Example

```
size_t strlen(const char *s);    /* available with #include <string.h> */

size_t (*fp)(const char *);
fp = &strlen;
printf("result = %zu\n", (*fp)("hello world"));
```

- This can be abbreviated:

```
printf("result = %zu\n", fp("hello world"));
```

- Nicer with `typedef`

```
typedef size_t (*func)(const char *);
func fp = &strlen;
```

Note Function pointers are heavily used in the real world! *E.g.*,

- pass a comparing function to a queue datastructure;
- installing signal handlers (*cf.* OS lecture, and later in this course); and
- abstractions (syscall interface, subclasses, VFS, ...).

Question Can you read `int (*(*f)(void))[2]` ?

4.7 Unscrambling C declarations

Precedence rules for reading C declarations.

1. Parentheses group parts of the declaration.
2. Read **type specifiers** as atomic tokens, *e.g.*,
   - `double`,
   - `struct foo`, or
   - `unsigned short int`.
3. The keyword `const`:
   - If *next to* a type specifier, it belongs to that, making the **value** constant.
   - Otherwise, it belongs to the asterisk to its *left*, making the **pointer** constant.
4. The **postfix** operators, being one of
   - **parentheses** `(...)` indicating a function, or
   - **brackets** `[...]` indicating an array.
5. The **prefix** operator **asterisk** `*` indicating a pointer.

Note Inside parenthesis, a declaration may contain *further* declarations of function arguments! These do *not necessarily* have a name.

An algorithm for reading declarations

1. Start at the leftmost identifier that is *not* a type specifier. That is being declared.
2. Do not leave parenthesis while:
   2.1 Handle the **postfix** operators, *i.e.*, optional `(...)` or `[...]` to the **right**, do so from left to right.
      - For a function, apply the whole algorithm to each parameter.
      - For an array, optionally note the size.

   2.2 Handle the **prefix** operators `*` to the **left**, do so from right to left.
3. If inside parenthesis, leave them, and restart with 2.
4. Read the **type specifier** on the left.

tl;dr — look right, look left.

Example `int *(*list[42])(void)`

- `int *(*list[42])(void)` `list` is...
- `int *(*list[42])(void)` ...an array of 42...
- `int *(*list[42])(void)` ...pointers to

Leaving parenthesis, we're done with them. Goto step 2 of algorithm:

- `int *(*list[42])(void)` ...function of ...
- `int *(*list[42])(void)` ...no arguments...
- `int *(*list[42])(void)` ...returning a pointer to...
- `int *(*list[42])(void)` ...an integer.

Example `int (*f)(const char *s)`

- `int (*f)(const char *s)` `f` is...
- `int (*f)(const char *s)` ...a pointer to...
- `int (*f)(const char *s)` ...a function of **(**...
- `int (*f)(const char *s)` ...`s`, which is...
- `int (*f)(const char *s)` ...a pointer to...
- `int (*f)(const char *s)` ...a `const`ant `char`acter **)**...
- `int (*f)(const char *s)` ...returning an integer.

Example `void f(char *x[])`

- `void f(char *x[])` `f` is a...
- `void f(char *x[])` ...function of (...
- `void f(char *x[])` ...`x`, which is...
- `void f(char *x[])` ...an array of unspecified size of...
- `void f(char *x[])` ...pointers to...
- `void f(char *x[])` ...`char`acter)...
- `void f(char *x[])` ...not returning anything.

The declaration `void f(char **x)` is equivalent, specifying array dimensions does not make any sense in this case (*cf.* page 79).

Example `void *f(char *(*p)[5])`

- `void *f(char *(*p)[5])` `f` is a…
- `void *f(char *(*p)[5])` …function of **(**…
- `void *f(char *(*p)[5])` …`p`, which is…
- `void *f(char *(*p)[5])` …a pointer to…
- `void *f(char *(*p)[5])` …an array of five…
- `void *f(char *(*p)[5])` …pointers to…
- `void *f(char *(*p)[5])` …character **)**…
- `void *f(char *(*p)[5])` …returning a pointer to…
- `void *f(char *(*p)[5])` …data of unspecified type.

In this case, specifying the array dimensions makes sense: In the body of `f`, `sizeof(*p)` will return 40 if the size of a pointer is 8. This also effects pointer arithmetics on `p`.

Note Function parameters need not be named in a **declaration**!

`double (*f)(double x)` ≡ `double (*f)(double)`

This makes it occasionally hard to find out what is being declared.

Example `int f(char *[])` (Example from page 136)

- `int f(char *[])` f is a…
- `int f(char *[])` …function of **(**…

No identifier: So *"it"* is to the right of all `*`, and to the left of all `(...)` and `[...]`.

- `int f(char *[])` …an array of…
- `int f(char *[])` …pointers to…
- `int f(char *[])` …character **)**…
- `int f(char *[])` …returning an integer.

This is actually equivalent to `int f(char **)`.

Question What is this: `int f(char (*)[23])` ?

Easily spotted mistakes

Some observations about **parentheses** in declarations (Note: ... is a meta-placeholder!):

- **Invalid types**, *i.e.*, in a type declaration, you will **never** see

  `foo(...)(...)` Functions cannot return functions.

  `foo(...)[...]` Functions cannot return arrays.

  `foo[...](...)` Arrays cannot contain functions.

- **Valid types**

  `int bar[...][...];` `bar` is an array of arrays.

  `int (*fun(...))(...);` Function `fun` returns a *pointer to* a function.

  `int (*foo(...))[];` Function `foo` returns a *pointer to* an array.

  `int (*arr[...])(...);` `arr` is an array of *pointers to* functions.

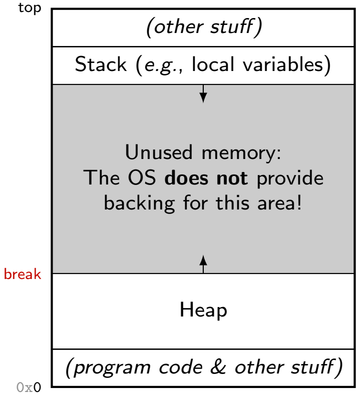
5
Excursus: Inside Malloc

This excursus demonstrates:

- How to impose a meaning on a region of memory.
- Heavy use of pointer arithmetics.
- Glimpse under the hood of memory allocation.

5.1 Process memory layout

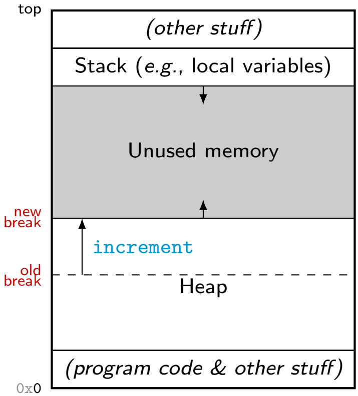
The program's view of its memory: **virtual RAM**.



(simplified picture)

- The stack contains the variables local to a function call. Grows downwards.
- The **program break** is the first location after the program's *data segment* (*cf.* later).
- Incrementing the break is **allocating heap memory**: Ask the OS to provide backing for the increased consumption of memory.

Allocating heap memory



- `sbrk`(2) can **move the break**[23].

```
void *sbrk(intptr_t increment);
```

- Returns address of old break, or `(void *)-1` on error.
- If the break was *increased*, then the returned value is a pointer to **newly allocated** memory, backed by the OS!
- (There are other system calls to get memory from the OS, *cf.* later)

Note Avoid using `brk()` and `sbrk()`: the `malloc`(3) memory allocation package is the portable and comfortable way of allocating memory.

[23] there's also `brk`(2), for the same purpose — we use `sbrk`(2) only.

5.2 Implementing a memory allocator

How to write your own malloc[25]

- We know that we can get **fresh memory** from the OS via sbrk(2).
- "The real" malloc(3) uses this, and other techniques. There are many different, *very sophisticated* implementations of memory allocators.
- We implement a **very simple** allocator.[24] Most prominently, we ignore data **alignment**.

The Interface

```
void *kr_malloc(size_t b);
void kr_free(void *ap);
```

- kr_malloc allocates b bytes and returns a pointer to the allocated memory, or NULL on error.
- kr_free frees memory pointed to by ap, which must have been returned by a previous call to kr_malloc.

[24] adapted from: Kernighan, Ritchie. The C Programming Language. Prentice Hall Software Series. Section 8.7, *A Storage Allocator*.

[25] Just because you can — in general, this is not a smart idea.

The rough plan

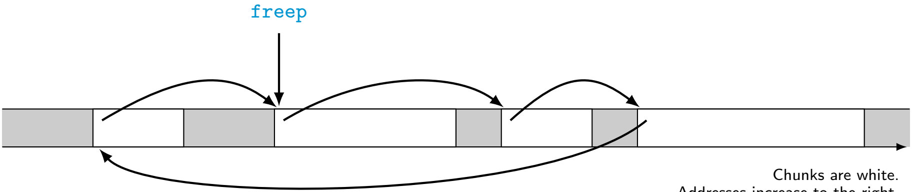
- System calls are **expensive**: Avoid using them often.
- So `kr_malloc` tries to get **a big chunk** of memory from the OS, and hands **smaller pieces** of that to the calling program.
- So we need to maintain a **list of free memory chunks**, that
  - have been allocated **from the OS** via `sbrk`(2), but
  - have **not yet** been handed to the **program**,
  - or have been **returned from** the program.
- If the program frees memory, `kr_free` adds that to the list of free memory, but does not return it to the OS.
  - Obvious weakness: Memory consumption of the process never shrinks.

Note The functions `sbrk`(2), `kr_malloc` and `kr_free` implement a concept of **transferring ownership** of memory between the OS, the allocator, and the program.

Chunks of free memory

Our allocator maintains a list of **free memory chunks**. These are not currently used by the program, *i.e.* they

- lie in memory allocated from the OS via `sbrk`,
- have not been given to the program by returning from `kr_malloc`, or have been given back to the allocator via `kr_free`.



Chunks are white.
Addresses increase to the right.

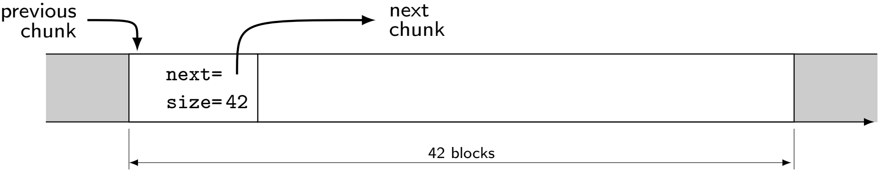
- **Circular list**: Every chunk points to the next chunk.
- List is **ordered by memory address**, with the obvious exception.
- A pointer `freep` is the **entry point** into the list. It may point to any chunk, and we will move it around quite a bit.

To maintain the list of free chunks, we install a header **at the start** of each chunk:

```
typedef struct header Header;
struct header {
    Header *next;
    size_t size;
};

#define BLOCKSIZE (sizeof(Header))
```

- `next` points to the next chunk in the circular list.
- `size` is the size of the **entire** chunk, given in the unit `BLOCKSIZE` bytes.



Questions If `Header *p` points to the header, then

- where does `p + 1` point to?
- where does `p + p->size` point to?

5.3 The `kr_malloc()` function

```
Header *freep = NULL,   /* a global pointer to the free list */
        base;           /* and a dummy for the empty list */

void *kr_malloc(size_t bytes) {

    /* number of blocks required, including one more for the header */
    size_t reqd = 1 + (bytes + BLOCKSIZE - 1) / BLOCKSIZE;
    Header *prevp = freep;  /* ptr to previous chunk */

    if (!freep) {                         /* make empty list if called for the first time */
        base.next = freep = prevp = &base;
        base.size = 0;
    }

    for (Header *p = prevp->next; ; prevp = p, p = p->next) {
```

Check the chunk `*p`. `return` a pointer if it is big enough. See the following slides.

```
        if (p == freep) {         /* if we have unsuccessfully traversed the whole list... */
            p = morecore(reqd);           /* ...get more from the OS (cf. page 154)... */
            if (p == NULL) return NULL;                               /* ...or fail. */
        }
    }
}
```

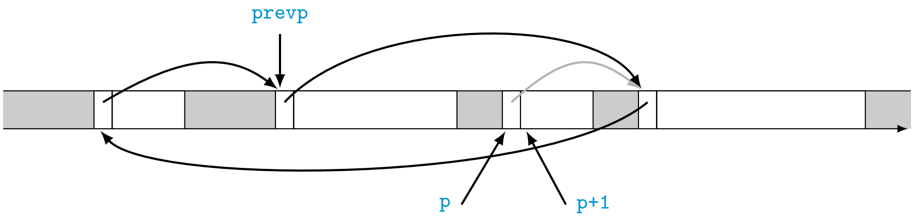
kr_malloc() — using a chunk that fits exactly

- Header *p points to current chunk, *prevp to the previous one.
- We need reqd blocks of free memory.

```
        if (p->size >= reqd) {  /* this chunk is large enough */
            if (p->size == reqd)  /* it fits exactly */
                prevp->next = p->next;  /* remove chunk from free list */
            else {
```

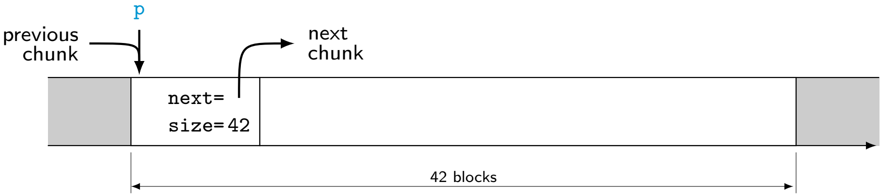
Split the chunk p points to. See the following slides.

```
            }
            freep = prevp;  /* next search continues from here */
            return p + 1;  /* memory address the program may write to */
        }
```

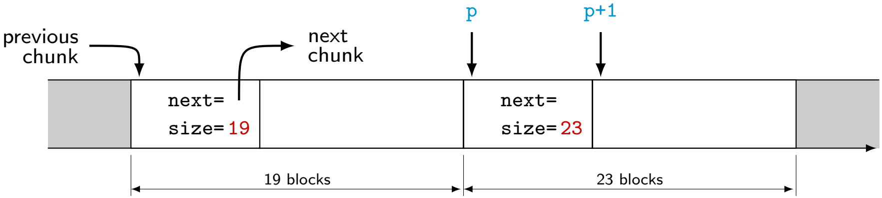


kr_malloc() — split a chunk that is too large

Assume we need reqd = 23 blocks, but the chunk has 42...




```
p->size -= reqd;
p += p->size;
p->size = reqd;             /* Enjoy: We just make up a header here! */
```

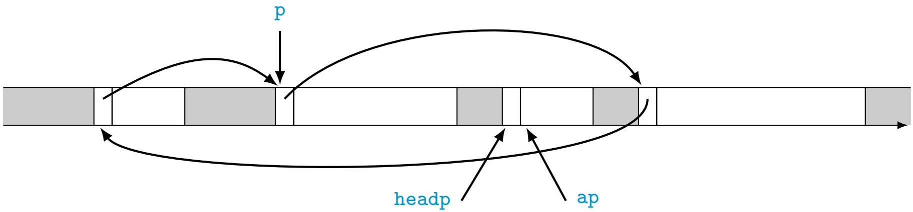


5.4 The `kr_free()` function

Where in the list should the freed chunk be linked?

```
void kr_free(void *ap) {
    Header *p,
      *headp = (Header *)ap - 1;  /* determine header of chunk *ap */

     /* Find p so, that headp belongs between p and p->next. */
    for (p = freep; !(p < headp && headp < p->next); p = p->next)
        if (p >= p->next && (p < headp || headp < p->next)) break;
```

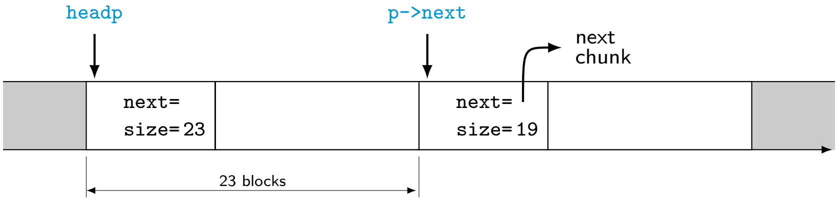


Question Now it would be easy to hook the freed chunk into the list:

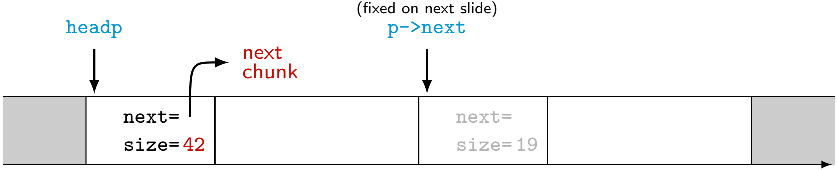
`headp->next = p->next;`   `p->next = headp;`

Why may this not be the smartest thing to do?

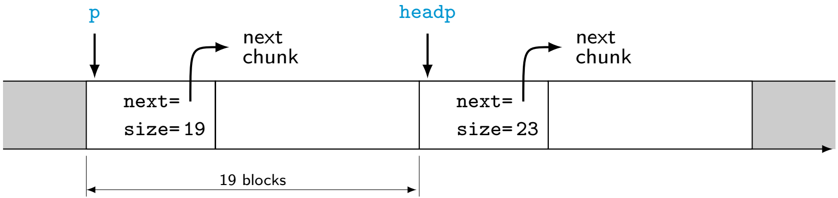
kr_free() — fuse/link with the following chunk



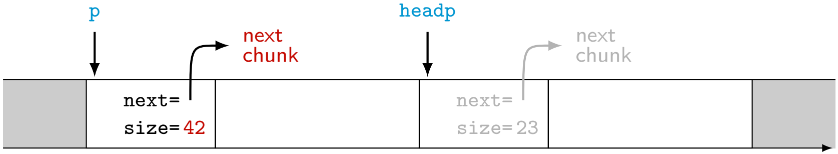

```
    /* If the chunk is adjacent to the following chunk, fuse the two into one... */
    if (headp + headp->size == p->next) {
        headp->size += p->next->size;
        headp->next = p->next->next;
    } else
        headp->next = p->next;  /* ...otherwise just link without fusing. */
```



kr_free() — fuse/link with the previous chunk




```
/* If the chunk is adjacent to the previous chunk, fuse the two into one... */
if (p + p->size == headp) {
    p->size += headp->size;
    p->next = headp->next;                  /* Exercise: Why is this required? */
} else
    p->next = headp;  /* ...otherwise just link without fusing. */
```



kr_free() — after linking into the list

```
    /* We set ‘freep‘ to point just before, or at the freed chunk.  Used by morecore. */
    freep = p;
}
```