Dynamic Memory Allocation

(summary)

Dynamic memory allocation

- Ask for memory, as we need it.
- Biggest limitation: We **must** free it ourselves once done with it.
- Uses:
  - Dynamic array allocation - when we don’t know input size
  - Large array / data structure - start small, increase size as we need it
  - Other data structures - Linked lists, etc.

Dynamic memory allocation example

Build a program that reads in a list of integers, and outputs its sum.

The program should handle lists of **any** (reasonable) **length**.

Dynamic memory allocation example

Build a program that reads in a list of integers, and outputs its sum.

The program should handle lists of **any** (reasonable) **length**.

How to do this?

1. Use a variable to hold the *capacity* of the array :: *size*
2. Start with ‘small’ capacity - say 10.
3. Use **malloc** to allocate a block of memory of that size to hold the array.
4. Read in each integer once at a time.
5. If we read in *size* integers, **double** *size* and **realloc**.
6. Finish reading the integers
7. Do our required task -- calculate sum, output them.
8. Release the block of memory allocated with **free**.

Dynamic memory allocation - <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.

When to use Dynamic Memory Allocation

- No upper bound on size of input
- There *is* an upper bound on size of input, but it's large -> waste memory
- Want to use a lot of memory
- Want to have memory / data structures persist in your program
- Want memory on the heap

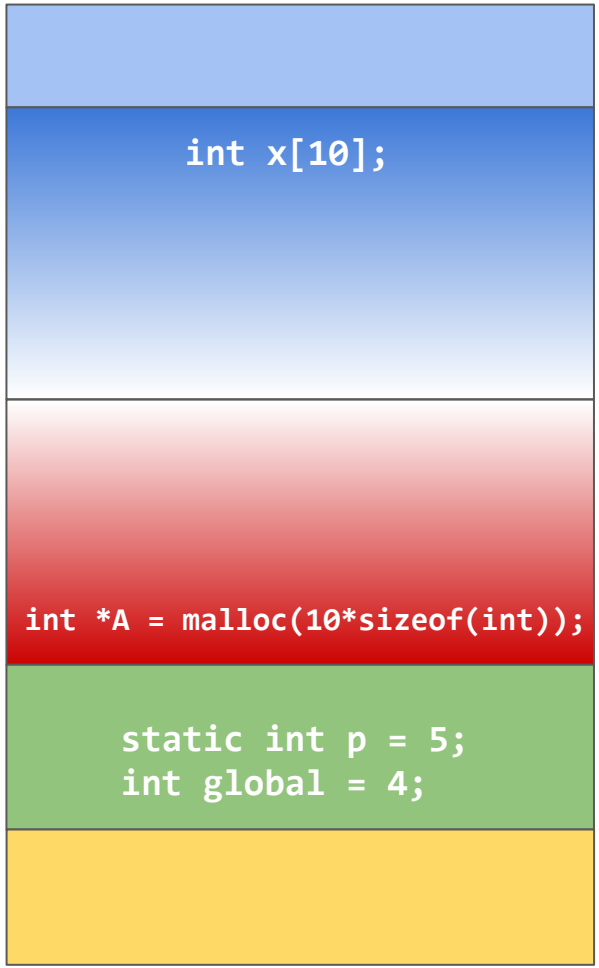
Dynamic memory allocation issues.

Gotta watch out when doing dynamic memory allocation.

Issues to consider:

- Failed to allocate memory
- Double Free
- Memory Corruption

‘High’ address 0xFFFF FFFF



(Command-line arguments
Environment variables)

stack local variables

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘Low’ address 0x0000 0000

Memory allocation:	**Automatic**	**Static**	**Dynamic**
Managed by...	the compiler	the compiler	the programmer
Alloc'd...	during variable initialisation	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;

Static & Automatic allocation

Scenario

Build a program that reads in a list of integers, and outputs its sum.

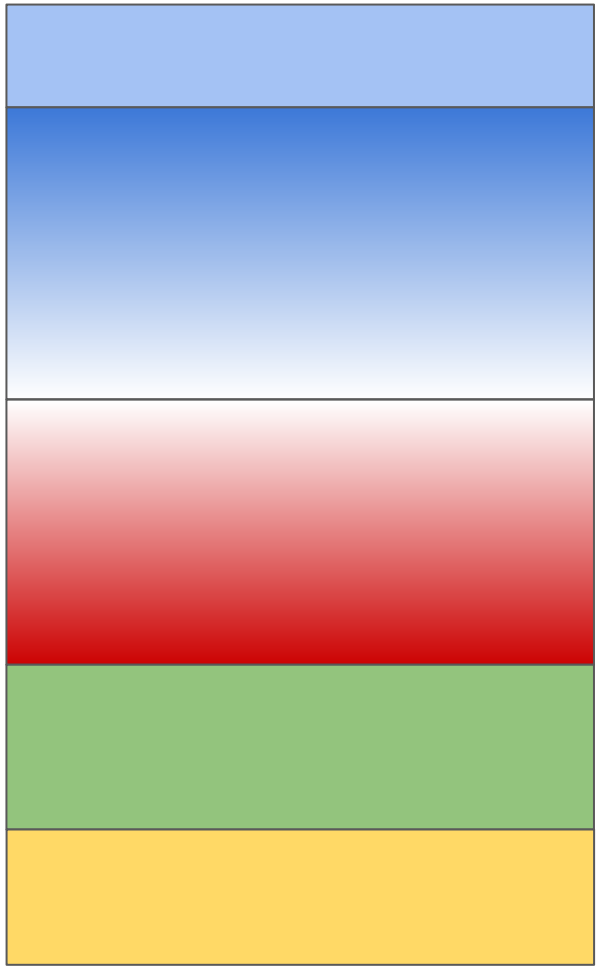
The program should handle lists of **any** (reasonable) **length**.

Memory Allocation

- **Automatic**
  - By the compiler, at runtime, stored in **stack**
- **Static**
  - By the compiler, at compile time, stored in **data**
- **Dynamic**
  - By the programmer, at runtime, stored in **heap**

‘High’ address 0xFFFF FFFF

Recall memory layout



(Command-line arguments
Environment variables)

stack local variables

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘Low’ address 0x0000 0000

Static & Automatic Memory Allocation

What's in memory, and what's in scope at each point?

(1)

```
int g = 4;
void func(int arg, char *str) {
    static int x = 5;    (2)
    double pair[2] = {1.4, 4.3};
    for (int i = 0; i < g; i++) {
        printf("%d ", i * x++); (3)
    }
}
```

(4)

(5)

Static & Automatic Memory Allocation

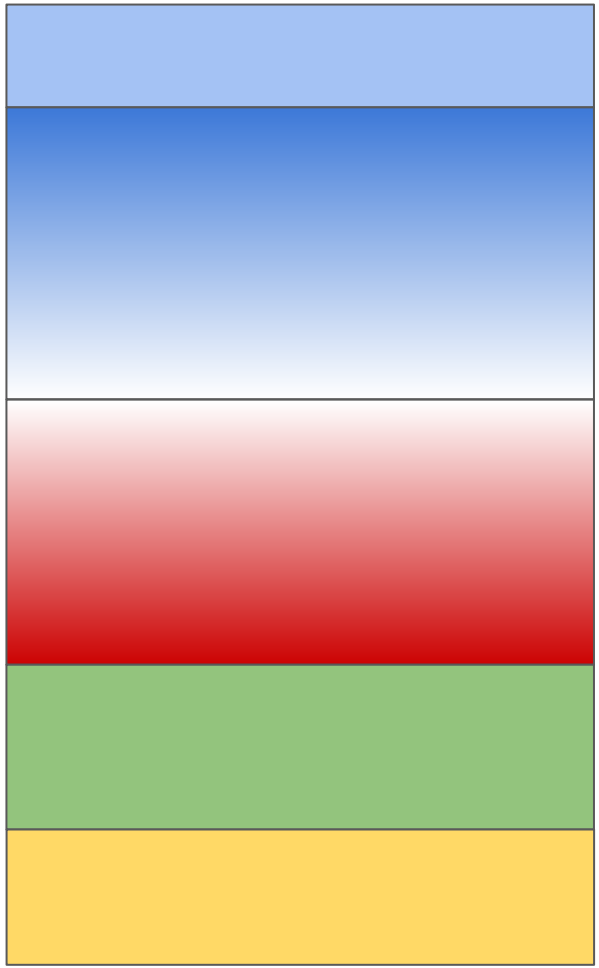
```
int g = 4;
void func(int arg, char *str) {
    static int x = 5;
    double pair[2] = {1.4, 4.3};
    for (int i = 0; i < g; i++) {
        printf("%d ", i * x++);
    }
}
```

Automatic memory allocation

- automatically allocated by the compiler at runtime, once we enter scope of the variable
- freed once we exit the scope of the variable.
- stored within **stack**

‘High’ address 0xFFFF FFFF

Where do those variables go?



(Command-line arguments Environment variables)

stack local variables

heap dynamic memory allocation (malloc)

data static, global variables function pointers

code program code (binary)

‘Low’ address 0x0000 0000

Static & Automatic Memory Allocation

```
int g = 4;
void func(int arg, char *str) {
    static int x = 5;
    double pair[2] = {1.4, 4.3};
    for (int i = 0; i < g; i++) {
        printf("%d ", i * x++);
    }
}
```

Static memory allocation

- allocated by the compiler at compile time, **once**.
- freed at program exit.
- stored within **data**

Problems with **automatic** allocation

- Size calculated at compile time
  - no dynamically sized arrays
  - eg: Can’t create array based on input size
- Allocated size can’t be changed after initialisation
  - Need to allocate upper bound of requirement -- wastes memory. eg: storing between 10-10000 items, must create array of size 10000 to ensure there’s enough space
- Limited stack memory
  - Limited size to memory blocks
  - “Array too large”

Problems with **static** allocation

- Once a variable is allocated, it is permanently allocated
  - Free’d at program exit
  - Waste memory
- Created at compile time -- can only have one of each item
- Same problems as automatic allocation

Scenario

Build a program that reads in a list of integers, and outputs its sum.

The program should handle lists of **any** (reasonable) **length**.

Can we do this with automatic / static allocation?

stdlib.h memory functions

Dynamic memory allocation - <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)

Allocates a contiguous block of memory of SIZE bytes within the heap, returning a void pointer to the first byte. Elements are left *uninitialised*.

```
int *arr = (int *) malloc(5 * sizeof(int));
```

^ Allocates a block of size 5 ints within the heap. Stores an address in the stack to the first element of that block in arr. Effectively the same as:

```
int arr[5];
```

Difference? Malloc: we need a pointer to access the memory block - an additional 8B of memory.

```
arr[0] = 5;
```
works for both.

void *malloc(size_t size)

Equivalencies:

```
int *arr = (int *) malloc(5 * sizeof(int));

int *arr = malloc(5 * sizeof(int));
```

Not necessary to cast, but may make your code clearer.

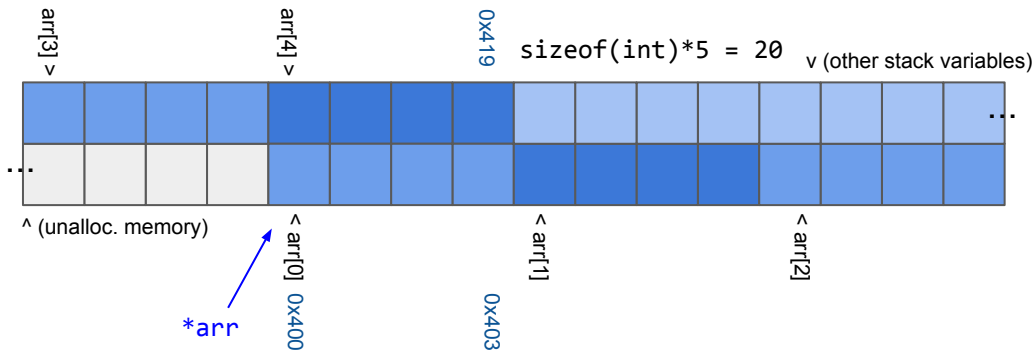
```
int *arr;
... some code ...
arr = malloc(5 * sizeof(int));          // what is the type of arr?
// OR
arr = (int *) malloc(5 * sizeof(int)); // ah, it's an int *

arr = malloc(5 * sizeof(*arr));         // sizeof takes variables
```

void *malloc(size_t size)

```
int arr[5];
```

Stack

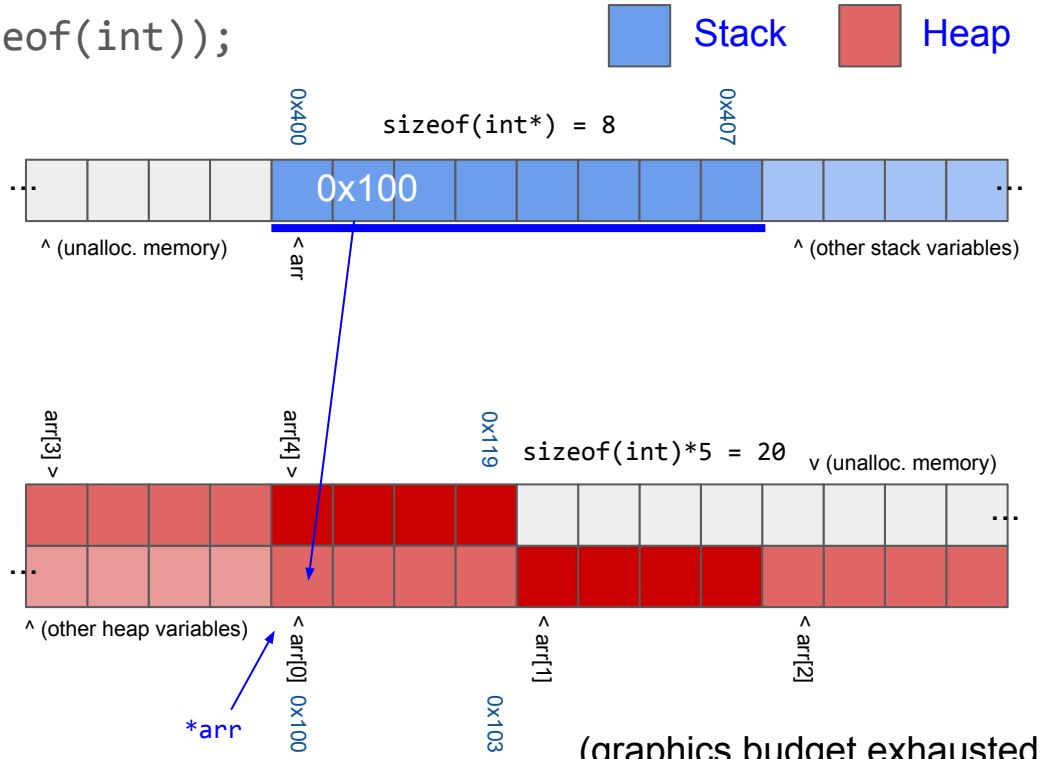
```
                 ?? ...
[0x400 / stack]  arr[0] = uninit
                 arr[1] = uninit
                 arr[2] = uninit
                 arr[3] = uninit
                 arr[4] = uninit
                 ?? ...
```



void *malloc(size_t size)

```
int *arr = (int *) malloc(5 * sizeof(int));
```

```
[0x400 / stack]  arr    = 0x100

                  ?? ...
[0x100 / heap ]   arr[0] = uninit
                  arr[1] = uninit
                  arr[2] = uninit
                  arr[3] = uninit
                  arr[4] = uninit
                  ?? ...
```



(graphics budget exhausted)

void *malloc(size_t size)

We wish to create an int array of size 5 on the heap.
Which is the correct malloc call?

```
// (a)
int *arr = (int *) malloc(5 * sizeof(*arr));
```

```
// (b)
int *arr = (int *) malloc(5 * sizeof(arr));
```

Dynamic memory allocation - <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 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.

When we call malloc, we **MUST** free it later.

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

// do stuff with arr

free(arr);    // arr is free'd
```

Some things to be careful about ….

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.

Because of (2) - this can be slow!

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

Returns a pointer to an allocated block of memory to store 10000 ints.
All ints are set to 0. => 10000 operations! Necessary??

```
void *calloc( size_t nmemb, size_t size) {  // (roughly)
    if (nmemb > MAX_INT / size) return; // size will overflow
    void *p = malloc(nmemb * size);
    return memset(p, 0, nmemb*size);
}
```

memset

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

Sets the first **n** bytes of memory at **ptr** to value **c**.

```
memset(p, 0, n);
```

Same as:

```
int arr[] = {0, 0, 0, 0, 0, 0};   // for n = 6
```

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.

Behaviour

If ptr == NULL: == void *malloc(size_t size)

If size == 0 && ptr == NULL: == void free(void *ptr)

If size > original size: the new elements are uninitialised
If size < original size: some elements are removed / truncated

Google Chrome's main method leaked

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

Dynamic allocation issues

Dynamic memory allocation issues.

Gotta watch out when doing dynamic memory allocation.

Issues to consider:

- Failed to allocate memory
- Double Free
- Memory Corruption

Demonstration. Slides ahead for your own reading!

When malloc fails

malloc can fail (rarely) when no memory can be allocated.
- System is out of memory (embedded systems have <100mb ram)
- No contiguous free area of requested size can be located in the heap

This happens rarely - but you **should** still take care of it.

When malloc fails, it returns NULL. NULL = (void *) 0
(void pointer with value 0)

```
char *str = malloc(INIT_SIZE * sizeof(char));  // sizeof(char) == 1
if (str == NULL) {  // handle memory allocate fail
    fprintf(stderr, “Failed to allocate memory for str!”);
    exit(EXIT_FAILURE);
}
// … do stuff with str
```

Detect malloc fail - variants

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

if (str == NULL) {
    fprintf(stderr, “Failed to allocate memory for str!”);
    exit(EXIT_FAILURE);
}
// … do stuff with str
```

Detect malloc fail - variants

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

if (!str) {
    fprintf(stderr, “Failed to allocate memory for str!”);
    exit(EXIT_FAILURE);
}
// … do stuff with str
```

Detect malloc fail - variants

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

assert(str);

// do stuff with str
```

Detect malloc fail - variants

A useful library function to create:

```
void check_ptr(void *p, char *msg) {
    if (p == NULL) {
        fprintf(stderr, “check_ptr: %s”, msg);
        // .. do other stuff ..
        exit(EXIT_FAILURE);
    }
}

char *str = malloc(INIT_SIZE * sizeof(char));
check_ptr(str, “str alloc failed”);

// do stuff with str
```

Double Free

```
void func() {
    int *arr = malloc(n * sizeof(int));

    // do stuff with arr

    // we're done with arr!
    free(arr);

    // do some other stuff

    // ah, let's make sure we free'd all our malloc'd variables
    free(arr);  // bad crash.
}
```

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

Double Free

```
void func() {
    int *arr = malloc(n * sizeof(int));

    // do stuff with arr

    // we're done with arr!
    free(arr);
    arr = NULL;
    // do some other stuff

    // ah, let's make sure we free'd all our malloc'd variables
    free(arr);   // nothing happens
}
```

Memory corruption

```
void func() {
    int *arr = malloc(n * sizeof(int));

    // do stuff with arr

    // we're done with arr!
    free(arr);

    arr[4] = 4;   // BAD! that's not malloc'd anymore.  But no error!
}
```

A **silent** error! Be careful! Very hard to debug!

A solution?

Memory corruption

```
void func() {
    int *arr = malloc(n * sizeof(int));

    // do stuff with arr

    // we're done with arr!
    free(arr);
    arr = NULL;
    arr[4] = 4;   // NULL[4] <- Segmentation fault (core dumped)
}
```

A segmentation fault is still bad, but at least we can debug it.
The program will simply crash on that line.

free issues: a more realistic scenario

```
void func() {
    int *arr = malloc(n * sizeof(int));
    int *p = arr;    // a copy of that array pointer
    // do stuff with arr
    // we're done with arr! Let's free it
    free(arr);

    // do some other stuff
    // do stuff with p -> memory corruption (silent error)
    // done with p! Let's free it
    free(p);     // Double free (hard crash)
}
```

Program would crash on free(p), but if we deleted it, we'd still have memory corruption!

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

```
// Q2   :: do this one last!
Now: this is an _efficient_ solution.  DON'T do an efficent solution!

#include <stdlib.h>  // malloc, realloc

char *compute_interleave(char *strs[], int k) {
   // endstr[i] tells us if we've reached the end
   // of string strs[i], so shouldn't add \0 or garbage
   // The end of strs[i] is not necessarily \0s!
   // can buffer overflow!
   char *endstr = malloc(k);
   memset(endstr, 1, k);  // or use a for loop
   int strsleft = k;

   // need to allocate memory properly!
   int len = 0, size = 2*k;
   char *retstr = malloc(size); // don't need sizeof -- char

   // strs[i][j].
   // strs is an array of strings
   // strs[0] is the first string
   // strs[0][0] is the first char of the first string
   for (int j = 0; strsleft; j++) {
```