

Computer Programming

# Dynamic Memory Allocation

Marius Minea

marius@cs.upt.ro

24 November 2015

## When to use pointers ?

When the language *forces* us to:

*arrays* (memory blocks) cannot be passed / returned from functions  
only their *address* (array name is its address)

addresses carry *no size* information  $\Rightarrow$  must pass size parameter

*strings*: a string (constant or not) is a `char *`  
need not pass size, since null-terminated

*functions*: a function name is its address

When a function needs to modify variable passed from outside  
pass *address* of variable

**WARNING!** Any address passed to a function needs to be valid  
(point to allocated memory)

functions *use* their arguments  $\Rightarrow$  pointers must be valid

## When is allocation the job of the callee (called function)?

If a function needs arrays only for *temporary storage*, one can use *variable-length arrays* (since C99)

array of  $n$  elements,  $n$  known at runtime: `int a[n];`

But, if the function has an array result, array must be *allocated* and *passed from outside*

(including length, function has no way of knowing it!)

see examples: add two vectors, multiply two matrices

The more flexible the inputs, the higher the *burden on caller*

concatenate array of strings – caller must precompute length

multiply two bignums – caller must compute size of product

also, function is less natural (has address of *result* as *argument*)

⇒ would like called function to be able to *create* result object

# Dynamic allocation

*Dynamic memory allocation* (functions from `stdlib.h`)

allows us to obtain *at runtime* a memory block of the desired size

`void *malloc(size_t size);`                      allocates size bytes

`void *calloc(size_t n, size_t size);`        n\*size bytes set to 0

Return value: address of allocated memory or `NULL` on error

(insufficient memory)  $\Rightarrow$  *must test result!*

Frequent use: dynamically allocate array of n objects of type T:

```
T *p = malloc(n * sizeof(T)); // T may be int, char *, etc
```

```
if (p) // non-null=success: use p
```

```
    for (int i = 0; i < n; ++i) // room for n objects
```

```
        p[i] = ...;           // use p like an array
```

## Reallocating and freeing memory

*Changing the size* of a memory zone allocated with malloc/calloc:

```
void *realloc(void *ptr, size_t size);    requests new size
```

Can only resize memory allocated *dynamically* (not static arrays)

`size` is the complete new size, NOT an extra to add

May move memory contents and return address different from ptr

```
if (p1 = realloc(p, size)) { p = p1; /* now use p */ }  
else { /* reallocation failed, but we still have p */ }
```

realloc(NULL, len) works like malloc(len)

⇒ loop can init p = NULL, do realloc(p, ...) in first cycle

Allocated memory *must be freed* when no longer needed

```
void free(void *ptr);    frees memory block allocated with c/malloc
```

If forgotten, long-running programs (server, browser, etc.)

may consume memory (*memory leaks*) until exhausted.

## When and how to use dynamic allocation

*NO* when needed memory amount known in advance

*YES*, when needed memory amount not known at compile-time  
(dynamically linked structures: lists, trees; arbitrarily large input)

*YES*, when we must return an object created in a function  
(Can't return address of local variable, lifetime is function scope)

```
char *strdup(const char *s) {           // creates copy of s
    char *d = malloc(strlen(s) + 1); // enough for s and '\0'
    return d ? strcpy(d, s) : NULL; // copy and return dest
}
```

*YES*, to copy and keep an object read into a temporary variable

```
char *tab[10], buf[81];
int i = 0;
while (i < 10 && fgets(buf, 81, stdin))
    tab[i++] = strdup(buf); // save address of copy
```

## Example: reading an arbitrarily long line

```
#include <stdio.h>
#include <stdlib.h>
#define BLOCK 64          // suitable size, not too small
char *getline(void) {
    char *tmp, *s = NULL; // initialize for realloc
    unsigned cnt = 0, size = 0; // keep room for \0
    for (int c; (c = getchar()) != EOF; ) {
        if (cnt >= size) // allocated block full
            if (!(tmp = realloc(s, (size+=BLOCK)+1))) { // +1 for \0
                ungetc(c, stdin); break; // if no more room
            } else s = tmp; // use new address
        s[cnt++] = c; // add last char
        if (c == '\n') break; // end on newline
    } // end with \0, reallocate only size needed
    if (s) { s[cnt++] = '\0'; s = realloc(s, cnt); }
    return s;
}
```

## Read long line piecewise – better than many getchar()

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 64
char *getline(void)
{
    char *line = NULL, *tmp;
    unsigned len, sz = 0; // allocated so far
    do { // if no more mem, return piece read
        if (!(tmp = realloc(line, sz + INCR))) return line;
        line = tmp; // realloc OK
        if (!fgets(line + sz, INCR, stdin)) // no more ?
            if (sz) break; else { free(line); return NULL; }
        sz += (len = strlen(line + sz)); // add length read
    } while (line[sz-1] != '\n' && len == INCR-1); // not EOL
    return realloc(line, sz + 1); // shrink to size
}
```



## How to allocate a matrix

```
void *pm = malloc(LIN * COL * sizeof(elementype));
```

but what is the right type of the pointer ?

A matrix is an array of lines. A line is an array of COL elements.

By writing `typedef double line[5];` (line is now a type name) we see that the type of a pointer to a line is `double (*)[5]`

So for a pointer to a matrix (i.e., to its first line), we should write:

```
double (*pm)[5] = malloc(3 * 5 * sizeof(double));
```

We could also write `line *pm = ...`

How to declare a function that returns such a type?

```
double (*allocmat(unsigned lin, unsigned col))[] {  
    double (*pm)[col] = malloc(lin * col * sizeof(double));  
    for (int i = 0; i < lin; ++i)  
        for (int j = 0; j < col; ++j) pm[i][j] = i*col + j;  
    return pm;  
}
```

What does this syntax say? We'd like to be able to use the result of, say, `allocmat(3, 5)` like the pointer `pm` above. Thus we get:

```
double (*allocmat(...))[...]
```

## How to allocate a matrix (cont'd)

We can't put `[col]` in the function header, since `col` is only visible inside the parameter list (...) and function body {...}

The (incomplete) type returned by the function: `double (*)[]` is compatible with the (more precise) type of `pm`: `double (*)[col]`. So the **return** statement is well typed. In `main` we could write:

```
int main(void) {
    double (*m)[5] = allocmat(3, 5);
    printf("%g\n", m[2][4]);
    return 0;
}
```

Alternatively, we could use **typedef**:

```
typedef double (*matpointer)[];
matpointer allocmat(unsigned lin, unsigned col) { /*same code*/ }
```

If the number of columns is fixed, we can use it in `[ ]` with either the **typedef** or the original function declaration:

```
double (*allocmat(unsigned lin))[5] { /*col fixed */ }
```