Chapter 46: Memory management

name	description
size (`malloc`, `realloc` and `aligned_alloc`)	total size of the memory in bytes. For `aligned_alloc` the size must be a integral multiple of alignment.
size (`calloc`)	size of each element
nelements	number of elements
ptr	pointer to allocated memory previously returned by `malloc`, `calloc`, `realloc` or `aligned_alloc`
alignment	alignment of allocated memory

For managing dynamically allocated memory, the standard C library provides the functions `malloc()`, `calloc()`, `realloc()` and `free()`. In C99 and later, there is also `aligned_alloc()`. Some systems also provide `alloca()`.

Section 46.1: Allocating Memory

Standard Allocation

The C dynamic memory allocation functions are defined in the **`<stdlib.h>`** header. If one wishes to allocate memory space for an object dynamically, the following code can be used:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

This computes the number of bytes that ten `int`s occupy in memory, then requests that many bytes from `malloc` and assigns the result (i.e., the starting address of the memory chunk that was just created using `malloc`) to a pointer named `p`.

It is good practice to use `sizeof` to compute the amount of memory to request since the result of `sizeof` is implementation defined (except for *character types*, which are `char`, `signed char` and `unsigned char`, for which `sizeof` is defined to always give 1).

Because `malloc` might not be able to service the request, it might return a null pointer. It is important to check for this to prevent later attempts to dereference the null pointer.

Memory dynamically allocated using `malloc()` may be resized using `realloc()` or, when no longer needed, released using `free()`.

Alternatively, declaring `int array[10];` would allocate the same amount of memory. However, if it is declared inside a function without the keyword `static`, it will only be usable within the function it is declared in and the functions it calls (because the array will be allocated on the stack and the space will be released for reuse when the function returns). Alternatively, if it is defined with `static` inside a function, or if it is defined outside any function, then its lifetime is the lifetime of the program. Pointers can also be returned from a function, however a function in C can not return an array.

Zeroed Memory

The memory returned by `malloc` may not be initialized to a reasonable value, and care should be taken to zero the memory with `memset` or to immediately copy a suitable value into it. Alternatively, `calloc` returns a block of the

desired size where all bits are initialized to `0`. This need not be the same as the representation of floating-point zero or a null pointer constant.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

A note on `calloc`: Most (commonly used) implementations will optimise `calloc()` for performance, so it will be faster than calling `malloc()`, then `memset()`, even though the net effect is identical.

Aligned Memory

`Version ≥ C11`

C11 introduced a new function `aligned_alloc()` which allocates space with the given alignment. It can be used if the memory to be allocated is needed to be aligned at certain boundaries which can't be satisfied by `malloc()` or `calloc()`. `malloc()` and `calloc()` functions allocate memory that's suitably aligned for *any* object type (i.e. the alignment is `alignof(max_align_t)`). But with `aligned_alloc()` greater alignments can be requested.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

The C11 standard imposes two restrictions: 1) the *size* (second argument) requested must be an integral multiple of the *alignment* (first argument) and 2) the value of *alignment* should be a valid alignment supported by the implementation. Failure to meet either of them results in undefined behavior.

Section 46.2: Freeing Memory

It is possible to release dynamically allocated memory by calling free().

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

The memory pointed to by `p` is reclaimed (either by the libc implementation or by the underlying OS) after the call to `free()`, so accessing that freed memory block via `p` will lead to undefined behavior. Pointers that reference memory elements that have been freed are commonly called dangling pointers, and present a security risk. Furthermore, the C standard states that even accessing the value of a dangling pointer has undefined behavior.

Note that the pointer `p` itself can be re-purposed as shown above.

Please note that you can only call `free()` on pointers that have directly been returned from the `malloc()`, `calloc()`, `realloc()` and `aligned_alloc()` functions, or where documentation tells you the memory has been allocated that way (functions like `strdup ()` are notable examples). Freeing a pointer that is,

- obtained by using the `&` operator on a variable, or
- in the middle of an allocated block,

is forbidden. Such an error will usually not be diagnosed by your compiler but will lead the program execution in an undefined state.

There are two common strategies to prevent such instances of undefined behavior.

The first and preferable is simple - have `p` itself cease to exist when it is no longer needed, for example:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

By calling `free()` directly before the end of the containing block (i.e. the `}`), `p` itself ceases to exist. The compiler will give a compilation error on any attempt to use `p` after that.

A second approach is to also invalidate the pointer itself after releasing the memory to which it points:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Arguments for this approach:

- On many platforms, an attempt to dereference a null pointer will cause instant crash: Segmentation fault. Here, we get at least a stack trace pointing to the variable that was used after being freed.

  Without setting pointer to `NULL` we have dangling pointer. The program will very likely still crash, but later, because the memory to which the pointer points will silently be corrupted. Such bugs are difficult to trace because they can result in a call stack that completely unrelated to the initial problem.

  This approach hence follows the fail-fast concept.

- It is safe to free a null pointer. The C Standard specifies that `free(NULL)` has no effect:

  > The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not

> match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to `free` or `realloc`, the behavior is undefined.

- Sometimes the first approach cannot be used (e.g. memory is allocated in one function, and deallocated much later in a completely different function)

Section 46.3: Reallocating Memory

You may need to expand or shrink your pointer storage space after you have allocated memory to it. The `void *realloc(void *ptr, size_t size)` function deallocates the old object pointed to by `ptr` and returns a pointer to an object that has the size specified by `size`. `ptr` is the pointer to a memory block previously allocated with `malloc`, `calloc` or `realloc` (or a null pointer) to be reallocated. The maximal possible contents of the original memory is preserved. If the new size is larger, any additional memory beyond the old size are uninitialized. If the new size is shorter, the contents of the shrunken part is lost. If `ptr` is NULL, a new block is allocated and a pointer to it is returned by the function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);
```

```
    return EXIT_SUCCESS;
}
```

The reallocated object may or may not have the same address as `*p`. Therefore it is important to capture the return value from `realloc` which contains the new address if the call is successful.

Make sure you assign the return value of `realloc` to a `temporary` instead of the original `p`. `realloc` will return null in case of any failure, which would overwrite the pointer. This would lose your data and create a memory leak.

Section 46.4: realloc(ptr, 0) is not equivalent to free(ptr)

`realloc` is *conceptually equivalent* to `malloc + memcpy + free` on the other pointer.

If the size of the space requested is zero, the behavior of `realloc` is implementation-defined. This is similar for all memory allocation functions that receive a `size` parameter of value `0`. Such functions may in fact return a non-null pointer, but that must never be dereferenced.

Thus, `realloc(ptr,0)` is not equivalent to `free(ptr)`. It may

- be a "lazy" implementation and just return `ptr`
- `free(ptr)`, allocate a dummy element and return that
- `free(ptr)` and return `0`
- just return `0` for failure and do nothing else.

So in particular the latter two cases are indistinguishable by application code.

This means `realloc(ptr,0)` may not really free/deallocate the memory, and thus it should never be used as a replacement for `free`.

Section 46.5: Multidimensional arrays of variable size

`Version ≥ C99`

Since C99, C has variable length arrays, VLA, that model arrays with bounds that are only known at initialization time. While you have to be careful not to allocate too large VLA (they might smash your stack), using *pointers to VLA* and using them in `sizeof` expressions is fine.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
   size_t m = argc*8;
   double (*matrix)[m] = malloc(sizeof(double[n][m]));
   // initialize matrix somehow
   double res = sumAll(n, m, matrix);
   printf("result is %g\n", res);
   free(matrix);
}
```

Here `matrix` is a pointer to elements of type `double[m]`, and the `sizeof` expression with `double[n][m]` ensures that it contains space for `n` such elements.

All this space is allocated contiguously and can thus be deallocated by a single call to `free`.

The presence of VLA in the language also affects the possible declarations of arrays and pointers in function headers. Now, a general integer expression is permitted inside the [ ] of array parameters. For both functions the expressions in [ ] use parameters that have declared before in the parameter list. For `sumAll` these are the lengths that the user code expects for the matrix. As for all array function parameters in C the innermost dimension is rewritten to a pointer type, so this is equivalent to the declaration

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

That is, `n` is not really part of the function interface, but the information can be useful for documentation and it could also be used by bounds checking compilers to warn about out-of-bounds access.

Likwise, for `main`, the expression `argc+1` is the minimal length that the C standard prescribes for the `argv` argument.

Note that officially VLA support is optional in C11, but we know of no compiler that implements C11 and that doesn't have them. You could test with the macro `__STDC_NO_VLA__` if you must.

Section 46.6: alloca: allocate memory on stack

Caveat: `alloca` is only mentioned here for the sake of completeness. It is entirely non-portable (not covered by any of the common standards) and has a number of potentially dangerous features that make it un-safe for the unaware. Modern C code should replace it with *Variable Length Arrays* (VLA).

Manual page

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Allocate memory on the stack frame of the caller, the space referenced by the returned pointer is automatically free'd when the caller function finishes.

While this function is convenient for automatic memory management, be aware that requesting large allocation could cause a stack overflow, and that you cannot use `free` with memory allocated with `alloca` (which could cause more issue with stack overflow).

For these reason it is not recommended to use `alloca` inside a loop nor a recursive function.

And because the memory is `free`'d upon function return you cannot return the pointer as a function result (the behavior would be undefined).

Summary

- call identical to `malloc`
- automatically free'd upon function return
- incompatible with `free`,`realloc` functions (undefined behavior)
- pointer cannot be returned as a function result (undefined behavior)

- allocation size limited by stack space, which (on most machines) is a lot smaller than the heap space available for use by `malloc()`
- avoid using `alloca()` and VLAs (variable length arrays) in a single function
- `alloca()` is not as portable as `malloc()` et al

Recommendation

- Do not use `alloca()` in new code

`Version ≥ C99`

Modern alternative.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

This works where `alloca()` does, and works in places where `alloca()` doesn't (inside loops, for example). It does assume either a C99 implementation or a C11 implementation that does not define `__STDC_NO_VLA__`.

Section 46.7: User-defined memory management

`malloc()` often calls underlying operating system functions to obtain pages of memory. But there is nothing special about the function and it can be implemented in straight C by declaring a large static array and allocating from it (there is a slight difficulty in ensuring correct alignment, in practice aligning to 8 bytes is almost always adequate).

To implement a simple scheme, a control block is stored in the region of memory immediately before the pointer to be returned from the call. This means that `free()` may be implemented by subtracting from the returned pointer and reading off the control information, which is typically the block size plus some information that allows it to be put back in the free list - a linked list of unallocated blocks.

When the user requests an allocation, the free list is searched until a block of identical or larger size to the amount requested is found, then if necessary it is split. This can lead to memory fragmentation if the user is continually making many allocations and frees of unpredictable size and and at unpredictable intervals (not all real programs behave like that, the simple scheme is often adequate for small programs).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Many programs require large numbers of allocations of small objects of the same size. This is very easy to implement. Simply use a block with a next pointer. So if a block of 32 bytes is required:

```
union block
```

```
{
   union block * next;
   unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

This scheme is extremely fast and efficient, and can be made generic with a certain loss of clarity.