# 目录

# 第 1 章
# 字符串

## 1.1 ACM 题输入处理

【注：此章节均为补充内容。主要来源于 https://blog.csdn.net/jeffscott/article/details/107969136】

【注：删除字符串】

```
string str1="God bless you!";
string str2="Father be with you!";
str1.erase(3);//删除[3]及以后的字符，并返回新的字符串
str2.erase(3,5);//删除从 [3] 开始的后 5 个字符，并返回新字符串
// God
// Fate with you!
```

【注：追加字符 (串)】

```
string str = "God bless you!";
str.push_back('&');//在str的末尾添加字符'&'，注意只能是字符，不能是字符串如"&"
str.append("Father be with you!");//在 str 末尾添加字符串
// God bless you!&
// God bless you!&Father be with you!
```

【注：插入字符串】

```
string str = "God bless you!";
str.insert(0,"Father&");//从位置0开始添加字符串
str.insert(0,"Maybe",3);//从位置0开始插入字符串"Maybe" 的前三个即 "May"
str.insert(0,"You Wish",4,4);//从位置0开始插入子串"You Wish"从[4]开始的后四个字符，即"Wish"
// Father&God bless you!
// MayFather&God bless you!
// WishMayFather&God bless you!
```

【注：替换字符串】

```
string str = "God bless you!";
str.replace(0,3,"Father");//把str从[0]开始的3个字符，即God替换为Father
str.replace(0,6,"God be with you!",3);//把str从[0]开始的6个字符,即Father替换God
// Father bless you!
// God bless you!
```

【注：获取字符串子串】

```
string str = "God bless you!";
cout<<str.substr(4)<<endl;//返回从[4]开始的后所有字符，即bless you!
cout<<str.substr(4,5)<<endl;//返回[4]以后的5个字符，即bless
```

【注：查找子串】

```
string str = "God bless you!";
cout<<str.find("God")<<endl; // 返回God在str中的起始位置，即0
cout<<str.find("God",3)<<endl; // 在 str[3]~str[n−1] 范围内查找并返回字符串 God 在 str 的位置，即
    npos=18446744073709551615
cout<<string::npos<<endl; // 18446744073709551615
cout<<str.rfind("God",3)<<endl; // 在 str[0]~str[3] 范围内查找并返回字符串 God 在 str 的位置，即0
```

【注：去除一个字符串中的所有空格】

```
/*返回去除空格的字符串*/
string rmSpace(string s){
    while(s.find(" ")!=s.npos)
        s.replace(s.find(" "),1,"");
    return s;
}
```

【注：将一个字符串统一大小写】

```
string str;
getline(cin,str);
transform(str.begin(),str.end(),str.begin(),::toupper);//统一转换为大写
transform(str.begin(),str.end(),str.begin(),::tolower);//统一转换为小写
```

【注：字符串分割】

```
#include<iostream>
#include<string>
#include<vector>

vector<string> split(const string &str, conststring &pattern){
    vector<string> res;
    if(str == "") return res;
    //在字符串末尾也加入分隔符，方便截取最后一段
    string strs = str + pattern;
    size_t pos = strs.find(pattern);
    while(pos != strs.npos){
        string temp = strs.substr(0, pos);
        res.push_back(temp);
        //去掉已分割的字符串,在剩下的字符串中进行分割
        strs = strs.substr(pos+1, strs.size());
        pos = strs.find(pattern);
    }
    return res;
}
```

【注：字符串类型转换】

```
#include<iostream>
#include<string>
using namespace std;

int main(){
    //int --> string
    int i = 5;
    string s = to_string(i);
    cout << s << endl;
    //double --> string
    double d = 3.14;
    cout << to_string(d) << endl;
    //long --> string
    long l = 123234567;
    cout << to_string(l) << endl;
```

```
    //char --> string
    char c = 'a';
    cout << to_string(c) << endl; //自动转换成int类型的参数
    //char --> string
    string cStr; cStr += c;
    cout << cStr << endl;

    s = "123.257";
    //string --> int;
    cout << stoi(s) << endl;
    //string --> long
    cout << stol(s) << endl;
    //string --> float
    cout << stof(s) << endl;
    //string --> doubel
    cout << stod(s) << endl;

  return 0;
}
```

【注：string 转 char*】

```
string str;
char *c = str.c_str();
```

【注：判断字符的类型】

```
cout << isdigit('8') << endl; // 判断是否是数字
cout << isalnum('g') << endl; // 判断是否是字母或数字
cout << isspace(' ') << endl; // 判断是否是空格
cout << islower('a') << endl; // 判断是否是小写
cout << isupper('A') << endl; // 判断是否是大写
cout << isalpha('y') << endl; // 判断是否是字母
cout << tolower('G') << endl; // 将大写字母转为小写字母
```

【注：[1,2,3,4]】

```
#include<sstream>
// ...
int main(){
```

```
    string str;
    getline(cin, str);
    str = str.substr(1, str.size() - 2) + ",";
    string tmp;
    stringstream ss(str);

    while( getline(ss, tmp, ',') ) cout << tmp;
    return 0;
}
// [1,2,3,4]
// 1234
```

## 1.2 字符串 API

面试中经常会出现，现场编写 strcpy，strlen，strstr，atoi 等库函数的题目。这类题目看起来简单，实则难度很大，区分都很高，很容易考察出你的编程功底，是面试官的最爱。

### 1.2.1 strlen

#### 描述

实现 strlen，获取字符串长度，函数原型如下：

```
size_t strlen(const char *str);
```

#### 分析

#### 代码

```
size_t strlen(const char *str) {
    const char *s;
    for (s = str; *s; ++s) {}
    return(s - str);
}
```

【注：1. const int*p=&a（在 * 前面）；当把 const 放最前面的时候，它修饰的就是 *p，*p 表示的是指针变量 p 所指向的内存单元里面的内容，此时这个内容不可变。
2. int*const p=&a；此时 const 修饰的是 p，所以 p 中存放的内存单元的地址不可变，而内存单元中的内容可变。（变成一个引用）
3. const int*const p=&a；此时 *p 和 p 都被修饰了，那么 p 中存放的内存单元的地址和内存单元中的内容都不可变。】

### 1.2.2 strcpy

#### 描述

实现 strcpy，字符串拷贝函数，函数原型如下：

```
char* strcpy(char *to, const char *from);
```

#### 分析

#### 代码

```
char* strcpy(char *to, const char *from) {
    assert(to != NULL && from != NULL);
    char *p = to;
    while ((*p++ = *from++) != '\0');
    return to;
}
```

### 1.2.3 strstr

#### 描述

实现 strstr，子串查找函数【注：在 haystack 中找 needle】，函数原型如下：

```
char * strstr(const char *haystack, const char *needle);
```

#### 分析

暴力算法的复杂度是 $O(m*n)$，代码如下。其他算法见第 §??节“子串查找”。

#### 代码

```
char *strstr(const char *haystack, const char *needle) {
    // if needle is empty return the full string
    // if (!*needle) return (char*) haystack;

    const char *p1;
    const char *p2;
    const char *p1_advance = haystack;
    for (p2 = &needle[1]; *p2; ++p2) {
        p1_advance++;   // advance p1_advance M-1(strlen(needle)-1) times
    }

    for (p1 = haystack; *p1_advance; p1_advance++) {
        char *p1_old = (char*) p1;
        p2 = needle;
        while (*p1 && *p2 && *p1 == *p2) {
            p1++;
            p2++;
```

```
            }
            if (!*p2) return p1_old;

            p1 = p1_old + 1;
        }
        return NULL;
    }
```

```
int strStr(string haystack, string needle) {
    if(needle.empty()) return 0;
    if(haystack.empty() || haystack.length() < needle.length()) return -1;
    const char *p1;
    const char *p2;
    const char *p1_advance = &haystack[needle.length() - 1];
    for (p1 = &haystack[0]; *p1_advance; p1_advance++) {
        char *p1_old = (char*) p1;
        p2 = &needle[0];
        while (*p1 && *p2 && *p1 == *p2) {
            p1++;
            p2++;
        }
        if (!*p2) return p1_old - &haystack[0];
        p1 = p1_old + 1;
    }
    return -1;
}
```

#### 相关题目

与本题相同的题目：

- LeetCode Implement strStr(), http://leetcode.com/oldoj#question_28

与本题相似的题目：

- 无

### 1.2.4 atoi

#### 描述

实现 `atoi`，将一个字符串转化为整数，函数原型如下：

```
int atoi(const char *str);
```

### 分析

注意，这题是故意给很少的信息，让你来考虑所有可能的输入。

来看一下 `atoi` 的官方文档 (http://www.cplusplus.com/reference/cstdlib/atoi/)，看看它有什么特性：

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, `INT_MAX (2147483647)` 【注：#7FFF FFFF】 or `INT_MIN (-2147483648)` 【注：# 8000 0000】 is returned.

注意几个测试用例：

1. 不规则输入，但是有效，”-3924x8fc”，” + 413”,
2. 无效格式，” ++c”, ” ++1”
3. 溢出数据，”2147483648”

### 代码

```
int atoi(const char *str) {
    int num = 0;
    int sign = 1;
    const int len = strlen(str);
    int i = 0;

    while (str[i] == ' ' && i < len) i++; //删除前面的空格

    if (str[i] == '+') i++;

    if (str[i] == '-') {
        sign = -1;
        i++;
    }

    for (; i < len; i++) {
```

```
        if (str[i] < '0' || str[i] > '9') break;
        if (num > INT_MAX / 10 ||                    //再增加一位便溢出
                (num == INT_MAX / 10 &&
                        (str[i] - '0') > INT_MAX % 10)) {
            return sign == -1 ? INT_MIN : INT_MAX;
        }
        num = num * 10 + str[i] - '0';
    }
    return num * sign;
}
```

**相关题目**

与本题相同的题目：

- LeetCode String to Integer (atoi), http://leetcode.com/oldoj#question_8

与本题相似的题目：

- 无

# 1.3 字符串排序

# 1.4 单词查找树

# 1.5 子串查找

字符串的一种基本操作就是**子串查找** (substring search)：给定一个长度为 $N$ 的文本和一个长度为 $M$ 的模式串 (pattern string)，在文本中找到一个与该模式相符的子字符串。

最简单的算法是暴力查找，时间复杂度是 $O(MN)$。下面介绍两个更高效的算法。

## 1.5.1 KMP 算法

KMP 算法是 Knuth、Morris 和 Pratt 在 1976 年发表的。它的基本思想是，当出现不匹配时，就能知晓一部分文本的内容（因为在匹配失败之前它们已经和模式相匹配）。我们可以利用这些信息避免将指针回退到所有这些已知的字符之前。这样，当出现不匹配时，可以提前判断如何重新开始查找，而这种判断只取决于模式本身。

详细解释请参考《算法》[①]第 5.3.3 节。这本书讲的是确定有限状态自动机 (DFA) 的方法。

推荐网上的几篇比较好的博客，讲的是部分匹配表 (partial match table) 的方法（即 next 数组），“字符串匹配的 KMP 算法” http://t.cn/zTOPfdh，图文并茂，非常通俗易懂，作者是阮一

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

峰；“KMP 算法详解” http://www.matrix67.com/blog/archives/115，作者是顾森 Matrix67；”Knuth-Morris-Pratt string matching” http://www.ics.uci.edu/ eppstein/161/960227.html。

使用 next 数组的 KMP 算法的 C 语言实现如下。

kmp.c

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

/*
 * @brief 计算部分匹配表，即 next 数组.
 *
 * @param[in] pattern 模式串
 * @param[out] next next 数组
 * @return 无
 */
void compute_prefix(const char *pattern, int next[]) {
    int i;
    int j = -1;
    const int m = strlen(pattern);

    next[0] = j;
    for (i = 1; i < m; i++) {
        // 不匹配时回溯
        while (j > -1 && pattern[j + 1] != pattern[i]) j = next[j];
        // 匹配时 j++
        if (pattern[i] == pattern[j + 1]) j++;
        next[i] = j;
    }
}

/*
 * @brief KMP 算法.
 *
 * @param[in] text 文本
 * @param[in] pattern 模式串
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int kmp(const char *text, const char *pattern) {
    int i;
    int j = -1;
    const int n = strlen(text);
    const int m = strlen(pattern);
    if (n == 0 && m == 0) return 0; /* "","" */
    if (m == 0) return 0;  /* "a","" */
    int *next = (int*)malloc(sizeof(int) * m);

    compute_prefix(pattern, next);

    for (i = 0; i < n; i++) {
        while (j > -1 && pattern[j + 1] != text[i]) j = next[j];
        if (text[i] == pattern[j + 1]) j++;
        if (j == m - 1) { // 全部匹配结束
```

```
                free(next);
                return i-j;
            }
        }
        // 匹配失败
        free(next);
        return -1;
    }


    int main(int argc, char *argv[]) {
        char text[] = "ABC ABCDAB ABCDABCDABDE";
        char pattern[] = "ABCDABD";
        char *ch = text;
        int i = kmp(text, pattern);

        if (i >= 0) printf("matched @: %s\n", ch + i);
        return 0;
    }
```

kmp.c

```
#include<iostream>
#include<string>
using namespace std;

void calculte_next(const string &str, int *next){
    const int len = str.length();
    next[0] = 0;
    for(int i = 1, j = 0; i < len; ++i){
        while(j > 0 && str[j] != str[i]) j = next[j];
        if(str[j] == str[i]) j++;
        next[i] = j;
    }
}

int KMP(const string &text, const string &str){
    if(text.length() == 0 || str.length() == 0) return 0;

    const int len_text = text.length();
    const int len_str = str.length();
    int *next = new int[len_text];
    calculte_next(str, next);
```

```
    for(int i = 0, j = 0; i < len_text; ++i){
        while (j > 0 && str[j] != text[i]) j = next[j − 1];
        if (str[j] == text[i]) j++;
        if (j == len_str) return i − j + 1;
    }
    return −1;
}

int main(){
    string text = "ABC ABCDAB ABCDABCDABDE";
    string str = "ABCDABD";
    cout << KMP(text, str) << endl;
    return 0;
}
```

### 1.5.2 Boyer-Moore 算法【注：暂时先不看 2021/4/4】

详细解释请参考《算法》[①]第 5.3.4 节。

推荐网上的几篇比较好的博客，“字符串匹配的 Boyer-Moore 算法” http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html，图文并茂，非常通俗易懂，作者是阮一峰；Boyer-Moore algorithm, http://www-igm.univ-mlv.fr/ lecroq/string/node14.html。

有兴趣的读者还可以看原始论文[②]。

Boyer-Moore 算法的 C 语言实现如下。

boyer_moore.c

```
/**
 * 本代码参考了 http://www-igm.univ-mlv.fr/~lecroq/string/node14.html
 * 精力有限的话，可以只计算坏字符的后移，好后缀的位移是可选的，因此可以删除
 * suffixes(), pre_gs() 函数
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ASIZE 256  /* ASCII 字母的种类 */

/*
 * @brief 预处理，计算每个字母最靠右的位置.
 *
```

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

[②]BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

```
 * @param[in] pattern 模式串
 * @param[out] right 每个字母最靠右的位置
 * @return 无
 */
static void pre_right(const char *pattern, int right[]) {
    int i;
    const int m = strlen(pattern);

    for (i = 0; i < ASIZE; ++i) right[i] = -1;
    for (i = 0; i < m; ++i) right[(unsigned char)pattern[i]] = i;
}


static void suffixes(const char *pattern, int suff[]) {
    int f, g, i;
    const int m = strlen(pattern);

    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && pattern[g] == pattern[g + m - 1 - f])
                --g;
            suff[i] = f - g;
        }
    }
}

/*
 * @brief 预处理，计算好后缀的后移位置.
 *
 * @param[in] pattern 模式串
 * @param[out] gs 好后缀的后移位置
 * @return 无
 */
static void pre_gs(const char pattern[], int gs[]) {
    int i, j;
    const int m = strlen(pattern);
    int *suff = (int*)malloc(sizeof(int) * (m + 1));

    suffixes(pattern, suff);

    for (i = 0; i < m; ++i) gs[i] = m;

    j = 0;
    for (i = m - 1; i >= 0; --i) if (suff[i] == i + 1)
        for (; j < m - 1 - i; ++j) if (gs[j] == m)
            gs[j] = m - 1 - i;
```

```
    for (i = 0; i <= m - 2; ++i)
        gs[m - 1 - suff[i]] = m - 1 - i;
    free(suff);
}

/**
 * @brief Boyer-Moore 算法.
 *
 * @param[in] text 文本
 * @param[in] pattern 模式串
 * @return 成功则返回第一次匹配的位置, 失败则返回-1
 */
int boyer_moore(const char *text, const char *pattern) {
    int i, j;
    int right[ASIZE];  /* bad-character shift */
    const int n = strlen(text);
    const int m = strlen(pattern);
    int *gs = (int*)malloc(sizeof(int) * (m + 1));  /* good-suffix shift */

    /* Preprocessing */
    pre_right(pattern, right);
    pre_gs(pattern, gs);

    /* Searching */
    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && pattern[i] == text[i + j]; --i);

        if (i < 0) { /* 找到一个匹配 */
            /* printf("%d ", j);
            j += bmGs[0]; */
            free(gs);
            return j;
        } else {
            const int max = gs[i] > right[(unsigned char)text[i + j]] -
                    m + 1 + i ? gs[i] : i - right[(unsigned char)text[i + j]];
            j += max;
        }
    }
    free(gs);
    return -1;
}


int main() {
    const char *text="HERE IS A SIMPLE EXAMPLE";
    const char *pattern = "EXAMPLE";
    const int pos = boyer_moore(text, pattern);
    printf("%d\n", pos); /* 17 */
    return 0;
}
```

boyer_moore.c

### 1.5.3 Rabin-Karp 算法【注：暂时先不看 2021/4/4】

详细解释请参考《算法》[1]第 5.3.5 节。

Rabin-Karp 算法的 C 语言实现如下。

rabin_karp.c

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

const int R = 256;  /** ASCII 字母表的大小，R 进制 */
/** 哈希表的大小，选用一个大素数，这里用 16 位整数范围内最大的素数 */
const long Q = 0xfff1;

/*
 * @brief 哈希函数.
 *
 * @param[in] key 待计算的字符串
 * @param[int] M 字符串的长度
 * @return 长度为 M 的子字符串的哈希值
 */
static long hash(const char key[], const int M) {
    int j;
    long h = 0;
    for (j = 0; j < M; ++j) h = (h * R + key[j]) % Q;
    return h;
}

/*
 * @brief 计算新的 hash.
 *
 * @param[int] h 该段子字符串所对应的哈希值
 * @param[in] first 长度为 M 的子串的第一个字符
 * @param[in] next 长度为 M 的子串的下一个字符
 * @param[int] RM R^(M-1) % Q
 * @return 起始于位置 i+1 的 M 个字符的子字符串所对应的哈希值
 */
static long rehash(const long h, const char first, const char next,
                   const long RM) {
    long newh = (h + Q - RM * first % Q) % Q;
    newh = (newh * R + next) % Q;
    return newh;
}

/*
 * @brief Las Vegas version, do real comparison.
 *
 * @param[in] pattern 模式串
 * @param[in] substring 原始文本长度为 M 的子串
 * @return 两个字符串相同，返回 1，否则返回 0
 */
static int check(const char *pattern, const char s[]) {
```

[1]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

```
    return strcmp(pattern, s);
}

/*
 * Monte Carlo version, always return true.
 */
static int check(const char *pattern, const char s[]) {
    return 1;
}

/**
 * @brief Rabin-Karp 算法.
 *
 * @param[in] text 文本
 * @param[in] n 文本的长度
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int rabin_karp(const char *text, const char *pattern) {
    int i;
    const int n = strlen(text);
    const int m = strlen(pattern);
    const long pattern_hash = hash(pattern, m);
    long text_hash = hash(text, m);
    int RM = 1;
    for (i = 0; i < m - 1; ++i) RM = (RM * R) % Q;

    for (i = 0; i <= n - m; ++i) {
        if (text_hash == pattern_hash) {
            if (check(pattern, &text[i])) return i;
        }
        text_hash = rehash(text_hash, text[i], text[i + m], RM);
    }
    return -1;
}


int main() {
    const char *text = "HERE IS A SIMPLE EXAMPLE";
    const char *pattern = "EXAMPLE";
    const int pos = rabin_karp(text, pattern);
    printf("%d\n", pos); /* 17 */
    return 0;
}
```

rabin_karp.c

### 1.5.4 总结

| 算法 | 版本 | 复杂度 | | 在文本中回退 | 正确性 | 辅助空间 |
|---|---|---|---|---|---|---|
| | | 最坏情况 | 平均情况 | | | |
| KMP 算法 | 完整的 DFA | 2N | 1.1N | 否 | 是 | MR |
| | 部分匹配表 | 3N | 1.1N | 否 | 是 | M |
| | 完整版本 | 3N | N/M | 是 | 是 | R |
| Boyer-Moore 算法 | 坏字符向后位移 | MN | N/M | 是 | 是 | R |
| Rabin-Karp 算法 * | 蒙特卡洛算法 | 7N | 7N | 否 | 是 * | 1 |
| | 拉斯维加斯算法 | $7N^*$ | 7N | 是 | 是 | 1 |

* 概率保证，需要使用均匀和独立的散列函数

## 1.6 正则表达式