字符函数&内存函数使用和剖析

笔记本: My Notebook
创建时间: 2023/10/16 14:38 **更新时间:** 2023/10/20 22:32
作者: lwb

函数介绍
求字符串长度
　strlen
长度不受限制的字符串函数 - 遇到'\0'才停止
　strcpy
　strcat
　strcmp
长度受限制的字符串函数
　strncpy
　strncat
　strncmp
字符串查找
　strstr
　strtok
错误信息报告
　strerror

strlen
size_t strlen (const char * str)
字符串已经'\0'作为结束标志，strlen函数返回的是在字符串中'\0'前面出现的字符个数（不包含'\0'）
参数指向的字符串必须要以'\0'结束
注意函数的返回值为size_t，是无符号的(易错)

```
#include <stdio.h>
int main()
{
const char*str1 = "abcdef";
const char*str2 = "bbb";
if(strlen(str2)-strlen(str1)>0) // 恒为真
{
printf("str2>str1\n");
}
else
{
printf("srt1>str2\n");
}
return 0;
}
```

strlen函数的模拟实现

```
//1.计数法
int my_strlen(const char* str)
{
        int count = 0;
        assert(str != NULL);
        while (*str != '\0')
        {
               count++;
               str++;
        }
```

```
        return count;
}

//递归
int my_strlen(const char* str)
{
        if (*str != '\0')
        {
                return my_strlen(str + 1) + 1;
        }
        else
        {
                return 0;
        }
}
```

strcpy
char* strcpy(char* destination, const char* source);
注意事项：
源字符串必须以'\0'结束
会将源字符串中的'\0'拷贝到目标空间
目标空间必须足够大，以确保能存放源字符串
目标空间必须可变 - 不能是常量字符串
模拟实现

```
char* my_strcpy(char* dest, const char* src)
{
        assert(dest != NULL);
        assert(src != NULL);
        char* ret = dest;
        // 拷贝src指向的字符串到dest指向的空间，包含'\0'
        while (*dest++ = *src++)
        {
                ;
        }
        //目的空间的地址
        return ret;
}
```

strcat
char* strcat(char* destination, const char* source)
源字符串和目标都必须以'\0'结束
目标空间必须足够大，能够容纳源字符串的内容
目标空间必须可修改
字符串不能自己给自己追加，会导致死循环

```
char* my_strcat(char* dest, const char* src)
{
        assert(dest != NULL);
        assert(src != NULL);
        char* ret = dest;
        while (*dest != '\0')
        {
                dest++;
        }
        while (*dest++ = *src++)
        {
                ;
        }
        return ret;
}
```

strcmp - 比较字符串大小
int strcmp(const char* str1, const char* str2);
标准规定：

第一个字符串大于第二个字符串，则返回大于0的数字
第一个字符串等于第二个字符串，则返回0
第一个字符串小于第二个字符串，则返回小于0的数字

```
// 模拟实现
int my_strcmp(const char* str1, const char* str2)
{
        assert(str1 && str2);
        // cmp
        while (*str1 == *str2)
        {
               if (*str1 == '\0')
                       return 0;
               str1++;
               str2++;
        }
        return *str1 - *str2;
}
```

strncpy
char* strncpy(char* destination, const char* source, size_t num);
拷贝num个字符从源字符串到目标空间
如果源字符串的长度小于num，则拷贝完源字符串之后，在目标的后边追加0，直到num个。

```
// 模拟实现strncpy
char* my_strncpy(char* dest, const char* src, int count)
{
        char* start = dest;
        while (count && (*dest++ = *src++))
               count--;
        if (count)
               while (--count)
                       *dest++ = '\0';
        return start;
}
```

strncat
char* strncat(char* destination, const char* source, size_t num);
如果追加的长度比我的source长，补完source就不管了
不管要追加几个，后面都会补加一个'\0'

```
// 模拟实现strncat
char* my_strncat(char* front, const char* back, int count)
{
        char* start = front;;
        while (*front++);
        front--;
        while (count--)
               if (!(*front++ = *back++))
                       return start;
        *front = '\0';
        return start;
}
```

strncmp
int strncmp(const char *string1, const char *string2, size_t num);
比较到出现另个字符不一样或者一个字符串结束或者num个字符全部比较完成

```
// 模拟实现
int strncmp(const char *str1, const char *str2, int num)
{
    if(!num)
        return 0;
    while (--num && *str1 && *str1 == *str2)
    {
        str1++;
```

```
        str2++;
    }
    return (*(unsigned char*)str1 - *(unsigned char)str2);
}
```

strstr - 查找子字符串
char* strstr(const char* str, const char* str2);
返回str2在str1中出现的第一个位置，不存在则返回NULL

```
//暴力实现
char* my_strstr(const char* p1, const char* p2)
{
        assert(p1 != NULL);
        assert(p2 != NULL);
        if (*p2 == '\0')
               return p1;
        char* cp = p1;
        char* s1, *s2;
        while (*cp)
        {
               s1 = cp;
               s2 = (char*)p2;
               while (*s1 && *s2 && *s1 == *s2)
               {
                       s1++;
                       s2++;
               }
               if (!*s2)
               {
                       return cp;
               }
               cp++;
        }
        return NULL;
}
```

```
// KMP算法优化
思路：求出next数组，next[j]标识子串的前缀T[0~j] == 后缀T[i-j~i]
void NextVal(char T[], int* next)
{
        int len = strlen(T);
        int k = -1;
        int j = 0;
        next[0] = -1;
        while (j < len)
        {
               if (k == -1 || T[j] == T[k])
               {
                       j++;
                       k++;
                       if (T[j] != T[k])
                       {
                              next[j] = k;
                       }
                       else
                       {
                              next[j] = next[k];
                       }
               }
               else
               {
                       k = next[k];
               }
        }
}
int KMP(const char S[], const char T[])
{
```

```
        assert(S != NULL);
        assert(T != NULL);
        int i = 0, j = 0, lenS, lenT;
        lenS = strlen(S);
        lenT = strlen(T);
        int ne[maxsize];
        NextVal(T, ne);
        while (i < lenS && j < lenT)
        {
                if (j == -1 || S[i] == T[j])
                {
                        i++;
                        j++;
                }
                else
                {
                        j = ne[j];
                }
                if (j == lenT)
                {
                        return i - j;
                }
        }
        return -1;
}
```

strtok

char* strtok(char* str, const char* sep);

sep参数是个字符串，定义了用作分隔符的字符集合

第一个参数指定一个字符串，它包含了0个或者多个由sep字符串中一个或者多个分隔符分隔的标记

strtok函数会找到str中的下一个标记，并将其用'\0'结尾，返回一个指向这个标记的指针

strtok函数会改变被操作的字符串，所以在使用strtok函数切分的字符串一般都是临时拷贝的内容并且可修改

strtok函数的第一个参数不为NULL，函数将找到str中第一个标记，strtok函数将保存它在字符串中的位置

strtok函数的第一个参数为NULL，函数将在同一个字符串中被保存的位置开始，查找下一个标记

如果字符串不存在更多的标记，则返回NULL指针

```
//模拟实现
//inline - 内联函数，为了解决一些频繁的调用的小函数大量消耗空间（栈空间）的问题，它只是一个对编译器的建议，最终是否采用还是看编译器的意思
// 栈空间就是指放置程序的局部数据的内存空间
inline int get_pos(unsigned char x)
{
        return x % 32;
}
char* my_strtok(char* s, const char* ct)
{
        char* sbegin, * send;
        static char* ssave = NULL;
        sbegin = s ? s : ssave; // 如果s为NULL就继续上一次的缓存
        unsigned char cset[32] = { 0 }; // 用32个unsigned char对每个位进行bool运算可以更节省内存
        while ((*ct) != '\0') // 更新set
        {
                unsigned char t = (unsigned char)*ct++;
                cset[get_pos(t)] |= 1 << (t / 32); // 映射分隔符
        }
        // 让sbegin指向不在set中的位置
        while (*sbegin != '/0' && (cset[get_pos(*sbegin)] & (1 << ((unsigned char)*sbegin  / 32))))
        {
                ++sbegin;
        }
        if (*sbegin == '\0')
        {
```

```
                ssave = NULL;
                return NULL;
        }
        int idx = 0;
        // 寻找下一个分隔符的位置
        while (sbegin[idx] != '\0' && !(cset[get_pos(sbegin[idx])] & (1 <<
((unsigned  char)sbegin[idx]) / 32)))
        {
                ++idx;
        }
        send = sbegin + idx;
        if (*send != '\0')
        {
                *send++ = '\0'; // 画上终止符
        }
        ssave = send; // 更新下一次处理的缓存位置
        return sbegin;
}
int main()
{
        char arr[] = "zpw@bitedu.tech";
        char* p = "@.";
        char buf[1024] = { 0 };
        strcpy(buf, arr);
        char* ret = NULL;
        for (ret = my_strtok(arr, p); ret != NULL; ret = my_strtok(NULL, p)) //
会创建静态变量，保存之前返回的标记位置
        {
                printf("%s\n", ret);
        }
        /*char* ret = strtok(arr, p);
        printf("%s\n", ret);
        ret = strtok(NULL, p);
        printf("%s\n", ret);
        ret = strtok(NULL, p);
        printf("%s\n", ret);*/
        return 0;
}
```

strerror
char* strerror(int errnum);
返回错误码，所对应的错误信息
例
0 - No error
1 - Operation not permitted
2 - No such file or directory
errno是一个全局的错误码变量，需要引errno.h头文件

```
FILE* pf = fopen("test.txt", "r");
if (pf == NULL)
{
    printf("%s\n", strerror(errno));
}
    else
{
    printf("open file success\n");
}
    return 0;
```

字符分类函数
函数 如果他的参数符合下列条件就返回真
#include <ctype.h>
iscntrl　任何控制字符
isspace　空白字符：空格 ‘ ’ ，换页 ‘\f’ ，换行'\n'，回车 ‘\r’ ，制表符'\t'或者垂直制表符'\v'
isdigit　十进制数字 0~9

isxdigit　十六进制数字，包括所有十进制数字，小写字母a~f，大写字母A~F
islower　小写字母a~z
isupper　大写字母A~Z
isalpha　字母a~z或A~Z
isalnum　字母或者数字，a~z,A~Z,0~9
ispunct　标点符号，任何不属于数字或者字母的图形字符（可打印）
isgraph　任何图形字符
isprint　任何可打印字符，包括图形字符和空白字符

字符转换
int tolower(int c); // 将字符转换成小写
int toupper(int c); // 将字符转换成大写

strcpy strcat strcmp strncpy strncat strncmp 操作的对象是字符串，\0
整形数组、浮点型数组、结构体数组就需要使用memcpy

memcpy
void *memcpy(void* destination, const void* source, size_t num);
void* 通用类型的指针-无类型指针
函数memcpy从source的位置开始向后复制num个字节的数字到destination的内存位置
这个函数在遇到'\0'的时候并不会停下来
如果source和destination有任何的重叠，复制的结果都是未定义的 - 原来的数据可能会被改变，一旦被改变了，复制的数据有不一样了

```
//模拟实现memcpy
void* my_memcpy(void* dest, const void* src, size_t num) // 拷贝的字节数 - num
{
        void* ret = dest;
        assert(dest != NULL);
        assert(src != NULL);
        while (num--) {
               *(char*)dest = *(char*)src;
               ++(char*)dest;
               ++(char*)src;
        }
        return ret;
}
```

memmove
void* memmove (void* destination, const void* source, size_t num);
和memcpy的差别就是memmove函数处理的源内存块和目标内存块是可以重叠的
如果源空间和目标空间出现重叠，就得使用memmove函数处理

```
// 模拟实现memmove，处理重叠内存的拷贝
void* my_memmove(void* dest, const void* src, size_t count)
{
        void* ret = dest;
        assert(dest != NULL);
        assert(src != NULL);
        if (dest < src)
        {
               // 由前向后拷贝
               while (count--)
               {
                       *(char*)dest = *(char*)src;
                       ++(char*)dest;
                       ++(char*)src;
               }
        }
        else
        {
               // 由后向前拷贝
               while (count--)
               {
```

```
                        *((char*)dest + count) = *(((char*)src + count));
                        // (char*)dest + count - 最后一个字节的地址，然后随着count--，不断往前拷贝
                }
        }
        return ret;
}
```

memcmp
int memcmp(const void* ptr1, const void* ptr2, size_t num);
比较从ptr1和ptr2指针开始的num个字节

memset - 内存设置,注意是设置num个字节