Dynamic Memory Allocation

Cr: Liam Saliba Week 9 slides 2022

Motivation

- I want to make an array, but I don't know how many items I want to store.
- I don't want to make it too large, otherwise it will be inefficient.
- The array should grow with input.
- I want to create structs / arrays in another function to make neater code.

<stdlib.h>

void ***malloc**(size_t size)
    Allocates a contiguous block of memory of SIZE bytes, returning a void
    Pointer to the first byte. Elements are left *uninitialised*.

void **free**(void *ptr)
    Frees the memory block pointed to by ptr, IF that pointer once returned
    by malloc / calloc / realloc. Pointer address is *NOT* changed.

void ***calloc**(size_t nmemb, size_t size)
    Same as malloc, but (1) checks if nmemb * size will integer overflow, and
    (2) initialises all elements to 0. (Slow)

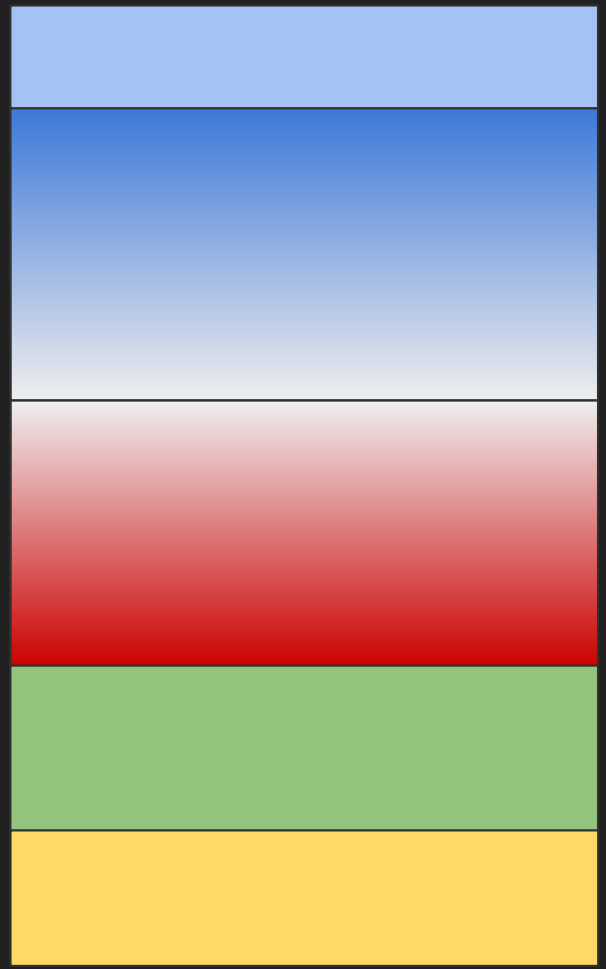
void ***realloc**(void *ptr, size_t size)
    Changes size of memory block pointed to by ptr to one having size SIZE,
    returning a pointer to that memory block. Some caveats.

void ***malloc**(size_t size)

Allocate memory of **size** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails.

```
#include <stdlib.h>

// create an array of size 20.   (80 bytes)
int *arr = malloc(sizeof(int) * 20);

// equivalent to…
int B[20];
```

‘High’ address 0xFFFF FFFF



(Command-line arguments
Environment variables)

stack local variables

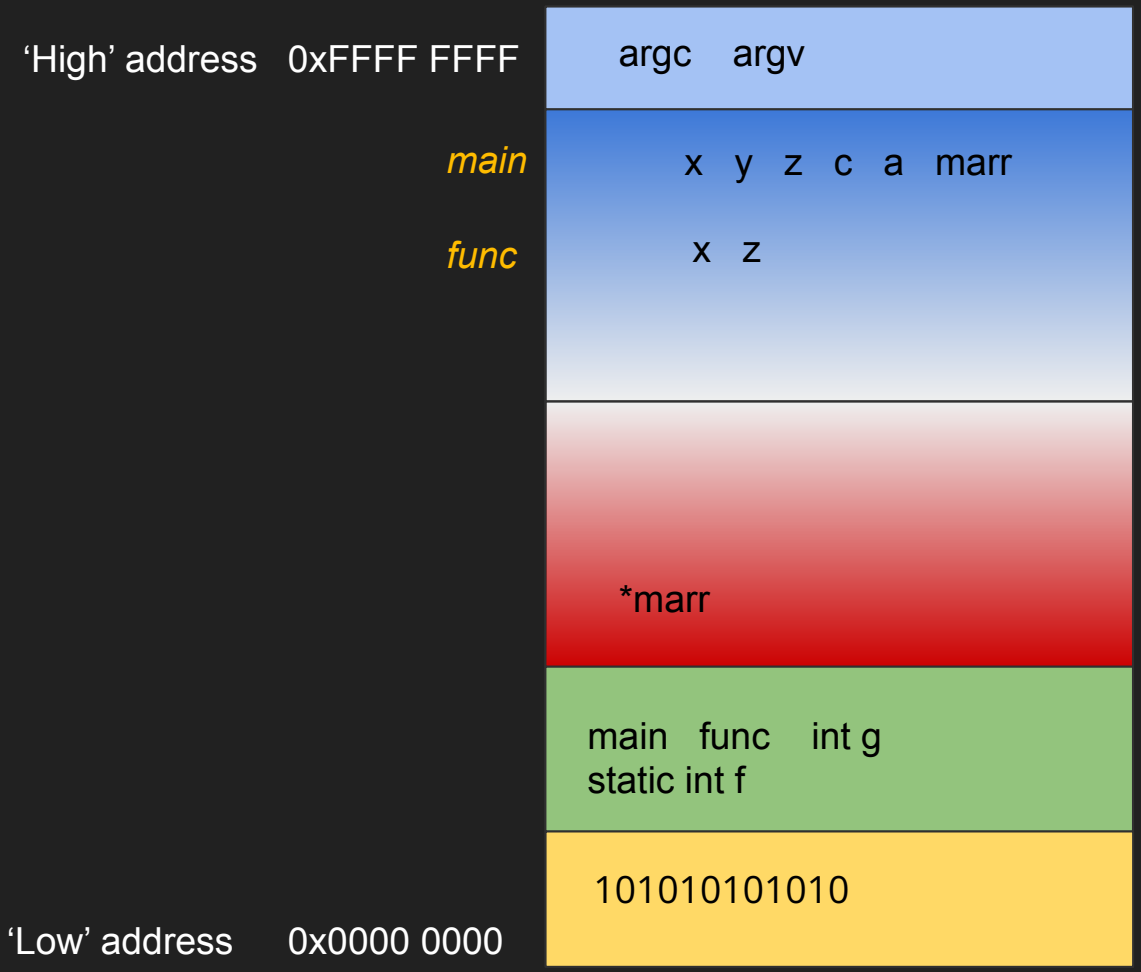
The memory available to your C program.

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘Low’ address 0x0000 0000



(Command-line arguments Environment variables)

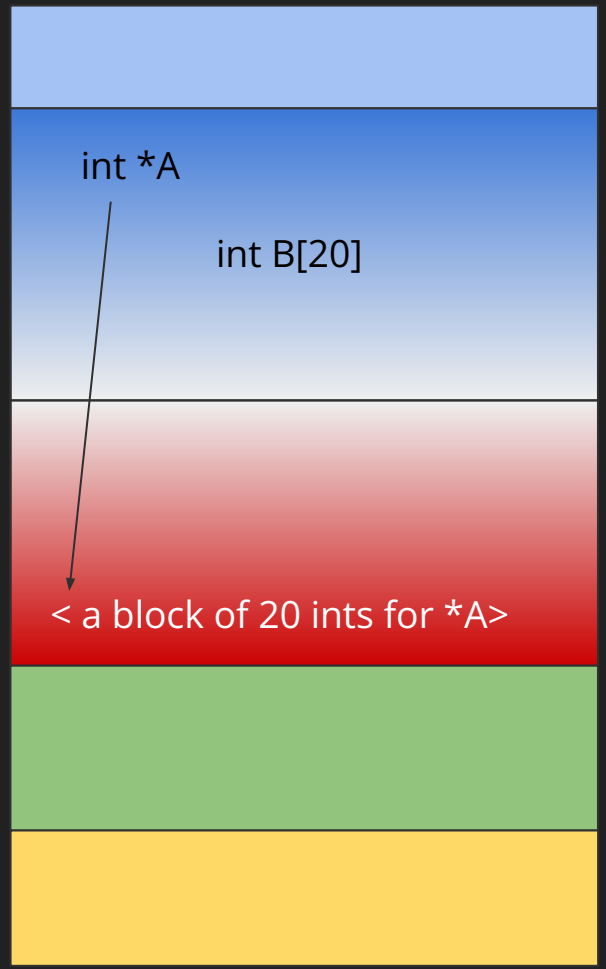
stack local variables

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘High’ address 0xFFFF FFFF



*(Command-line arguments
Environment variables)*

stack local variables

```
int *A
 = malloc(20 *
     sizeof(int));

int B[20];
```

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘Low’ address 0x0000 0000

void ***malloc**(size_t size)

Allocate memory of **size** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails.

```
#include <stdlib.h>

int *arr = malloc(sizeof(int) * 20);
```

void ***malloc**(size_t size)

Allocate memory of **size** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails.

```
#include <stdlib.h>

int *arr = malloc(sizeof(int) * 20);
if (arr == NULL) {
    printf("Memory allocation failed!\n");
    exit(EXIT_FAILURE);
}
```

void ***malloc**(size_t size)

Allocate memory of **size** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails.

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

int *arr = malloc(sizeof(int) * 20);
assert(arr != NULL);
```

void ***malloc**(size_t size)

Allocate memory of **size** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails.

```
#include <stdlib.h>

int *arr = malloc(sizeof(int) * 20);
assert(arr);        // #include <assert.h>
```

```
NULL == (void *) 0
so arr != NULL  is the same as   arr != 0
```

If malloc fails …

```
int *arr = malloc(sizeof(int) * 20);
arr = NULL;  // assume malloc returns NULL

printf("%d\n", arr[0]);
```

If malloc fails ...

```
int *arr = malloc(sizeof(int) * 20);
arr = NULL;  // assume malloc returns NULL

printf("%d\n", arr[0]);

> segmentation fault

arr[0]   =>   *(arr + 0)   => *(NULL)   -- undefined!
```

drop-in replacement checked malloc

```
void *checked_malloc(size_t size) {
    void *p = malloc(size);
    if (p == NULL) {
        fprintf(stderr, "Failed to create memory!\n");
        assert(0); // FAIL
    }
}

int *arr = checked_malloc(sizeof(int) * 20);
```

alt version

```
void *safe_malloc(size_t size, char *msg) {
    void *p = malloc(size);
    if (p == NULL) {
        fprintf(stderr, "Failed to create memory: %s\n", msg);
        exit(EXIT_FAILURE);
    }
}

int *arr = safe_malloc(sizeof(int) * 20, "arr");
```

malloc does not initialise elements.

```
int n = 20;
int *arr = malloc(sizeof(int) * n);
assert(arr);

print_int_array(arr, n);
// 343484389 9 234292309 -234923   ...
```

void *calloc(size_t nmemb, size_t size)

Allocate memory of **size * nmemb** bytes, return a **void pointer** to the memory block, or NULL if the allocation fails; sets all elements to 0.

```
// create an array of size 20; all elements set to 0
int *arr = calloc(20, sizeof(int));
assert(arr); // not NULL
```

void *calloc(size_t nmemb, size_t size)

nmemb: number of members in array
size: size of type of elements in array

returns pointer to the block of memory created

void *calloc(size_t nmemb, size_t size)

```
int n = 20;

// create an array of size n; all elements set to 0
int *arr = calloc(sizeof(int), n); assert(arr);

// equivalent to:
int *arr = malloc(sizeof(int) * n); assert(arr);
memset(arr, 0, n);

// equivalent to:
int arr[10];
memset(arr, 0, n);
```

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

written (roughly) in terms of malloc:

```
void *calloc( size_t nmemb, size_t size) {
    // will the size overflow?
    if (nmemb > MAX_INT / size) return NULL;

    void *p = malloc(nmemb * size);

    return memset(p, 0, nmemb*size);
}
```

void *calloc(size_t nmemb, size_t size)

Warning:

```
int *x = calloc(10000, sizeof(int));
```

only use calloc if **necessary**

do you **need** all elements set to 0?

void *memset(void *ptr, int c, size_t n)

Copies c to n **bytes** of memory pointed to by ptr; returns ptr.

```
int *arr = malloc(sizeof(int) * n);
assert(arr);
memset(arr, 1, n);
// sets every byte to 1, not every element to 1.
// use a for loop to set each element instead.
// fine to use memset for `char`s -- everything else,
// be careful!

arr[0] == 0b00000001000000010000000100000001 // 16843009
```

```
void *memcpy(void *dest,
             const void *src, size_t n)
```

Copies **n** bytes from **src** to **dest.**

```
// Example: copying an array
int A[10];
for (int i = 0; i < 10; A[i++] = i);

int B[20];
memcpy(B, A, 10*sizeof(int) );
// B = {0,1,2,3,4,5,6,7,8,9, followed by garbage}
```

```
void *memcpy(void *dest,
             const void *src, size_t n)
```

Copies **n** bytes from **src** to **dest**

Careful: make sure there's enough room in **dest** to store n bytes of **src**!

```
// Example: copying an array
int A[10];
for (int i = 0; i < 10; A[i++] = i);

int B[2];
memcpy(B, A, 10*sizeof(int) );
// possible segmentation fault!
```

Motivation

- I want to make an array, but I don't know how many items I want to store. I don't want to make it too large, otherwise it will be inefficient. **The array should grow with input.**

Read ints

```
#define INITIAL_SIZE 10

int *arr = checked_malloc(INITIAL_SIZE * sizeof(int)); // int arr[10];
int max_len = INITIAL_SIZE; // 10     // capacity
int n = 0;  // size length  - how many elements actually stored?

int next_int;  // 1 2 3 4 5 6 7 8 9 10 11
while (scanf("%d", &next_int) == 1) {
    if (n == max_len) {
        embiggen_arr(arr, &max_len, sizeof(int));
    }
    arr[n++] = next_int;
}
```

```
// Double the size of the block pointed to by p.
void *embiggen_arr(void *p, int *max_len, size_t size) {
    int oldsize = size * *max_len;
    *max_len *= 2; // double the capacity
    int newsize = size * *max_len;

    void *newp = malloc(newsize);
    assert(newp);
    memcpy(newp, p, oldsize);
    free(p);   // We don't need the old array anymore.
               // WARN: invalidates all existing pointers to p

    return newp;
}
```

```
// Double the size of the block pointed to by p.
void *embiggen_arr(void *p, int *max_len, size_t size) {
    int oldsize = size * *max_len; // 4 * 10 = 40 bytes
    *max_len *= 2; // double the capacity  (20)
    int newsize = size * *max_len; // 4 * 20 = 80 bytes

    void *newp = malloc(newsize);
    assert(newp);
    memcpy(newp, p, oldsize);
    free(p);   // We don't need the old array anymore.
               // WARN: invalidates all existing pointers to p

    return newp;
}
```

```
// Double the size of the block pointed to by p.
void *embiggen_arr(void *p, int *max_len, size_t size) {
    int oldsize = size * *max_len;
    *max_len *= 2; // double the capacity  (20)
    int newsize = size * *max_len;

    p = realloc(p, newsize);
    assert(p);




    return p;
}

// dynamic array  (variable / dynamic size)
```

void *__realloc__(void *p, size_t size)

Changes the size of the memory block pointed to by p to one having a new size of **size**; and returns that pointer (or NULL if failed).

void ***realloc**(void *p, size_t size)

`realloc(NULL, size)` == `malloc(size)`

~~`realloc(p, 0)`~~ ~~==~~ ~~`free(p)`~~ Before C23…

where p -> a block of size **s**: (allocated)

size > s:	the new elements are uninitialised
size == s:	... realloc does nothing?
size < s:	some elements are removed / truncated (chop the end)

https://en.cppreference.com/w/c/memory/realloc

if `new_size` is zero, the behavior is implementation defined (null pointer may be returned (in which case the old memory block may or may not be freed), or some non-null pointer may be returned that may not be used to access storage). Such usage is deprecated (via C DR 400 🔒).(since C17)	(until C23)
if `new_size` is zero, the behavior is undefined.	(since C23)

from the manual

The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr. If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory. If ptr is NULL, realloc() is identical to a call to malloc() for size bytes. If size is zero and ptr is not NULL, a new, minimum sized object is allocated and the original object is freed. When extending a region allocated with calloc(3),

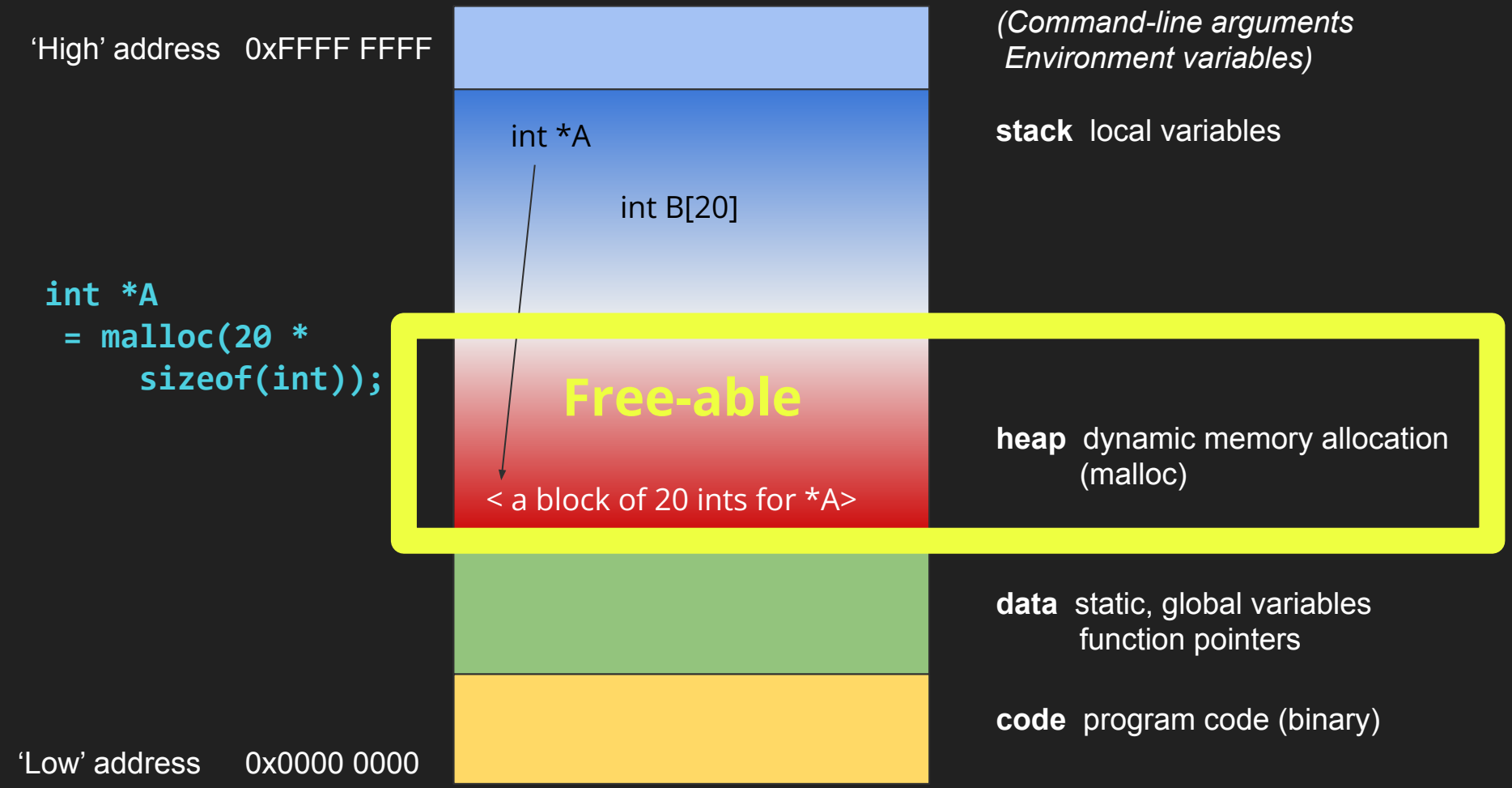
```
typedef struct {
    size_t length, capacity;
    int *els;  // this can be used just like a normal array!
} int_arr_t;

int_arr_t int_arr_new() {
    int_arr_t arr = {0, INITIAL_SIZE, NULL};
    arr.els = checked_malloc(INITIAL_SIZE * sizeof(int)); // int arr[10];
    return arr;
}

void int_arr_free(int_arr_t *arr) { free(arr->els); arr->els = NULL; }

int_arr_t int_arr_insert(int_arr_t arr, int el) {
    if (arr.length == arr.capacity) embiggen_arr(arr.els, &arr.capacity);
    arr.els[arr.length++] = el;
    return arr;
}
```

void *free(void *p)

Frees the memory block pointed to by p
iff p was malloc'd (/ calloc'd / realloc'd).

If you malloc memory, you **must** free it yourself once you are done with it.

=> Memory Leak!


'High' address 0xFFFF FFFF
(Command-line arguments
Environment variables)
int *A
stack local variables
int B[20]
int *A
= malloc(20 *
sizeof(int));
Free-able
heap dynamic memory allocation
(malloc)
< a block of 20 ints for *A>
data static, global variables
function pointers
code program code (binary)
'Low' address 0x0000 0000

Memory leak! (and worse)

Don’t try this at home I’m not liable for lost data or assignments this is not an excuse for an extension

```
#include <stdlib.h>

int main() {
    for (int i = 0; i < 1000000; i++) {
        int *p = calloc(400000, sizeof(int));
        p[10000] = 5;
        // free(p);
    }
}
```

```c
#include <stdlib.h>

int main() {
    int *p = calloc(400000, sizeof(int));
}
```

Google Chrome's main method leaked

```
int main() {
    void *v = malloc(6000000000);
    launchchrome();
    free(v);
    return 0;
}
```

Correctly using malloc

```
int *arr = checked_malloc(INITIAL_SIZE * sizeof(int));


// ...do stuff with arr

// now we're done with arr -- let's free it!
free(arr);

// do some other stuff
```

Correctly using malloc

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);


// ...do stuff with arr

// now we're done with arr -- let's free it!
free(arr);

// do some other stuff

// Oh, gotta remember to free our malloc'd variables
free(arr);
```

Double free

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);
// say arr == 0x100

// ...do stuff with arr

// now we're done with arr -- let's free it!
free(arr);
// now the memory at 0x100 is free but arr still == 0x100.


free(arr);
*** Error in `./program': double free or corruption (fasttop): 0x085f6008
```

Double free

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);


// ...do stuff with arr

free(arr);
arr = NULL;

free(arr);   // nothing happens!
```

Memory corruption

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);


// ...do stuff with arr

free(arr);

// forgot that it's already been free'd ...
arr[4] = 5;
```

what happens?

Memory corruption

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);


// ...do stuff with arr

free(arr);

// forgot that it's already been free'd ...
arr[4] = 5;
```

silent error, very hard to debug.

Memory corruption

```
int *arr = malloc(INITIAL_SIZE * sizeof(int));
assert(arr);


// ...do stuff with arr

free(arr);
arr = NULL;  // 0
// forgot that it's already been free'd …
arr[4] = 5;
```

segmentation fault -- but at least we can debug it!

Memory allocation:	**Automatic**	**Static**	**Dynamic**
Managed by...	the compiler	the compiler	the programmer
Alloc'd...	during variable declaration int x;	at compile time	using `malloc` / `calloc` / `realloc`
Free'd...	automatically, when exiting scope of variable	automatically, at end of program	manually, when `free` is called *!! memory leaks !!*
Memory assigned to...	**stack**, within **stack frame** (at runtime)	**data** (at compile time)	**heap** (at runtime)
Examples / Uses	local variables, function arguments, array size known	global, static variables	Dynamically sized arrays, structs, array size unknown, Large data structures, custom data structures
Issues	Limited control of lifetime; Can't change size after init; Size calculated at compile time; Size limits	Variable is allocated permanently, wastes memory; Same limitations as automatic	Memory leak (no free); Double free; Additional memory required to store pointer to memory;