



软件工程师丛书

C语言函数大全

张翔 袁岚 张晓芸 等编著



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

821

TP312C
Z35



软件工程师丛书

C 语言函数大全

张翔 袁岚 张晓芸 等编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 提 要

本书介绍了包括 Unix C, Turbo C 和 Microsoft C 共 3 类 C 语言函数, 列出了每个函数的功能、语法格式和使用说明等, 涉及到了当今 C 语言使用的各个层面。

本书适用于使用 C 语言进行软件开发的人员, 是一本覆盖面大、用途广泛的工具书。本书也可作为 C 语言初学者的参考书。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目 (CIP) 数据

C 语言函数大全 / 张翔等编著. —北京: 电子工业出版社, 2002.4
(软件工程师丛书)
ISBN 7-5053-7538-5

I. C... II. 张... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2002) 第 017051 号

责任编辑: 段来盛

印 刷: 北京市天竺颖华印刷厂

出版发行: 电子工业出版社 <http://www.phei.com.cn>

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 26.5 字数: 626 千字

版 次: 2002 年 4 月第 1 版 2002 年 4 月第 1 次印刷

印 数: 6000 册 定价: 40.00 元

凡购买电子工业出版社的图书, 如有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系。联系电话: (010) 68279077

目 录

| | | |
|--------------|-------------------|------------|
| 第 1 章 | Unix C 函数 | 1 |
| 1.1 | 错误报告 | 2 |
| 1.2 | 内存分配 | 2 |
| 1.3 | 字符处理 | 11 |
| 1.4 | 字符串和数组处理 | 14 |
| 1.5 | 流输入/输出 | 26 |
| 1.6 | 底层输入/输出 | 42 |
| 1.7 | 文件系统接口 | 49 |
| 1.8 | 管道和队列 | 62 |
| 1.9 | 套接口 | 65 |
| 1.10 | 底层终端接口 | 78 |
| 1.11 | 数学函数 | 82 |
| 1.12 | 初等数学函数 | 87 |
| 1.13 | 搜索和排序 | 93 |
| 1.14 | 匹配 | 94 |
| 1.15 | 日期和时间 | 99 |
| 1.16 | 扩展字符集 | 107 |
| 1.17 | 本地和国际化 | 108 |
| 1.18 | 非本地退出 | 109 |
| 1.19 | 信号处理 | 110 |
| 1.20 | 进程的启动和终止 | 119 |
| 1.21 | 进程 | 123 |
| 1.22 | 任务控制 | 126 |
| 1.23 | 用户和群组 | 129 |
| 1.24 | 系统信息 | 138 |
| 1.25 | 系统配置参数 | 140 |
| 第 2 章 | Turbo C 函数 | 143 |
| 2.1 | ALLOC.H | 144 |
| 2.2 | ASSERT.H | 146 |
| 2.3 | BIOS.H | 146 |
| 2.4 | CONIO.H | 148 |
| 2.5 | CTYPE.H | 153 |

| | | |
|--------------|----------------------------|------------|
| 2.6 | DIR.H..... | 155 |
| 2.7 | DOS.H..... | 157 |
| 2.8 | FLOAT.H..... | 169 |
| 2.9 | GRAPHICS.H..... | 170 |
| 2.10 | IO.H..... | 184 |
| 2.11 | MATH.H..... | 190 |
| 2.12 | MEM.H..... | 195 |
| 2.13 | PROCESS.H..... | 197 |
| 2.14 | SETJMP.H..... | 198 |
| 2.15 | SIGNAL.H..... | 199 |
| 2.16 | STDIO.H..... | 199 |
| 2.17 | STDLIB.H..... | 210 |
| 2.18 | STRING.H..... | 218 |
| 2.19 | SYSSTAT.H..... | 224 |
| 2.20 | SYSTIMEB.H..... | 225 |
| 2.21 | TIME.H..... | 225 |
| 第 3 章 | Microsoft C 函数..... | 229 |
| 3.1 | 以字为单位的内存处理函数..... | 230 |
| 3.2 | 单个字符处理函数..... | 235 |
| 3.3 | 数字与字符串转换函数..... | 237 |
| 3.4 | 目录结构和信息处理函数..... | 244 |
| 3.5 | 文件处理函数..... | 248 |
| 3.6 | 图形字体处理函数..... | 255 |
| 3.7 | 图形原语函数..... | 259 |
| 3.8 | 图形和图表函数..... | 283 |
| 3.9 | 流 I/O 处理函数..... | 290 |
| 3.10 | 低端 I/O 函数..... | 304 |
| 3.11 | 控制台和端口 I/O 函数..... | 309 |
| 3.12 | 定位函数..... | 312 |
| 3.13 | 数学函数..... | 315 |
| 3.14 | 内存分配、释放及重新分配函数..... | 326 |
| 3.15 | 进程处理函数..... | 337 |
| 3.16 | 字符串处理函数..... | 353 |
| 3.17 | BIOS 中断服务..... | 361 |
| 3.18 | DOS 中断例程..... | 364 |
| 3.19 | 系统时间处理函数..... | 380 |
| 3.20 | 可变长参数列表..... | 386 |
| 索引 | | 387 |

第 1 章

Unix C 函数



本章分类介绍每个 Unix C 函数的功能、原型、原型所在的包含文件、使用说明等内容，对一些重要函数还给出了样例。

1.1 错误报告

perror

【功能】该函数在 `stderr` 流中输出错误消息。

【原型】`void perror (const char *message)`

【位置】`stdio.h` (ISO)

【说明】如果调用 `perror` 时使用的 `message` 参数是一个空指针或者一个空字符串，则 `perror` 函数将打印与 `errno` 相对应的错误信息，以及一个换行符。

如果提供的 `message` 参数非空，则 `perror` 函数将作为前缀首先输出该字符串的内容，然后添加一个冒号和空格字符，最后是 `errno` 相对应的错误信息。

strerror

【功能】将由参数 `errno` 指定的错误代码映射到描述错误信息的串，返回值是指向串的指针。

【原型】`char * strerror (int errno)`

【位置】`string.h` (ISO)

【说明】`errno` 的值通常来自 `errno`，不应修改 `strerror` 所返回的串，以后再次调用 `strerror` 时会重写该串。

strerror_r

【功能】该函数与 `strerror` 基本相同，只是使用该函数时错误信息写入的缓存区由用户提供，并且从 `buf` 开始，长度为 `n` 字节，而不是程序中所有线程共享的静态分配缓存区。

【原型】`char * strerror_r (int errno, char *buf, size_t n)`

【位置】`string.h` (GNU)

【说明】至多写 `n` 个字符，除非用户选择足够大的缓存。由于无法保证返回的串属于当前线程的最后调用，所以常用于多线程程序。

该函数是 GNU 扩展，其声明在 `string.h` 中。

1.2 内存分配

alloca

【功能】分配存储块。

【原型】`void * alloca (size_t size)`

【位置】`stdlib.h` (GNU, BSD)

【说明】该函数的返回值是 `size` 字节长的存储块地址，该存储块在调用函数的栈帧中分配。

不要在函数调用的参数中使用 `alloca` 函数，这将产生不可预知的结果。因为用于 `alloca` 的栈空间在堆栈中将处在函数参数使用空间的中间位置，所以要避免出现像 `foo(x, alloca(4), y)` 这样的调用。

calloc

【功能】分配内存并且将内存清 0。

【原型】`void * calloc (size_t count, size_t eltsize)`

【位置】`malloc.h, stdlib.h (ISO)`

【说明】该函数分配一块包含 `count` 个元素的内存，其中每个元素的大小为 `eltsize`。其内容在 `calloc` 返回之前清 0。

例如，可以定义 `calloc` 如下。

```
void * calloc (size_t count, size_t eltsize)
{
    size_t size = count * eltsize;
    void *value = malloc (size);
    if (value != 0)
        memset (value, 0, size);
    return value;
}
```

但是通常不能保证 `calloc` 在内部调用 `malloc`。如果某个应用程序在 C 函数库之外提供自己的 `malloc/realloc/free`，则也应当同时提供 `calloc`。

cfree

【功能】释放存储空间。

【原型】`void cfree (void *ptr)`

【位置】`stdlib.h (Sun)`

【说明】与 `free` 函数的功能类似。提供该函数是为了与 SUN OS 兼容，因此应当尽量使用 `free` 函数。

free

【功能】释放由指针 `ptr` 指向的存储空间。

【原型】`void free (void *ptr)`

【位置】`malloc.h, stdlib.h (ISO)`

mallinfo

【功能】该函数返回结构中当前的动态内存使用情况，该结构为 `struct mallinfo` 类型。

【原型】`struct mallinfo mallinfo (void)`



【位置】 malloc.h (SVID)

malloc

【功能】该函数用于分配一个新的存储块。

【原型】 void * malloc (size_t size)

【位置】 malloc.h, stdlib.h (ISO)

【说明】该函数返回一个指针，指向新分配的 size 字节长的存储块，如果返回空指针，则说明没有执行分配动作。

存储块的内容没有定义，必须自己进行初始化(或者使用 calloc 函数实现)。通常需要将返回值强制转换类型。以下提供一些样例，说明如何使用 memset 库函数进行初始化。

```
struct foo *ptr;
...
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

事实上不强制转换类型也可以将 malloc 的结果保存在任何一个指针变量中，因为 ISO C 在必要时自动将 void * 类型转换为其他类型的指针。但是如果不是使用赋值操作符或者需要在传统 C 环境中运行代码，则必须强制转换类型。

为字符串分配空间时，malloc 的参数必须是字符串长度加 1。这是因为尽管字符串结尾的空字符不计算在字符串的长度内，但是该字符同样需要一个字节长的空间。例如：

```
char *ptr;
...
ptr = (char *) malloc (length + 1);
```

mcheck

【功能】调用 mcheck 函数将通知 malloc 执行临时的一致性检查，诸如写操作超过 malloc 分配的存储块结尾等错误情况。

【原型】 int mcheck (void (*abortfn) (enum mcheck_status status))

【位置】 malloc.h (GNU)

【说明】abortfn 参数是发现不一致情况之后调用的函数。如果该参数提供一个空指针，则 mcheck 使用一个默认函数来打印一条消息，然后调用 abort 函数。所提供的函数在调用时带有一个参数，说明所检查到的不一致类型。

如果在使用 malloc 执行分配动作之后再分配检查，则为时已晚。此时 mcheck 不会做任何动作，并返回-1；否则返回 0(表示成功完成)。

为了尽早安排调用 mcheck，最简单的方法是在链接程序时使用-lmcheck 选项，之后将完全不需要再改动源程序。

memalign

【功能】该函数在 boundary 的倍数位置分配一块 size 大小的字节。

【原型】 `void * memalign (size_t boundary, size_t size)`

【位置】 `malloc.h, stdlib.h (BSD)`

【说明】 `boundary` 必须是 2 的阶乘的幂。函数执行时首先分配一个较大的块，然后返回块中一个位于指定边界的地址。

obstack_alloc

【功能】 该函数在对象堆栈中分配 `size` 个字节的存储块，不进行初始化，然后返回其地址。

【原型】 `void * obstack_alloc (struct obstack *obstack_ptr, int size)`

【位置】 `obstack.h (GNU)`

【说明】 这里的 `obstack_ptr` 用于指定在哪个对象堆栈中分配块，即对象堆栈中 `struct obstack` 对象的地址。每个 `obstack` 函数或宏都要求指定一个 `obstack_ptr` 作为第 1 参数。

如果需要分配一部分新的内存，则该函数调用对象堆栈的 `obstack_chunk_alloc` 函数；如果 `obstack_chunk_alloc` 返回，则该函数返回一个空指针。这种情况下，对象堆栈中分配的内存量没有改变。如果所提供的 `obstack_chunk_alloc` 函数在内存不足时调用 `exit` 或者 `longjmp`，那么 `obstack_alloc` 不返回空指针。

例如，下面的函数在一个指定的对象堆栈中分配字符串 `str` 的备份。

```
struct obstack string_obstack;
char *
copystring (char *string)
{
    size_t len = strlen (string) + 1;
    char *s = (char *) obstack_alloc (&string_obstack, len);
    memcpy (s, string, len);
    return s;
}
```

obstack_base

【功能】 该函数返回对象堆栈 `obstack_ptr` 中当前增长对象的起始地址。

【原型】 `void * obstack_base (struct obstack *obstack_ptr)`

【位置】 `obstack.h (GNU)`

【说明】 如果立即结束对象则可以即刻得到返回地址；如果对象还需要增长，则可能超出当前的块，其地址也将会发生改变。

如果没有正在增长的对象，则该函数的返回值说明了下一个分配对象的起始位置(再次假设其符合当前块的大小)。

obstack_blank_fast

【功能】 该函数为对象堆栈 `obstack_ptr` 中的增长对象添加 `size` 个字节，但不进行初始化。

【原型】 `void obstack_blank_fast (struct obstack *obstack_ptr, int size)`

【位置】 `obstack.h` (GNU)

【说明】使用 `obstack_room` 函数检查空间时，如果没有足够的空间可添加，那么使用快速增长函数将不够安全。在这种情况下，只需使用相应的普通增长函数，即立即将对象拷贝至新块，然后再次提供大量可用空间。

因此，每次使用普通增长函数之前，都需要使用 `obstack_room` 函数检查空间是否充足。一旦对象被拷贝至新块，就可以再次提供充足的空间，从而程序可以再次使用快速增长函数。

以下是一个样例。

```
void
add_string (struct obstack *obstack, const char *ptr, int len)
{
    while (len > 0)
    {
        int room = obstack_room (obstack);
        if (room == 0)
        {
            /* Not enough room. Add one character slowly,
               which may copy to a new chunk and make room. */
            obstack_lgrow (obstack, *ptr++);
            len--;
        }
        else
        {
            if (room > len)
                room = len;
            /* Add fast as much as we have room for. */
            len -= room;
            while (room-- > 0)
                obstack_lgrow_fast (obstack, *ptr++);
        }
    }
}
```

obstack_blank

【功能】该函数是为增长对象添加字节的最基础函数，添加空间之后不进行初始化。

【原型】 `void obstack_blank (struct obstack *obstack_ptr, int size)`

【位置】 `obstack.h` (GNU)

obstack_copy0

【功能】该函数与 `obstack_copy` 类似，但是 `obstack_copy0` 将追加一个额外的字节，其中包含一个空字符。

【原型】 `void * obstack_copy0 (struct obstack *obstack_ptr, void *address, int size)`

【位置】 `obstack.h` (GNU)

〔说明〕 额外追加的字节不计算在参数 `size` 的大小中。

在将字符序列作为以空字符结尾的字符串拷贝到对象堆栈中时,使用该函数非常便捷。以下是一个样例。

```
char *
obstack_savestring (char *addr, int size)
{
    return obstack_copy0 (&myobstack, addr, size);
}
```

obstack_copy

〔功能〕 该函数通过拷贝从 `address` 开始的 `size` 个字节的数据来分配一个数据块,并进行初始化。

〔原型〕 `void * obstack_copy (struct obstack *obstack_ptr, void *address, int size)`

〔位置〕 `obstack.h` (GNU)

〔说明〕 在遇到与 `obstack_alloc` 相同的情形时,该函数将返回一个空指针。

obstack_finish

〔功能〕 对象增长完毕时,使用该函数关闭对象并返回其最终地址。

〔原型〕 `void * obstack_finish (struct obstack *obstack_ptr)`

〔位置〕 `obstack.h` (GNU)

〔说明〕 一旦对象增长完毕,对象堆栈就可以用来完成普通的分配操作,或者增长另一个对象。

该函数在遇到 `obstack_alloc` 相同的情况时返回一个空指针。

增长并构建一个对象时,可能需要了解该对象最终的大小。此时无需跟踪对象的增长过程,因为可以在对象增长完毕之前使用 `obstack_object_size` 函数得到对象堆栈的长度。

obstack_free

〔功能〕 如果 `object` 是一个空指针,则释放对象堆栈中分配的所有对象。

〔原型〕 `void obstack_free (struct obstack *obstack_ptr, void *object)`

〔位置〕 `obstack.h` (GNU)

〔说明〕 如果 `object` 不是一个空指针,则必须是对象堆栈中分配的一个对象的地址。随后函数释放 `object` 及对象堆栈中从 `object` 之后分配的所有对象。

注意,如果 `object` 是一个空指针,则结果将得到一个未初始化的对象堆栈。要释放对象堆栈中的所有内容,且允许进行下一次分配,必须使用对象堆栈中分配的第 1 个对象的地址调用 `obstack_free`:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

对象堆栈中的对象被组合成组块。当某个组块中的所有对象都被释放时, `obstack` 将自动释放组块。随后,其他对象堆栈或非对象堆栈分配时可以重新使用组块的空间。

obstack_grow

【功能】需要添加一块初始化空间时使用该函数，该函数是增长对象的 `obstack_copy` 函数。

【原型】`void obstack_grow (struct obstack *obstack_ptr, void *data, int size)`

【位置】`obstack.h` (GNU)

【说明】该函数为增长对象添加 `size` 个字节的数据，从 `data` 中复制内容。

obstack_grow0

【功能】该函数是增长对象的 `obstack_copy0` 函数。

【原型】`void obstack_grow0(struct obstack *obstack_ptr, void *data, int size)`

【位置】`obstack.h` (GNU)

【说明】该函数添加从 `data` 中复制来的 `size` 个字节，随后添加一个额外的空字符。

obstack_init

【功能】初始化对象堆栈 `obstack_ptr`，以准备分配对象。

【原型】`int obstack_init (struct obstack *obstack_ptr)`

【位置】`obstack.h` (GNU)

【说明】该函数调用 `obstack_chunk_alloc` 函数。如果 `obstack_chunk_alloc` 函数返回一个空指针，则该函数返回 0，说明内存不足；否则返回 1。如果在内存不足时提供 `obstack_chunk_alloc` 函数调用 `exit` 或者 `longjmp`，则可以忽略 `obstack_init` 的返回值。

以下两个样例说明如何为对象堆栈分配空间，并进行初始化。首先声明 `obstack` 是一个静态变量。

```
static struct obstack myobstack;
...
obstack_init (&myobstack);
```

然后由对象堆栈自己完成动态分配。

```
struct obstack *myobstack_ptr
= (struct obstack *) xmalloc (sizeof (struct obstack));
obstack_init (myobstack_ptr);
```

obstack_int_grow

【功能】该函数为对象堆栈 `obstack_ptr` 中的增长对象添加一个 `int` 类型的数值。

【原型】`void obstack_int_grow_fast (struct obstack *obstack_ptr, int data)`

【位置】`obstack.h` (GNU)

【说明】函数添加 `sizeof` 个 `int` 类型的字节，并且将其初始化为 `data`。

obstack_int_grow_fast

【功能】该函数为对象堆栈 `obstack_ptr` 中的增长对象添加 `sizeof` 个字节，其中包含 `data`

值。

【原型】 `void obstack_int_grow_fast (struct obstack *obstack_ptr, int data)`

【位置】 `obstack.h` (GNU)

obstack_lgrow

【功能】 使用该函数为对象堆栈的增长对象添加字符，每次添加一个字节。添加的字节为字符 `c`。

【原型】 `void obstack_lgrow (struct obstack *obstack_ptr, char c)`

【位置】 `obstack.h` (GNU)

obstack_1grow_fast

【功能】 该函数为对象堆栈 `obstack_ptr` 中的增长对象添加一个字节，添加的字节为字符 `c`。

【原型】 `void obstack_1grow_fast (struct obstack *obstack_ptr, char c)`

【位置】 `obstack.h` (GNU)

obstack_next_free

【功能】 该函数返回 `obstack_ptr` 中当前组块的第 1 个空闲字节的地址。

【原型】 `void * obstack_next_free (struct obstack *obstack_ptr)`

【位置】 `obstack.h` (GNU)

【说明】 如果没有增长对象，则该函数的返回值与 `obstack_base` 相同。

obstack_object_size

【功能】 该函数返回当前增长对象的字节数。

【原型】 `int obstack_object_size (struct obstack *obstack_ptr)`

【位置】 `obstack.h` (GNU)

【说明】 该函数等效于如下表达式。

```
obstack_next_free (obstack_ptr) - obstack_base (obstack_ptr)
```

obstack_ptr_grow

【功能】 该函数为指针值加 1。

【原型】 `void obstack_ptr_grow(struct obstack *obstack_ptr, void *data)`

【位置】 `obstack.h` (GNU)

【说明】 添加 `sizeof` 个字节，其中包含 `data` 值。

obstack_ptr_grow_fast

【功能】 该函数为 `obstack_ptr` 中的增长对象添加 `sizeof` 个字节，其中包含 `data` 值。

【原型】 `void obstack_ptr_grow_fast (struct obstack *obstack_ptr, void *data)`



【位置】 `obstack.h` (GNU)

obstack_room

【功能】 该函数返回可以使用快速增长函数为对象堆栈的当前增长对象安全添加的字节数。

【原型】 `int obstack_room (struct obstack *obstack_ptr)`

【位置】 `obstack.h` (GNU)

【说明】 当确认有足够空间时，可以使用快速增长函数为增长对象添加数据。

r_alloc

【功能】 该函数用于分配一块可重定位块，大小为 `size` 个字节。

【原型】 `void r_alloc (void **handleptr, size_t size)`

【位置】 `malloc.h` (GNU)

【说明】 该函数在 `*handleptr` 中保存块地址，并且返回一个空指针说明函数执行成功。如果函数得不到所需的块，则在 `*handleptr` 中保存一个空指针，并返回一个空指针。

r_alloc_free

【功能】 该函数用于释放一块可重定位块。

【原型】 `void r_alloc_free (void **handleptr)`

【位置】 `malloc.h` (GNU)

【说明】 释放 `*handleptr` 指向的块，并在 `*handleptr` 中保存一个空指针，说明它不再指向某个分配块。

r_re_alloc

【功能】 该函数用于调整 `*handleptr` 指向的存储块的大小，并且令其长度等于 `size` 个字节。

【原型】 `void * r_re_alloc (void **handleptr, size_t size)`

【位置】 `malloc.h` (GNU)

【说明】 将调整大小之后的存储块地址保存在 `*handleptr` 中，并返回一个非空指针说明执行成功。

如果没有足够的内存，则函数返回一个空指针，并且不改变 `*handleptr` 的内容。

realloc

【功能】 该函数将地址为 `ptr` 的存储块大小更改为 `newsize`。

【原型】 `void * realloc (void *ptr, size_t newsize)`

【位置】 `malloc.h`, `stdlib.h` (ISO)

【说明】 因为存储块结尾后面的空间可能已经被使用，因此 `realloc` 可能必须将存储块拷贝到可用空间充足的新地址中。`realloc` 的值是存储块的新地址。如果需要移动存储块，则 `realloc` 将复制旧内容。

如果向 `ptr` 传递一个空指针，则 `realloc` 的动作类似 `malloc (newsize)`。这样可能很方便，但必须记住，旧版本(在 ISO C 之前)可能不支持这一动作，因此向 `realloc` 传递空指针可能导致系统崩溃。

该函数与 `malloc` 类似，如果没有足够的内存空间来扩大存储块，将返回一个空指针。此时不更改或重新定位原来的存储块。

在多数情况下，`realloc` 函数执行失败时不会改变原有存储块，因为内存不足时应用程序无法继续执行，惟一可做的动作就是发送致命错误消息。可以编写一个称为 `xrealloc` 的子程序，用来负责错误消息。

```
void *
xrealloc (void *ptr, size_t size)
{
    register void *value = realloc (ptr, size);
    if (value == 0)
        fatal ("Virtual memory exhausted");
    return value;
}
```

也可以根据自己的需要使用 `realloc` 函数缩小存储块的大小。在执行某些分配动作时，缩小存储块的大小必须进行拷贝，因此如果没有其他可用的空间，该动作将可能失败。

如果指定的新尺寸与原有尺寸相同，则 `realloc` 函数将不会改变任何内容，并返回原有地址。

valloc

【功能】该函数类似于 `memalign` 第 2 个参数为页面大小时的情况。

【原型】`void * valloc (size_t size)`

【位置】`malloc.h, stdlib.h (BSD)`

1.3 字符处理

isalnum

【功能】如果 `c` 是字母或者数字，则返回真。

【原型】`int isalnum (int c)`

【位置】`ctype.h (ISO)`

【说明】如果某个字符的 `isalpha` 函数为真或者 `isdigit` 函数为真，则该字符的 `isalnum` 函数也为真。

isalpha

【功能】如果 `c` 是字母则返回真。

【原型】`int isalpha (int c)`

【位置】 `ctype.h` (ISO)

【说明】如果某个字符的 `islower` 或者 `isupper` 函数返回真，则该字符的 `isalpha` 函数也返回真。

在某些场合下，可能存在一些例外的字符，其 `isalpha` 函数为真，而该字符既不是大写也不是小写。但是在标准 C 中不存在这样的例外字符。

isascii

【功能】如果 `c` 是符合 US/UK ASCII 字符集的 7 位无符号 `char` 值，则返回真。

【原型】 `int isascii (int c)`

【位置】 `ctype.h` (SVID, BSD)

【说明】该函数为 BSD 扩展，也是 SVID 扩展。

isblank

【功能】如果 `c` 为空白字符(即空格或者制表符)，则返回真。

【原型】 `int isblank (int c)`

【位置】 `ctype.h`(GNU)

【说明】该函数为 GNU 扩展。

isctrl

【功能】如果 `c` 为控制字符(也就是非打印字符)，则返回真。

【原型】 `int isctrl (int c)`

【位置】 `ctype.h` (ISO)

isdigit

【功能】如果 `c` 为十进制数字(从 0 到 9)，则返回真。

【原型】 `int isdigit (int c)`

【位置】 `ctype.h` (ISO)

isgraph

【功能】如果 `c` 为图形字符(即有相关轮廓的字符)，则返回真。

【原型】 `int isgraph (int c)`

【位置】 `ctype.h` (ISO)

【说明】空白字符不作为图形字符处理。

islower

【功能】如果 `c` 为小写字母，则返回真。

【原型】 `int islower (int c)`

【位置】 `ctype.h` (ISO)