Computer Architecture

Paul Mellies

Lecture 8 : Programming in C

E. the Unix interface

```
/* getchar: simple buffered version */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0){ /* buffer is empty */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

File descriptors in UNIX

File descriptors in UNIX

A homogeneous interface :
in the UNIX operating system, all input and output is done by writing a file.

In fact, all peripheral devices are treated as files in the file system !

Before you read or write a file, you must inform the system of your intent to do so : this process is called opening the file.

The system checks your right to do so, and when everything is fine, returns a small non-negative number called the file descriptor.

File descriptors in UNIX

Because input and output involving keyboard and screen are so common, three files are opened in UNIX whenever the shell runs a program :

- the file descriptor 0 for the standard input
- the file descriptor 1 for the standard output
- the file descriptor 2 for the standard error

So, if a program reads 0 and writes 1 or 2, it does not need to open the file.

The user of a program can redirect I/O to and from files by using < and > :

```
myprog <infile >outfile
```

Low level I / O

Input and output use the `read` and the `write` system calls which are accessed from C programs through the `read` and the `write` functions.

- first argument : a file descriptor
- second argument : a character string
- third argument : number of characters requested

```
read(int fd, char *buf, int n);
write(int fd, char *buf, int n);
```

The functions `read` and `write` return the number of bytes which the command was able to read or to write in the file.

In particular, the `read` function

- returns -1 when there is an error
- returns 0 when EOF is reached

while the `write` function

- returns an integer strictly less than `n` in case of an error

Low level I / O Read and Write

```
/* buffered copy from input to output */

main(){
  char buf[BUFSIZ];
  int n;

  while ((n = read(0,buf,BUFSIZ)) > 0)
    write(1, buf, n);

  return 0;
}
```

Kernighan and Ritchie, section 8.2

Here, the parameter `BUFSIZ` is chosen according to the system at hand.

An unbuffered implementation of `getchar`

```
#include <stdio.h>
#undef getchar

/* getchar: unbuffered single character input */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) ? (unsigned char) c : EOF;
}
```

Kernighan and Ritchie, section 8.2

The function returns an unsigned char when one byte has been read in the standard input and the integer EOF = -1 otherwise.

A buffered implementation of `getchar`

```
/* getchar: simple buffered version */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0){ /* buffer is empty */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Kernighan and Ritchie, section 8.2

The static declaration of the internal variables `buf` , `char` and `n` ensure that they remain in existence rather than coming and going each time the function `getchar` is called and returns. In particular, the static variables retain their values and thus have the same value next time the function is called.

Exercise

Can you explain how this implementation of `getchar` works ?

By the way, can you also guess how the `sizeof` function computes the size of the array `buf` of characters ?

File pointers in C

File pointers

An important point to know :

Files in the standard C library are described by **file pointers** rather than by **file descriptors.**

A file pointer is a **pointer to a structure** that contains several pieces of information about the file :

- a pointer to a buffer, so the file can be read in large chunks
- a count of the number of characters left in the buffer
- a pointer to the next character position in the buffer
- the file descriptor itself
- and flags describing read/write mode, error status, etc...

The data structure that describes a file is contained in `<stdio.h>`

The structure type `FILE`

```
typedef struct _iobuf {
int  cnt;       /* characters left */
char *ptr;      /* next character position */
char *base;     /* location of buffer */
int  flag;      /* mode of file access */
int  fd;        /* file descriptor */
} FILE;
```

Kernighan and Ritchie, Chapter 8.5, fragment of stdio.h

Creates a new structure type called « `FILE` » with five fields :

`.cnt`	number of characters left in the buffer
`.ptr`	pointer to the next character position in the buffer
`.base`	pointer to the location of the buffer
`.flag`	integer equal to the mode of file access
`.fd`	integer equal to the file descriptor

open

```
#include <fcntl.h>

int filedescr;
int open(char *name, int flags, int perms);

filedescr = open(name, flags, int perms);
```

- The `name` argument is a character string containing the file name.
- The `flags` argument is an integer which specifies how the file is to be opened :
  - O_RDONLY open for reading only
  - O_WRONLY open for writing only
  - O_RDWR open for both reading and writing
- The `perms` argument will always be zero in the cases discussed here.

`open` is like `fopen` except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

`open` also returns the value -1 in case an error occurred.

fopen

```
#include <stdio.h>

FILE *fileptr;
FILE *fopen(char *name, char *mode);

fileptr = fopen(name, mode);
```

- The `name` argument is a character string containing the file name.
- The `mode` argument is a character string which specifies how the file is to be opened :
  - `"r"` open for reading only
  - `"w"` open for writing only
  - `"rw"` open for both reading and writing
  - `"a"` open for appending

`fopen` is like `open` except that instead of returning a file descriptor, it returns a file pointer.

`fopen` returns the value `NULL` in case an error occurred.

Illustration :

an implementation of `getc`

An implementation of `getc`

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20  /* max #files open at once */
```

Kernighan and Ritchie, Chapter 8.5, fragment of stdio.h

Defines four major constants of C.

An implementation of `getc`

```
extern FILE _iob[OPEN_MAX];

#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])
```

Kernighan and Ritchie, Chapter 8.5, fragment of stdio.h

Defines the three standard file pointers :

- `stdin`
- `stdout`
- `stderr`

as the addresses of the first, second and third elements
of the array `_iob` of `OPEN_MAX` elements of structure type `FILE` .

An implementation of `getc`

```
enum _flags {
    _READ   = 01,   /* file open for reading */
    _WRITE  = 02,   /* file open for writing */
    _UNBUF  = 04,   /* file is unbuffered */
    _EOF    = 010,  /* EOF has occurred on this file */
    _ERR    = 020   /* error occurred on this file */
};

#define feof(p)     (((p)->flag & _EOF) != 0)
#define ferror(p)   (((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)
```

Kernighan and Ritchie, Chapter 8.5, fragment of the header file stdio.h

Test the field `flag` of the structure `FILE` with « masks » like `_EOF` or `_ERR` to know the current status of the file.

An implementation of `getc`

```
int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define getc(p)
   (--(p)->cnt >= 0 ? (unsigned char) *(p)->ptr++ : _fillbuf(p))

#define putc(x,p)
   (--(p)->cnt >= 0 ? *(p)->ptr++ = (x) : _flushbuf((x),p))
```

Kernighan and Ritchie, Chapter 8.5, fragment of the header file stdio.h

Exercise

1. Can you parse the three expressions below

```
--(p)->cnt >= 0
(unsigned char) *(p)->ptr++
*(p)->ptr++ = (x)
```

and tell what they are doing ?

2. Then, can you describe what the functions `getc` and `putc` do ?
In particular, can you guess the purpose of the two functions

```
int _fillbuf(FILE *);
int _flushbuf(int, FILE *);
```

Solution : the `getc` function

```
int _fillbuf(FILE *);

#define getc(p)
   (--(p)->cnt >= 0 ? (unsigned char) *(((p)->ptr)++) : _fillbuf(p))
```

The `getc` function normally :

- decrements the count,
- advances the pointer
- and returns the character.

If the count gets negative, `getc` calls the function `_fillbuf` to replenish the buffer, to re-initialize the structure contents, and to return a character.

Solution : the `putc` function

```
int _flushbuf(int, FILE *);

#define putc(x,p)
   (--(p)->cnt >= 0 ? *(((p)->ptr)++) = (x) : _flushbuf((x),p))
```

The `putc` function normally :

- decrements the count,
- stores the character `x` at the position of the pointer
- then advances the pointer

If the count gets negative, `putc` calls the function `_flushbuf` which flushes the buffer into the file pointed by the pointer `p`.
The `FILE` structure content of is then re-initialized,
with the character `x` as first element of the buffer.

fopen

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666 /* RW for owner, group, others */

/* fopen: open file, return file ptr */
FILE *fopen(char *name, char *mode)
{
  int fd;
  FILE *fp;

  if (*mode != 'r' && *mode != 'w' && *mode != 'a')
      return NULL;
  for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
      if ((fp->flag & (_READ | _WRITE)) == 0)
          break;      /* found free slot */
      if (fp >= _iob + OPEN_MAX)   /* no free slots */
          return NULL;

  if (*mode == 'w')
      fd = creat(name, PERMS);
  else if (*mode == 'a') {
      if ((fd = open(name, O_WRONLY, 0)) == -1)
          fd = creat(name, PERMS);
      lseek(fd, 0L, 2);  /* call lseek with parameters:
                            /* file descriptor = fd, offset = long 0
                            /* origin = 2 = end of file
  } else
      fd = open(name, O_RDONLY, 0);
  if (fd == -1)              /* could not access name */
      return NULL;
  fp->fd = fd;
  fp->cnt = 0;
  fp->base = NULL;
  fp->flag = (*mode == 'r') ? _READ : _WRITE;
  return fp;
}
```

Kernighan and Ritchie, Chapter 8.5, fragment of the header file stdio.h

_fillbuf

```
#include "syscalls.h"

/* _fillbuf: allocate and fill input buffer */
int _fillbuf(FILE *fp){

  int bufsize;

  if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
      return EOF;
  bufsize = (fp->flg & _UNBUF) ? 1 : BUFSIZ;
  if (fp->base == NULL)       /* no buffer yet */
    /* allocate the buffer of size bufsize */
    if ((fp->base = (char *) malloc(bufsize)) == NULL)
        return EOF;           /* cannot get buffer */
  fp->ptr = fp->base;
  /* call the read function */
  /* its output is -1 for ERR, 0 for EOF */
  /* or the number 0 < n <= bufsize of read characters */
  fp->cnt = read(fp->fd, fp->ptr, bufsize);
  /* decrement fp->cnt */
  if (--fp->cnt < 0) {      /* branch if EOF or ERR */
    if (fp->cnt == -1)      /* case of EOF  */
        fp->flag |= _EOF;
    else                    /* case of ERR  */
        fp->flag |= _ERR;
    fp->cnt = 0;            /* fp->cnt reinitialized to 0 */
    return EOF;
  }
  return (unsigned char) *fp->ptr++;
}
```

Kernighan and Ritchie, Chapter 8.5, fragment of the header file stdio.h

Definition of `getchar` and `putchar`

```
#define getchar() getc(stdin)
#define putchar(x) putc((x), stdout)
```

Kernighan and Ritchie, Chapter 8.5, fragment of stdio.h

Unions

The unions

```
union int_or_double {    // declare the union type
  int integer;           // fields
  double with_dot;
};

int main() {
  union int_or_double u;

  u.integer = 42;              // access to u as a variable
  printf("%d",u.integer);      // of type int

  u.with_dot= 3.14;            // access to u as a variable
  printf("%d",u.with_dot);     // of type double
  return 0;
}
```

Unions

```
union int_or_double {    // declare the union type
  int integer;           // fields
  double with_dot;
};
```

is a union type declaration which creates a new type
without allocating any variable.

At a given time, a variable of that type will contain a value
of type `int` or of type `double` (but not the two at the same time)

The choice of a name of a field like `integer` or `with_dot`
specifies the way one wants to read and/or write in the union.

Unions

```
union int_or_double u;
```

allocation in memory of a sufficiently large block in order to store a value of type `int` or of type `double`.

its size is the size of the type of the field of the largest size.

Since `sizeof(int) < sizeof(double)`
the block has the size of a `double`.

```
u.integer = 42;
u.with_dot = 3.14;
```

store an `int` in `u`
store a `double`

Unions

The properties of unions and structures are the same: copy by assignment, call by value, returned value.

It is possible to initialise a union, but only with a value of its first field:

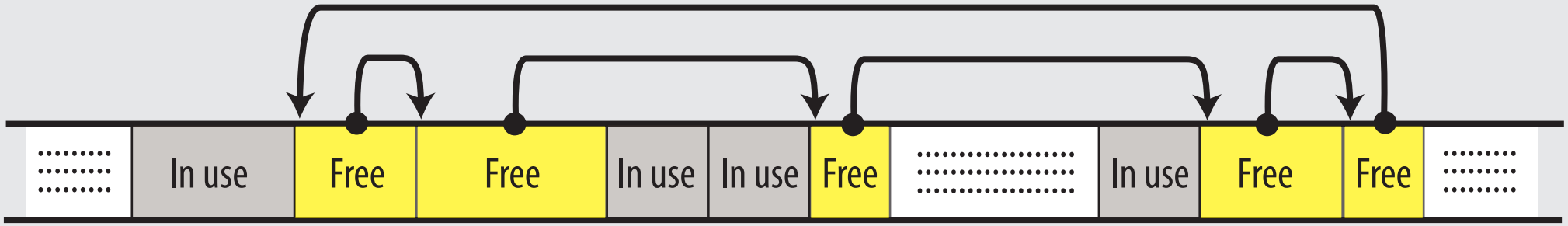
```
union int_or_double u = {42};
```

If one tries to read `u` using `u.with_dot` when the variable contains an `int` then the result is undefined. Similarly if one tries to read `u` using `u.integer` when the variable contains a `double`.

It is thus the programmer who should remember the type of the currently stored variable in the union.

A storage allocator

The `malloc` algorithm

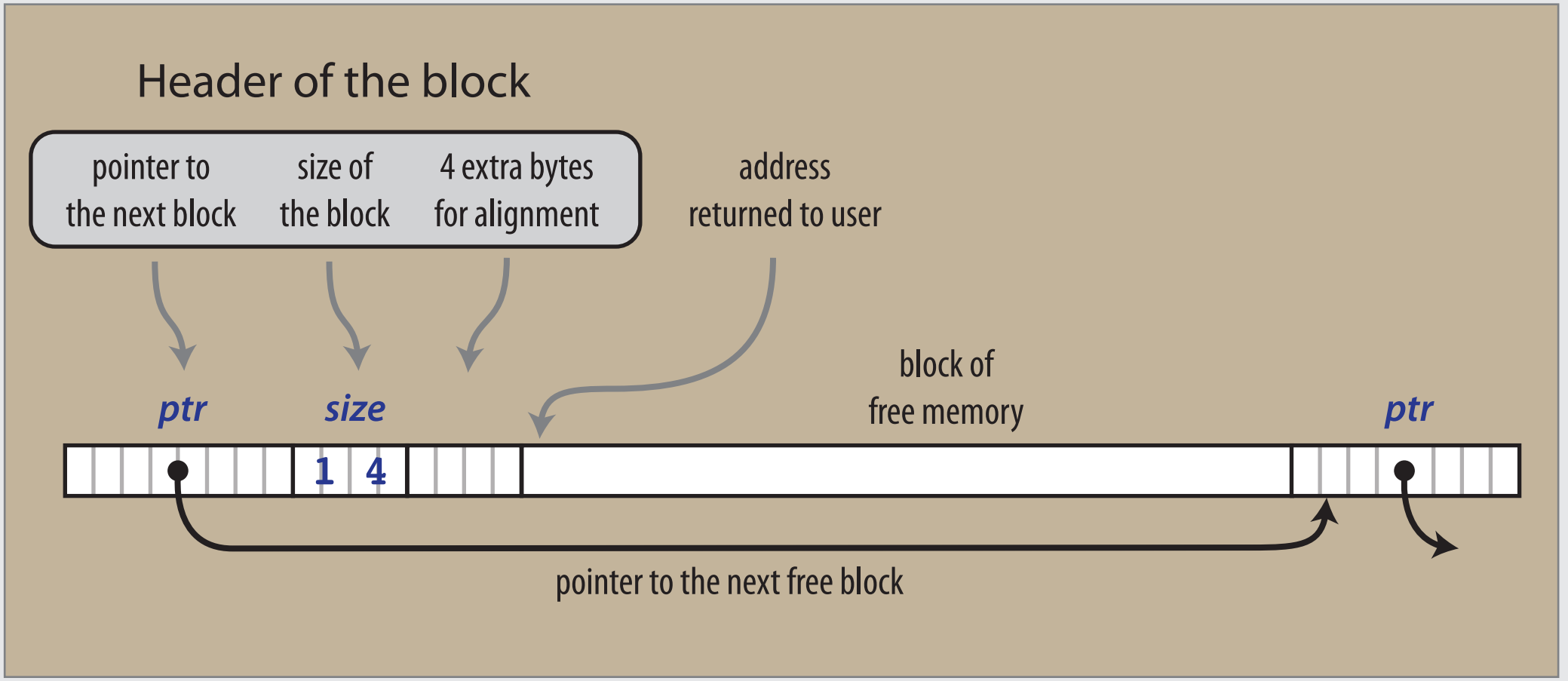


Free — Free, owned by `malloc`

In use — In use, owned by `malloc`

Not owned by `malloc`

A block returned by `malloc`



This is the reason why the `free` function does not need to indicate the size of the block of memory it wishes to free, only its pointer.

The `Header` union type

```
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;   /* this part of the union is never used */
               /* only here to force alignment of blocks */

typdef union header Header;    /* Header means union header
```

Kernighan and Ritchie, Chapter 8.5, fragment of the header file stdio.h

The `Align` part of the union type is never used :
it is simply here to enforce that every header (of type `Header`)
has just the same size as the most restrictive alignment type.

malloc

```
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;

     /* compute the number of units required */
     /* +1 is for the header */
     nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
     if ((prevp = freep) == NULL) {
     /* no free list yet : first call of malloc */
     /* use the address of base to get started  */
     /* note that base.s.ptr points to itself   */
     base.s.ptr = freep = prevp = &base;
     base.s.size = 0;
     }
     for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
     /* search for a free block of adequate size */
          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 */
     }
}
```

Kernighan and Ritchie, Chapter 8.7, fragment of the header file stdio.h

Implementation of `morecore`

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

/* morecore: ask system for more memory */
{
    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;
}
```

Kernighan and Ritchie, Chapter 8.7, fragment of the header file stdio.h

Implementation of `free`

```
/* 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 number */
        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 number */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
     freep = p;
}
```

Kernighan and Ritchie, Chapter 8.7, fragment of the header file stdio.h

Implementation of `free`

```
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 */
```

As noted by Kernighan and Ritchie, this specific implementation of `free` requires the ability to compare pointers in C.

Here, `freep` is the pointer to the beginning of the free list.

In the for loop, the pointer `p` starts from the beginning of the list and explores each element one after the other thanks to the increment instruction `p = p->s.ptr`.

The for loop stops when one of the following three cases occurs :

[1]	`p < bp < p->s.ptr`	middle of the list
[2]	`p->s.str =< p < bp`	end of the list
[3]	`bp < p->s.ptr =< p`	beginning of the list

Appendix

The precedence rule for understanding declarations in C

A. Declarations are read by starting with the name and then reading in precedence order.

B. The precedence, from high to low, is :
   B1. Parenthesis grouping together parts of a declaration
   B2. The postfix operators :
      - Parenthesis () indicating a function
      - Square brackets [] indicating an array
   B3. The prefix operator * denoting « pointer to »

C. If a **`const`** and/or **`volatile`** keyword is next to a type specifier (eg. int or long) it applies to this type specifier.
   Otherwise, the **`const`** and/or **`volatile`** keyword applies to the pointer asterisk * on its immediate left.

Extracted from the very nice (and already classical) book by Peter van der Linden : Expert C Programming -- Deep C Secrets

Illustration : `char* const *(*next)();`

A. First, go to the variable name : « next » and note that it is directly enclosed by parentheses.

B.1 So we group it with what else is in the parentheses, to get :
« next is a pointer to... »

B. Then we go outside the parentheses, and have a choice of a prefix asterisk, or a postfix pair of parentheses.

B.2 Rule B.2 tells us the highest precedence foes to the function « parentheses at the right » so we get :
« next is a pointer to a function returning… »

B.3 Then we process the prefix * to get :
« next is a pointer to a function returning a pointer to … »

C. Finally, take « char * const » as a constant pointer to a character and get :

« next is a pointer to a function returning a pointer to a const pointer-to-char »

Extracted from the very nice (and classical) book by Peter van der Linden : Expert C Programming -- Deep C Secrets