SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

THE C PROGRAMMING LANGUAGE

Second Edition

Brian W. Kernighan • Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Kernighan, Brian W.
The C programming language.
Includes index.
1. C (Computer program language) I. Ritchie, Dennis M. II. Title.
QA76.73.C15K47 1988 005.13'3 88-5934
ISBN 0-13-110370-9
ISBN 0-13-110362-8 (pbk.)

This book was typeset (`pic|tbl|eqn|troff -ms`) in Times Roman and Courier by the authors, using an Autologic APS-5 phototypesetter and a DEC VAX 8550 running the 9th Edition of the UNIX® operating system.

Prentice Hall Software Series
Brian Kernighan, Advisor

Printed in the United States of America

10 9 8 7 6

ISBN 0-13-110362-8 {PBK}
ISBN 0-13-110370-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

7.8.4 Command Execution

The function `system(char *s)` executes the command contained in the character string `s`, then resumes execution of the current program. The contents of `s` depend strongly on the local operating system. As a trivial example, on UNIX systems, the statement

```
system("date");
```

causes the program `date` to be run; it prints the date and time of day on the standard output. `system` returns a system-dependent integer status from the command executed. In the UNIX system, the status return is the value returned by `exit`.

7.8.5 Storage Management

The functions `malloc` and `calloc` obtain blocks of memory dynamically.

```
void *malloc(size_t n)
```

returns a pointer to `n` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied.

```
void *calloc(size_t n, size_t size)
```

returns a pointer to enough space for an array of `n` objects of the specified size, or `NULL` if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by `malloc` or `calloc` has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

```
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

`free(p)` frees the space pointed to by `p`, where `p` was originally obtained by a call to `malloc` or `calloc`. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling `calloc` or `malloc`.

It is also an error to use something after it has been freed. A typical but incorrect piece of code is this loop that frees items from a list:

```
for (p = head; p != NULL; p = p->next)    /* WRONG */
    free(p);
```

The right way is to save whatever is needed before freeing:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

Section 8.7 shows the implementation of a storage allocator like `malloc`, in

which allocated blocks may be freed in any order.

7.8.6 Mathematical Functions

There are more than twenty mathematical functions declared in `<math.h>`; here are some of the more frequently used. Each takes one or two `double` arguments and returns a `double`.

`sin(x)`	sine of x, x in radians
`cos(x)`	cosine of x, x in radians
`atan2(y,x)`	arctangent of y/x, in radians
`exp(x)`	exponential function e^x
`log(x)`	natural (base e) logarithm of x ($x>0$)
`log10(x)`	common (base 10) logarithm of x ($x>0$)
`pow(x,y)`	x^y
`sqrt(x)`	square root of x ($x \geqslant 0$)
`fabs(x)`	absolute value of x

7.8.7 Random Number Generation

The function `rand()` computes a sequence of pseudo-random integers in the range zero to `RAND_MAX`, which is defined in `<stdlib.h>`. One way to produce random floating-point numbers greater than or equal to zero but less than one is

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(If your library already provides a function for floating-point random numbers, it is likely to have better statistical properties than this one.)

The function `srand(unsigned)` sets the seed for `rand`. The portable implementation of `rand` and `srand` suggested by the standard appears in Section 2.7.

Exercise 7-9. Functions like `isupper` can be implemented to save space or to save time. Explore both possibilities. □

`sin(x)`	sine of x
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`asin(x)`	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.
`acos(x)`	$\cos^{-1}(x)$ in range $[0, \pi]$, $x \in [-1, 1]$.
`atan(x)`	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$.
`atan2(y,x)`	$\tan^{-1}(y/x)$ in range $[-\pi, \pi]$.
`sinh(x)`	hyperbolic sine of x
`cosh(x)`	hyperbolic cosine of x
`tanh(x)`	hyperbolic tangent of x
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm $\ln(x)$, $x>0$.
`log10(x)`	base 10 logarithm $\log_{10}(x)$, $x>0$.
`pow(x,y)`	x^y. A domain error occurs if `x`=0 and `y`$\leq$0, or if `x`$<$0 and `y` is not an integer.
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$.
`ceil(x)`	smallest integer not less than `x`, as a `double`.
`floor(x)`	largest integer not greater than `x`, as a `double`.
`fabs(x)`	absolute value $\lvert x \rvert$
`ldexp(x,n)`	$x \cdot 2^n$
`frexp(x, int *exp)`	splits `x` into a normalized fraction in the interval [1/2, 1), which is returned, and a power of 2, which is stored in `*exp`. If `x` is zero, both parts of the result are zero.
`modf(x, double *ip)`	splits `x` into integral and fractional parts, each with the same sign as `x`. It stores the integral part in `*ip`, and returns the fractional part.
`fmod(x,y)`	floating-point remainder of `x/y`, with the same sign as `x`. If `y` is zero, the result is implementation-defined.

B5. Utility Functions: <stdlib.h>

The header <stdlib.h> declares functions for number conversion, storage allocation, and similar tasks.

`double atof(const char *s)`
`atof` converts `s` to `double`; it is equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`
converts `s` to `int`; it is equivalent to `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`
converts `s` to `long`; it is equivalent to `strtol(s, (char**)NULL, 10)`.

`double strtod(const char *s, char **endp)`
`strtod` converts the prefix of `s` to `double`, ignoring leading white space; it stores a pointer to any unconverted suffix in `*endp` unless `endp` is `NULL`. If the answer

would overflow, HUGE_VAL is returned with the proper sign; if the answer would underflow, zero is returned. In either case errno is set to ERANGE.

`long strtol(const char *s, char **endp, int base)`
strtol converts the prefix of s to long, ignoring leading white space; it stores a pointer to any unconverted suffix in *endp unless endp is NULL. If base is between 2 and 36, conversion is done assuming that the input is written in that base. If base is zero, the base is 8, 10, or 16; leading 0 implies octal and leading 0x or 0X hexadecimal. Letters in either case represent digits from 10 to base-1; a leading 0x or 0X is permitted in base 16. If the answer would overflow, LONG_MAX or LONG_MIN is returned, depending on the sign of the result, and errno is set to ERANGE.

`unsigned long strtoul(const char *s, char **endp, int base)`
strtoul is the same as strtol except that the result is unsigned long and the error value is ULONG_MAX.

`int rand(void)`
rand returns a pseudo-random integer in the range 0 to RAND_MAX, which is at least 32767.

`void srand(unsigned int seed)`
srand uses seed as the seed for a new sequence of pseudo-random numbers. The initial seed is 1.

`void *calloc(size_t nobj, size_t size)`
calloc returns a pointer to space for an array of nobj objects, each of size size, or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

`void *malloc(size_t size)`
malloc returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied. The space is uninitialized.

`void *realloc(void *p, size_t size)`
realloc changes the size of the object pointed to by p to size. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

`void free(void *p)`
free deallocates the space pointed to by p; it does nothing if p is NULL. p must be a pointer to space previously allocated by calloc, malloc, or realloc.

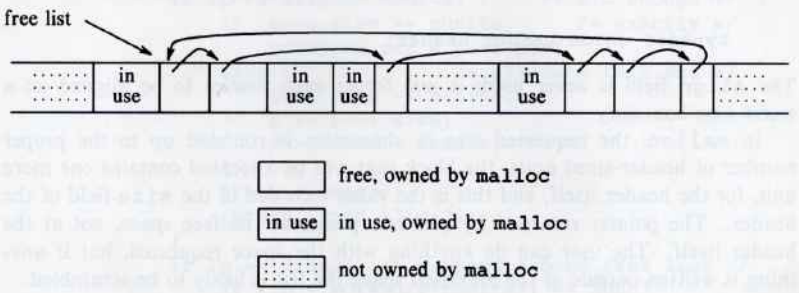
`void abort(void)`
abort causes the program to terminate abnormally, as if by raise(SIGABRT).

`void exit(int status)`
exit causes normal program termination. atexit functions are called in reverse order of registration, open files are flushed, open streams are closed, and control is returned to the environment. How status is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values EXIT_SUCCESS and EXIT_FAILURE may also be used.

8.7 Example—A Storage Allocator

In Chapter 5, we presented a very limited stack-oriented storage allocator. The version that we will now write is unrestricted. Calls to `malloc` and `free` may occur in any order; `malloc` calls upon the operating system to obtain more memory as necessary. These routines illustrate some of the considerations involved in writing machine-dependent code in a relatively machine-independent way, and also show a real-life application of structures, unions and `typedef`.

Rather than allocating from a compiled-in fixed-sized array, `malloc` will request space from the operating system as needed. Since other activities in the program may also request space without calling this allocator, the space that `malloc` manages may not be contiguous. Thus its free storage is kept as a list of free blocks. Each block contains a size, a pointer to the next block, and the space itself. The blocks are kept in order of increasing storage address, and the last block (highest address) points to the first.



When a request is made, the free list is scanned until a big-enough block is found. This algorithm is called "first fit," by contrast with "best fit," which looks for the smallest block that will satisfy the request. If the block is exactly the size requested it is unlinked from the list and returned to the user. If the block is too big, it is split, and the proper amount is returned to the user while the residue remains on the free list. If no big-enough block is found, another large chunk is obtained from the operating system and linked into the free list.

Freeing also causes a search of the free list, to find the proper place to insert the block being freed. If the block being freed is adjacent to a free block on either side, it is coalesced with it into a single bigger block, so storage does not become too fragmented. Determining adjacency is easy because the free list is maintained in order of increasing address.

One problem, which we alluded to in Chapter 5, is to ensure that the storage returned by `malloc` is aligned properly for the objects that will be stored in it. Although machines vary, for each machine there is a most restrictive type: if the most restrictive type can be stored at a particular address, all other types may be also. On some machines, the most restrictive type is a `double`; on others, `int` or `long` suffices.

A free block contains a pointer to the next block in the chain, a record of the size of the block, and then the free space itself; the control information at the beginning is called the "header." To simplify alignment, all blocks are multiples of the header size, and the header is aligned properly. This is achieved by a union that contains the desired header structure and an instance of the most restrictive alignment type, which we have arbitrarily made a `long`:

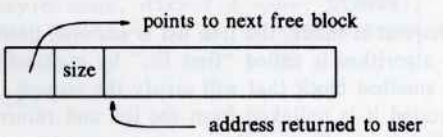
```
typedef long Align;  /* for alignment to long boundary */

union header {       /* block header: */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;         /* force alignment of blocks */
};

typedef union header Header;
```

The `Align` field is never used; it just forces each header to be aligned on a worst-case boundary.

In `malloc`, the requested size in characters is rounded up to the proper number of header-sized units; the block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the `size` field of the header. The pointer returned by `malloc` points at the free space, not at the header itself. The user can do anything with the space requested, but if anything is written outside of the allocated space the list is likely to be scrambled.



A block returned by `malloc`

The size field is necessary because the blocks controlled by `malloc` need not be contiguous—it is not possible to compute sizes by pointer arithmetic.

The variable `base` is used to get started. If `freep` is `NULL`, as it is at the first call of `malloc`, then a degenerate free list is created; it contains one block of size zero, and points to itself. In any case, the free list is then searched. The search for a free block of adequate size begins at the point (`freep`) where the last block was found; this strategy helps keep the list homogeneous. If a too-big block is found, the tail end is returned to the user; in this way the header of the original needs only to have its size adjusted. In all cases, the pointer returned to the user points to the free space within the block, which begins one unit beyond the header.

```
static Header base;          /* empty list to get started */
static Header *freep = NULL;     /* start of free list */

/* malloc:  general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {  /* no free list yet */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {    /* big enough */
            if (p->s.size == nunits)      /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {              /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)  /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* none left */
    }
}
```

The function `morecore` obtains storage from the operating system. The details of how it does this vary from system to system. Since asking the system for memory is a comparatively expensive operation, we don't want to do that on every call to `malloc`, so `morecore` requests at least `NALLOC` units; this larger block will be chopped up as needed. After setting the size field, `morecore` inserts the additional memory into the arena by calling `free`.

The UNIX system call `sbrk(n)` returns a pointer to `n` more bytes of storage. `sbrk` returns `-1` if there was no space, even though `NULL` would have been a better design. The `-1` must be cast to `char *` so it can be compared with the return value. Again, casts make the function relatively immune to the details of pointer representation on different machines. There is still one assumption, however, that pointers to different blocks returned by `sbrk` can be meaningfully compared. This is not guaranteed by the standard, which permits pointer comparisons only within an array. Thus this version of `malloc` is portable only among machines for which general pointer comparison is meaningful.

```
#define NALLOC  1024    /* minimum #units to request */

/* morecore:  ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)  /* no space at all */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

`free` itself is the last thing. It scans the free list, starting at `freep`, looking for the place to insert the free block. This is either between two existing blocks or at one end of the list. In any case, if the block being freed is adjacent to either neighbor, the adjacent blocks are combined. The only troubles are keeping the pointers pointing to the right things and the sizes correct.

```
/* free:  put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;    /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;  /* freed block at start or end of arena */

    if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {          /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Although storage allocation is intrinsically machine-dependent, the code above illustrates how the machine dependencies can be controlled and confined to a very small part of the program. The use of `typedef` and `union` handles

alignment (given that `sbrk` supplies an appropriate pointer). Casts arrange that pointer conversions are made explicit, and even cope with a badly-designed system interface. Even though the details here are related to storage allocation, the general approach is applicable to other situations as well.

Exercise 8-6. The standard library function `calloc(n,size)` returns a pointer to `n` objects of size `size`, with the storage initialized to zero. Write `calloc`, by calling `malloc` or by modifying it. □

Exercise 8-7. `malloc` accepts a size request without checking its plausibility; `free` believes that the block it is asked to free contains a valid size field. Improve these routines so they take more pains with error checking. □

Exercise 8-8. Write a routine `bfree(p,n)` that will free an arbitrary block `p` of `n` characters into the free list maintained by `malloc` and `free`. By using `bfree`, a user can add a static or external array to the free list at any time. □