软件工程师丛书

C语言函数大全

张 翔 裘 岚 张晓芸 等编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

软件工程师丛书

C语言函数大全

张 翔 裘 岚 张晓芸 等编著

電子工業出版社
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。

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

出 版 说 明

随着我国加入 WTO，现代化建设也将以前所未有的步伐向前迈进。我们面临更大的挑战，也面临更多的机遇。一个不争的事实是计算机的应用普及将更加深入，将需要数量更多、水平更高的软件工程师。

我国的软件工程师队伍已有了长足的发展，软件开发水平已有了长足的进步。作为中国人，我们期盼的是中国软件业走自主创新之路，在世界上的地位越来越高。作为出版工作者，为发展我国的软件事业尽最大努力，是我们义不容辞的责任，这正是我们于 1999 年底推出《软件工程师》丛书的初衷。

目前这套丛书已出版了 40 多种。从市场销售和读者反馈的情况看，这套丛书已经得到了读者的首肯和厚爱，这也是对我们下一步工作的激励。

可以说，计算机应用系统的多样化、规模化和复杂化对软件工程师提出了更高的要求，同时也为软件工程师提供了更多的施展个人才华的机会。

针对这种形势，我们正在扩充《软件工程师》丛书的选题范围，进一步界定这套丛书的特色，并把丛书按如下类型整合。

一是开发类，通过大量实例说明如何使用各种流行的高级语言、工具类软件开发不同的应用系统，说明开发思想、开发过程、难点及其解决方案。为了适应我国软件工程师开发综合软件系统的需求，我们把包含编程功能在内的高级应用软件的开发应用也纳入到丛书中。

二是技巧类，通过大量实例说明在不同应用系统开发过程中，有关缩短开发周期、提高开发质量、解决开发中的疑难问题的各种技巧。

三是技术类，介绍软件开发的有关理论和技术，以及在实践中的应用，如系统分析与系统设计、软件测试和系统安全等。

四是手册类，即每个软件工程师必备的案头书。

在新的一年开始之际，这套丛书从内容、开本、印刷及装帧等方面都将以全新的面貌与广大读者见面，目的在于使其更受读者的欢迎，每本书能容纳更多的信息。

我们以为软件工程师提供图书信息服务为宗旨，坚持以图书质量为生命。我们希望《软件工程师》丛书能对读者有所帮助，希望读者提出更多的宝贵建议和意见，包括工作中遇到的技术难点、疑点和问题。希望更多的作者加入我们的专家行列，推介自己的实践经验和累累硕果。我们的网址是 www.phei.com.cn，请和我们联系。

为了我国软件业的更加美好的明天，让我们共同努力。

電子工業出版社

前　言

C 语言是多年来国内外得到迅速推广使用的一种现代语言。C 语言功能丰富、表达能力强、使用灵活方便、应用面广、可移植性好，既具有高级语言的优点，又具有低级语言的许多特点。现在，C 语言已不仅为计算机专业人员所使用，而且也受到了广大计算机软件编程爱好者的喜爱。

随着操作系统的不断发展和演变，目前广泛使用的 C 语言可以大概分为 Microsoft C、Turbo C 和 Unix C 等 3 种类型。这 3 类 C 语言之间既有相同点，也存在差异。为了帮助大家更好地学习和使用 C 语言，本书按照功能分类介绍了这 3 类 C 语言函数的功能和使用方法，逐条给出了其语法格式、功能、使用说明等内容，对一些重要函数还给出了使用样例。同时，为了使读者在查找函数时更加方便快捷，本书还在书的末尾给出了函数名称索引。

近年来随着 Internet 的普及，Unix 操作系统日趋流行，在 Unix 平台上编写应用程序的需求也越来越大。为了满足读者的需要，本书在编写时收录了 GNU 函数库中的全部内容，相信完全能够作为 Unix 开发人员的参考工具使用。

本书第 1 章内容为 Unix 的 C 函数库 GNU C，其中包括错误报告、内存分配、字符处理、字符串和数组工具、流输入/输出、底层输入/输出、文件系统接口、管道和队列、套接口、底层终端接口、数学函数、初等数学函数、搜索和排序、匹配、日期和时间、扩展字符集、本地和国际化、非本地退出、信号处理、进程的启动和终止、进程、任务控制、用户和群组、系统信息、系统配置参数等。第 2 章为 Borland 公司的 Turbo C 函数库，其中包括 Turbo C 中 21 个头文件的所有函数说明。第 3 章为 Microsoft 公司的 MSC 中 20 个头文件中的所有函数说明。

本书第 1 章由张翔编写，第 2 章由裘岚编写，第 3 章由张晓芸编写。此外参与本书编写和整理工作的还有吴世祥、许进等同志。

由于编者的水平有限，书中可能存在不当之处，请读者提出宝贵意见。

作　者

2002 年 3 月

目　录

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 errnum 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 errnum)

〖位置〗string.h (ISO)

〖说明〗errnum 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 errnum, 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进行分配检查，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_1grow (obstack, *ptr++);
          len--;
        }
      else
        {
          if (room > len)
            room = len;
          /* Add fast as much as we have room for. */
          len -= room;
          while (room-- > 0)
            obstack_1grow_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空间，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。