

CSCI 2410 Systems Programming

Fall 2026

Instructor: Jie Zhou

Department of Computer Science
George Washington University



Slides materials are partially credited to Gabriel Parmer and Sibin Mohan at GWU.

Memory Allocation

Types of Memory Allocations

- Automatic and fixed-sized allocation
- Dynamic heap allocation

Automatic and Fixed-sized Allocation

- Global objects
 - Size is fixed when the program is compiled.
 - e.g, `char name[128];` defined globally (outside functions)
- Local objects
 - Storage is created automatically when execution enters a function.
 - Their size is typically known at compile time.
 - e.g., “`int i;`” and “`double *pd`” variables
- Compilers know exactly how much memory is needed.
 - `char arr[n];` would be illegal if `n` is not known at compile time.

Dynamic Allocation

- Memory allocated dynamically on the heap at runtime
- Allocated using `malloc` family library calls
 - Similar to using `new` to allocate memory in Java, but we need to do more work.

```
char* str = (char *)malloc(128 * sizeof(char));
```

- 128 bytes of memory allocated dynamically
- This is completely legal:

```
char* str = (char *)malloc(n * sizeof(char));
```

- The value of `n` can change at runtime.

malloc

malloc(3)

Library Functions Manual

malloc(3)

NAME

malloc, free, calloc, realloc, reallocarray – allocate and free dynamic memory

LIBRARY

Standard C library (`libc`, `-lc`)

SYNOPSIS

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

```
void *malloc(size_t size);
void free(void *_Nullable ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *_Nullable ptr, size_t size);
void *reallocarray(void *_Nullable ptr, size_t nmemb, size_t size);
```

Feature Test Macro Requirements for glibc (see `feature_test_macros(7)`):

```
reallocarray():
    Since glibc 2.29:
        _DEFAULT_SOURCE
    glibc 2.28 and earlier:
        _GNU_SOURCE
```

DESCRIPTION

`malloc()`

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. The memory is not initialized. If `size` is 0, then `malloc()` returns a unique pointer value that can later be successfully passed to `free()`. (See "Nonportable behavior" for portability issues.)

malloc

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



`size_t`: represents sizes and counts of bytes in memory.

- An unsigned integer type, defined using primitive integer types.
- Defined in `<stddef.h>`
- Size of `size_t` depends on the platform. Try `sizeof(size_t)`.
- Not a keyword of C

Platform	<code>size_t</code>
32 bit	unsigned int
64 bit	unsigned long

```
typedef unsigned long size_t;
```

malloc

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



`void *` is a *generic* pointer type, i.e., it may point to any type of data.

- Can assign a `void *` pointer to any other pointer type
- Cannot dereference a `void *`, because the compiler does not know the type of data to access.

```
int x = 10;  
void *p = &x;  
int y = *p;    // cannot compile
```

malloc

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

- Allocate raw memory of `size` bytes
- No initialization
 - “data” is whatever is in memory.
- It is a good practice to use `memset()` to set memory to 0.
- Returns a pointer to newly allocated memory

```
char* str = (char *)malloc(128 * sizeof(char));
```

- Allocate 128 bytes of memory
- `str` points to the beginning of this chunk of memory.

calloc

```
void *calloc(size_t nmemb, size_t size);
```

`calloc()`

The `calloc()` function allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is 0, then `calloc()` returns a unique pointer value that can later be successfully passed to `free()`.

- Conventionally used to allocate memory for an array
- `nmemb`: number of **m**embers—or, number of elements/items
- `size`: size of each member/element/item
- Totally memory allocated: `nmemb * size`
- The whole chunk of memory is initialized to 0.
- Returns a pointer to the newly allocated memory.
- e.g., `calloc(10, sizeof(int));`
 - Allocate memory for 10 integers, each one set to 0

What if allocated memory is insufficient?
Or, we do not need that much memory any more?

realloc

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

- Reallocate the memory pointed by `ptr`
- i.e., grow/shrink it to the new size; grows/shrinks *in place* if possible
- If not enough space to grow,
 - allocate new memory of `size`
 - copy old data to the new memory region
 - free the old memory region
 - return a new pointer to the new memory region
- If new size is smaller than the old memory region's size,
 - most likely to shrink in place, but may reallocate the old memory to a new place
 - copy only the new size of data to the new memory region

realloc

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

- Reallocate the memory pointed by `ptr`
- i.e., grow/shrink it to the new size; grows/shrinks *in place* if possible
- Returns one of
 - the original pointer, if no reallocation happens in place
 - a new pointer, if a new chunk of memory is allocated in another place
- `realloc` will not set extended memory to 0, even if originally memory was allocated using `calloc`
 - e.g., if `calloc` created a 10 byte array, `realloc(p, 20)` will not set the last 10 bytes to 0.

free

```
void free(void *ptr);
```

`free()`

The `free()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()` or related functions. Otherwise, or if `ptr` has already been freed, undefined behavior occurs. If `ptr` is NULL, no operation is performed.

- Releases memory block referenced by `ptr`
 - Other processes/OS can now use it
- Only works for dynamically allocated memory on the heap
- No action if `ptr` is NULL
 - NULL is not a keyword of C; it is defined in the standard library, usually `(void *)0`
- Does not return anything

Common Memory Management Errors

- Failed allocation: memory not allocated
 - Maybe not enough memory in system
 - or requested too large a piece of memory
 - returns NULL
 - check for the return value before use!

```
int foo (void)
{
    int *a = malloc(sizeof(int));
    /* Potential error: did not check return value! */
    *a = 1;
    free(a);

    return 0;
}
```

Common Memory Management Errors

- Failed allocation: memory not allocated
 - Maybe not enough memory in system
 - or requested too large a piece of memory
 - returns NULL
 - check for the return value before use!

```
int foo (void)
{
    int *a = malloc(sizeof(int));
    if (a == NULL) {
        perror("malloc");
    }
    *a = 1;
    free(a);

    return 0;
}
```

Common Memory Management Errors

- Dangling pointers: accessing free'd pointers
 - Accessing stale memory; or, worse,
 - Memory may have been reallocated
 - Use-After-Free (bug)
 - Bad things happen (crashes, leaking data, etc.)
 - Avoid `free()` until all pointer uses are done.

```
int foo (void)
{
    int *a = malloc(sizeof(int));
    if (a == NULL) {
        perror("malloc");
    }
    *a = 1;
    free(a);

    /* Error: accessing what `a` points to after `free`! */
    return *a;
}
```

Common Memory Management Errors

- Double/Invalid free: free twice or free an invalid address
 - Bad/Unpredictable things happen!
 - Accidentally free memory used elsewhere
 - Corrupt program data

```
#include <stdlib.h>

int
main(void)
{
    int *a = malloc(sizeof(int));
    if (!a) return -1;
    free(a);
    free(a);
    /* Error: yeah, don't do that! */

    return 0;
}
```

Common Memory Management Errors

- Memory leaks
 - Memory *never* get freed/released.
 - Overtime, less memory available
 - Can slow down/crash entire system!
 - Always pair a `free()` with an allocation

```
int foo (void)
{
    int *a = malloc(sizeof(int));
    if (a == NULL) {
        perror("malloc");
    }
    *a = 1;

    /* Error: never releasing memory pointed by a! */
    return 0;
}
```