() {

unsigned value = 25;

printf("%u", reverse(value)); //输出无符号数%u

}

结果：

1

### 5.9.4

代码：

#include<stdio.h>

//将指定位置为1

void set\_bit(char bit\_array[],

unsigned bit\_number)

{

if (bit\_number >= 0 && bit\_number <= 7) {

\*(bit\_array + 7 - bit\_number) = '1';

}

else {

printf("请输入有效位\n");

}

}

//将指定位置清零

void clear\_bit(char bit\_array[],

unsigned bit\_number)

{

if (bit\_number >= 0 && bit\_number <= 7) {

\*(bit\_array + 7 - bit\_number) = '0';

}

else {

printf("请输入有效位\n");

}

}

//如果value的值是0，将指定位清0，反之设置为1

void assign\_bit(char bit\_array[],

unsigned bit\_number, int value)

{

if (bit\_number >= 0 && bit\_number <= 7) {

if (value) {

\*(bit\_array + 7 - bit\_number) = '1';

}

else {

\*(bit\_array + 7 - bit\_number) = '0';

}

}

else {

printf("请输入有效位\n");

}

}

int test\_bit(char bit\_array[], unsigned bit\_number) {

if (bit\_number >= 0 && bit\_number <= 7) {

if (\*(bit\_array + 7 - bit\_number) == '1') {

return 1;

}

else {

return 0;

}

}

else {

printf("请输入有效位\n");

}

}

int main() {

char bit\_array[9] = "10110101";

printf("原位数组 ：%s\n\n", bit\_array);

set\_bit(bit\_array, 1);

printf("第一位设置为1 ：%s\n\n", bit\_array);

clear\_bit(bit\_array, 5);

printf("第五位设置清0 ：%s\n\n", bit\_array);

int value = 1;

assign\_bit(bit\_array, 3, value);

printf("当value为1时改变第3位：%s\n\n", bit\_array);

if(test\_bit(bit\_array, 4)) {

printf("第四位不是0\n\n");

}

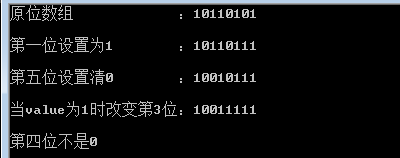
else {

printf("第四位是零\n\n");

}

}

结果：



### 5.9.5

代码（创建掩码那步看的答案）：

#include<stdio.h>

int store\_bit\_field(int original\_value,

int value\_to\_store,

unsigned starting\_bit, unsigned ending\_bit)

{

//创建掩码

unsigned mask = ~0;

mask >>= 32 - ( starting\_bit - ending\_bit + 1 );

mask <<= ending\_bit;

original\_value = original\_value & ~mask;

value\_to\_store = value\_to\_store << ending\_bit;

value\_to\_store = value\_to\_store & mask;

value\_to\_store = value\_to\_store | original\_value;

return value\_to\_store;

}

int main() {

printf("%x", store\_bit\_field(0xffff, 0x123, 13, 9));

}

结果：

3

**------------------------------------------------分割线(2016.11.4)-------------------------------------------------------**

# .指针

### 6.1 内存和地址

1. 每个字一般由两个或者四个字组成
2. 尽管一个字包括了四个字节，但是它的地址是它最左边的字节位置还是最右边字节位置，不同机器有不同规定。
3. 在要求边界对齐的机器上，整型值存储的起始位置只能是某些特定的字节，通常是2和4的倍数

### 6.2 值和类型

1. 不能简单地通过检查一个值的位来判断它的类型。

### 6.3 指针变量的内容

1. 一个变量的值就是分配给这个变量的内存位置所储存的值。即使是指针变量也不例外

### 6.5 未初始化和非法的指针

1.

int \*a;

...

\*a = 12;

这段代码看起来没什么问题，实际上我们声明了指针a，却没有对它进行初始化，所以我们没办法知道12储存在哪里，可能出现把不该修改的内存地址内容修改了，造成严重的后果。

### 6.6 NULL指针

1. 用\*对指针解引用可以获得它所指向的值，但是NULL指针并未指向任何东西，对一个NULL指针解引用是非法的。
2. **接上。**如果对NULL指针间接访问，它的结果由编译器决定，有些编译器会让它访问内存地址零，那里并未储存任何变量。我的编译器是直接返回错误。

### 6.9 指针常量

1.

假设变量a储存在位置100

\*100 = 25；

这条语句看起来是将25赋给a，但实际这是非法的，因为字面值100的类型是整型

所以要完成以上目的需要将语句改为：

(int \*)100 = 25;

但是这种技巧一般是不会使用到的，因为你通常不知到编译器会把特定的变量放在上面位置。除非你需要访问某些特定的地址，来访问硬件本身。

### 6.11 指针表达式

char ch = ‘a’;

char \*cp = ch;

1. 书上说&ch不能作为左值是因为表达式&ch求值时，结果应该保存在某个地方，但你不知道是什么位置，但我觉得，**&ch表达式返回的是一个int型的常量，常量是不能改变的**，所以不能做左值。
2. 书上有些啰嗦，总结下就是：指针变量cp的内容是ch的地址，cp的地址和ch地址并不一样。
3. 间接操作符是少数几个器结果为左值的操作符之一。
4. 前缀++和后缀++在对指针变量操作时，结果是不一样的，如，\*++cp，因自增自减的优先级大于间接访问，这个表达式的结果是ch所在地址的的下一个地址的内容，\*cp++表达式的结果却还是ch所在地址的内容，也就是’a’，cp地址却已经指向了ch地址的下一个地址

### 6.13 指针运算

1. 让指针指向数组最后一个元素后面那个元素是合法的，但是对这个指针间接访问有可能失败
2. 两个指针变量做减法运算(两个指针指向同一个数组的元素)，结果是两个指针在内存中的距离

3. 越界指针和指向未知值的指针是两个常见错误的根源

## 问题：

1. 书上给出了一个很大整数107823331，它既可以解释为整数，也可以被解释成浮点数。但是我有点疑问，为什么107823331会被转换成3.14，我在dev c++上试了下，结果是107823331.000000。

答：主要是浮点数的二进制表示有个规范。

1. 为什么i++是右值，++i是左值？

答：++i是将i加1然后返回i，i是变量，所以可以做左值；i++是将i的值赋给临时变量，再将i的值加1，最后返回临时变量；

## 编程练习

### 6.18.1

代码：

#include<stdio.h>

#define True 1

#define False 0

//字符串长度

int length(char const \*string) {

int len = 0;

while (\*(string + len) != '\0') {

len++;

}

return len;

}

//接收起始删除位置

int start;

//判断模式串j是否为目标串的子串

int index(char \*str, char const \*substr,

int length\_str, int length\_substr)

{

int i = 0, j = 0;

while (i <= length\_str &&

j < length\_substr)

{

if (\*(str + i) == \*(substr + j)) {

i++;

j++;

}

else {

i = i - j + 1;

j = 0;

}

}

if (j == length\_substr) {

start = i - length\_substr;

return True;

}

else return False;

}

int del\_substr(char \*str, char const \*substr) {

int length\_str = length(str);

int length\_substr = length(substr);

if (index(str, substr, length\_str, length\_substr) != 0) {

while (start <= length\_str) {

\*(str + start) = \*(str + start + length\_substr);

start++;

}

}

else {

return False;

}

}

int main() {

char str[100] = "ABCDEFG";

char substr[100] = "ABC";

printf("str : %s\n", str);

printf("substr: %s\n\n", substr);

if (del\_substr(str, substr)) {

printf("del\_substr() has been excuted\n\nstr: %s", str);

}

else {

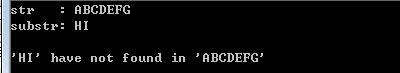
printf("'%s' have not found in '%s'", substr, str);

}

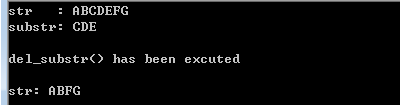
}

结果：

a. substr不是str的子串



b. Substr是str的子串



总结：这道题费了我挺长时间，主要是里面有个字符串匹配问题，又重新翻看了数据结构的书才想起来。最开始我写的时候有段代码有点问题：

下面是index()函数中的一小段代码

if (j == length\_substr) {

flag = 1;

return i - length\_substr;

}

else return False;

这是del\_substr()中的一个判断：

if ((i = index(str, substr, length\_str, length\_substr))

!= 0)

index()函数返回值是int，如果字符串是匹配的，我想让它返回两个值，True和删除的起始位置，return却只能返回一个值，所以我只好想了一个折中的方法，因为非0就是真。

当不匹配的时候，返回假；匹配的时候返回删除的起始位置，但是这里也有点问题啊，比如源字符串是’ABCDEFG’，substr是’ABC’，虽然是匹配的但是返回的却是0，这种方法不可取。

想了想除了设置一个全局变量来接收，好像别无他法了。

修改：在index函数前面写一个全局变量start

然后把index()函数里原句改成：

if (j == length\_substr) {

start = i - length\_substr;

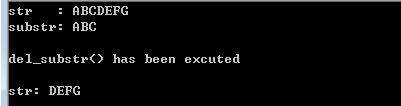
return True;

}

del\_substr()中判断改成：

if (index(str, substr, length\_str, length\_substr) != 0)

修改后测试：



哈哈，好像还简洁了些

### 6.18.3

代码：

#include<stdio.h>

void reverse\_string(char \*string) {

int length = 0;

while (\*(string + length) != '\0') {

length++;

}

for (int i = 0; i < length / 2; i++) {

char temp = \*(string + i);

\*(string + i) = \*(string + length - i - 1);

\*(string + length - i - 1) = temp;

printf("步骤%d：%s\n", i + 1, string);

}

}

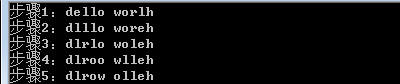
int main() {

char string[12] = "hello world";

reverse\_string(string);

}

结果：



### 6.18.4 & 6.18.6

代码：

#include<stdio.h>

#define MaxNum 100000

void Eratoshenes(int\* flag) {

//将flag数组全部置为1

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

flag[i] = 1;

}

flag[0] = flag[1] = 0;

for (int i = 2; i <= MaxNum; i++) {

if (flag[i] == 1) {

for (int j = 2 \* i; j <= MaxNum; j += i) {

flag[j] = 0;

}

}

}

}

int main() {

int flag[MaxNum];

Eratoshenes(flag);

int start = 0;

int end = 1000;

int total = 0;

while (end <= MaxNum) {

int count = 0;

while (start++ <= end) {

if (flag[start - 1] == 1) {

count++;

total++;

}

}

start -= 1002;

printf("%6d到%6d质数个数：%d\n", start, end, count);

start += 1001;

end += 1000;

}

printf("\n总共:\n%6d到%6d质数个数：%d\n", 0, MaxNum, total);

}

结果：我的机器分配1,000,000个数的数组就报内存错误，所以我就只分配了293,000个数（我机器能分配的最大值），结果太长，做了个图表方便看，里面的规律我是没看出来，最开始0-1000素数最多，到后面就趋于稳定了，也可能是我打印的数不够多，没能很好地看出规律

**-----------------------------------------------分割线(2016.11.11)-------------------------------------------------------**