N-D Arrays, Dynamic Memory & Structs

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

Feb 6, 2023

Lesson Plan

1. Multidimensional Arrays
2. Dynamic Memory Management
3. Structures (Structs)
4. Linked-Lists
5. ArrayLists

Acknowledgements

Part of the slides are borrowed from Karen Reid and Andi Bergen.

Section 1

Multidimensional (N-D) Arrays

Multidimensional (N-D) Arrays

Arrays can be multi-dimensional (N-D) to represent higher dimensional tensors.

```
const int Y = 0, R = 1, B = 2, G = 3, O = 4, W = 5;
int rubiks_face[3][3] = {
    {Y, Y, R},
    {W, G, B},
    {Y, Y, R}};
```

The name of a two-dimensional array is a pointer to a pointer – a double pointer. What's the type of the name of a three-dimensional array like `rubiks_cube`?

```
int rubiks_cube[6][3][3]; // 6 faces, int ***
```

For any two-dimensional array `A`, the expression `A[k]` is a pointer to the first element in row `k` of the array.

```
int k = 2, *p = rubiks_cube[k];
for (; p < rubiks_face[k] + 2; p++) *p = Y;
```

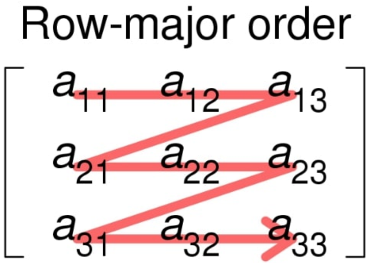

Figure 1: row-major order of a 3x3 matrix

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in *row-major order*, with row 0 first, then row 1, and so forth.

```
int *row_ptr = rubiks_face[0];
int *square_ptr = &row_ptr[2]; // pointing to rubiks_face[0][2]
square_ptr++; // pointing to rubiks_face[1][0]
```

Multidimensional Arrays: Go Big or Go Home

2/3-D arrays are common in modelling discrete surfaces/volumes. But in theory you can have any number of dimensions (memory permitting).

```
int square[2][2], cube[3][3][3], hypercube7[7][7][7][7][7][7][7];
```

Thanks to pointers, you can navigate in many ways by indexing.

```
// dimension-wise indexing
int hypercube7_mid = hypercube7[3][3][3][3][3][3][3];
// direct indexing
int cube_last = *(**cube + i*N*N + j*N + k);
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) square[i][j] = i * 2 + j;
}
// arithmetic indexing
for (int p = 0; p < 4; p++) {
    printf("%d ", square[p / 2][p % 2]);
    if (p % 2 == 1) printf("\n");
}
```

Multidimensional Arrays: Indexing & Traversal

How do you traverse down a column? Or along arbitrary dimension(s)? You can fix some of the indexing values by array or pointer arithmetic.

```c
// different dimensions, 3 rows, 4 columns
int rect[3][4];
for (int p = 0; p < 12; p++) rect[p / 4][p % 4] = p;

// first and last columns, and second row
for (int i = 0; i < 3; i++) printf("%d\n", rect[i][0]);
for (int i = 0; i < 3; i++) printf("%d\n", *(rect[i] + 3));
for (int j = 0; j < 4; j++) printf("%d ", (*rect[1] + j));
```

Full code

github.com/rhubarbwu/csc209/blob/master/lectures/lec05/traversal.c

Traversing in an Arbitrary Order

Example:

github.com/rhubarbwu/csc209/blob/master/lectures/lec05/3to3.c

Section 2

Dynamic Memory

Dynamic Memory: Pointers

1. & "returns" the address of any *named* variable, ∗ dereferences any *address*.
2. **Only** for variable declaration, ∗ serves to **identify** variables that are pointers.
3. When reading/writing a pointer variable without dereferencing, you are reading/writing the **address** contained in the pointer.

Casting Pointers

Arbitrary pointers can be cast as typed pointers. What does the following print?

```c
#include <stdio.h>
int main() {
    int x = 0x00616263; char *y = (char *)&x;
    printf("%s\n", y); // cba
    return 0;
}
```

- How? See ASCII Table
- Notice the ordering of the bytes.
- You are expected to understand hexadecimal...

Dynamic Memory: Variables

Local Variables

- Local variables are allocated in the function's *stack frame*.
  - In `gdb`, `backtrace` prints list of stack frames, tracing from currently-executing function up to `main()`.
- When a function returns, its stack frame is deallocated.
  - The freed-up space on the stack can be re-used by a future function that is called.

Global Variables

- Global variables are stored in another region of memory.
  - Includes read-only *string literals*.
- These remain in memory for the entire duration that the program is running.

Dynamically Allocated Variables

- Memory is allocated on the heap, referenced by a pointer.
- Persists on the heap even after the allocating function returns.

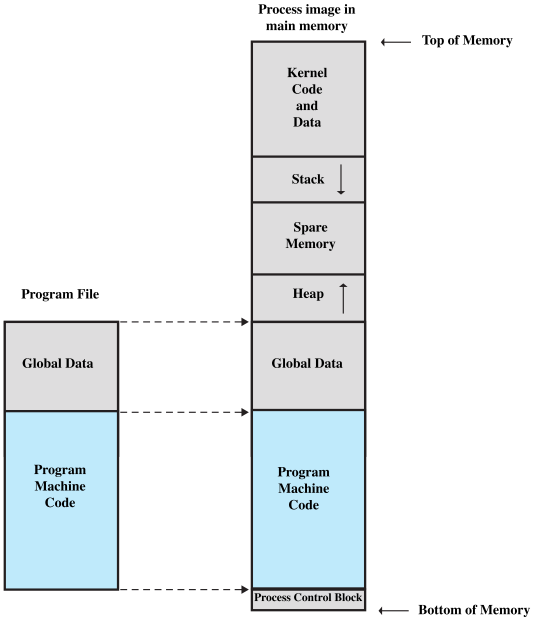

Figure 2: memory model

Dynamic Memory: Allocation

The most basic structure to allocate memory for is an array(list).

In Java

This is done automatically when creating objects.

```
ArrayList createArray() {
  ArrayList a = new ArrayList();
  return a;
}
```

In C

You'll have to be explicit using `malloc`, which takes the number of bytes for the structure and returns a pointer to it.

```
int *createArray() {
  int *a = malloc(sizeof(int)*ARRAY_LEN);
  return a;
}
```

Dynamic Memory: Allocation with malloc

The C library function malloc allocates the requested memory (of size bytes) and returns a void pointer to it.

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

This function returns a void * to the allocated memory or NULL if the request fails.

- The pointer generally needs to be cast to be used as a typed pointer.
- A return value of NULL is often a result of running out of memory.

```
char *str = (char *) malloc(15);
if (str == NULL) exit(1); // probably out of memory
strcpy(str, "tutorialspoint");
printf("String = %s,  Address = %u\n", str, str);
```

Use top/*top to see how much memory your system has. Then try to allocate more.

Source

www.tutorialspoint.com/c_standard_library/c_function_malloc

Dynamic Memory: Allocation with calloc

If you allocated with `malloc`, the memory region might contain garbage. `calloc` is a way to allocate memory while zeroing it too.

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

It has different parameters, allocating space for `nitems` elements, each of `size` bytes.

```
int n = 1000000;
long long *a = calloc(n, sizeof(long long)); // array of n long long
printf("from calloc\n");
for (int i = 0; i < n; i++)
    printf("%lld ", b[i]); // a bunch of 0s
printf("\n");
free(a);
```

`calloc` is not commonly used because most scenarios don't require 0-initialization, and doing so introduces computational costs, especially for large pieces of memory.

Dynamic Memory: Reallocation with realloc

The C library function realloc attempts to resize to size bytes the memory block pointed to by ptr that was previously allocated with a call to malloc, calloc or realloc

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

- If ptr is NULL, a new block is allocated and a pointer to it is returned.
- If size is 0 and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.

```
str = (char *) realloc(str, 25);
strcat(str, ".com");
printf("String = %s,  Address = %u\n", str, str);
```

Source

www.tutorialspoint.com/c_standard_library/c_function_realloc


C programmer: Forgets to call free()
Dynamically-allocated variables:
It's Free Real Estate

Dynamic Memory: Freeing

If we're done with an object, can we reclaim the memory space?

- In Java, the *garbage collector* asynchronously frees up memory when an object is no longer referenced by any variable.
- In Rust, each referenced piece of memory has a lifetime declared at runtime, so there's no garbage to speak of.
- In C/C++, you have to collect your own garbage.
  - Use `free()` to free up allocated space that is no longer being used.
  - Failure to do so results in *memory leaks*, which unnecessarily occupy space.
  - These can occur if you lose references to these piece of memory.
  - Use `*top` (like `htop`) to check memory consumption.
  - Use `valgrind` to detect memory leaks.

Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/malloc.zip`

Dynamic Memory: Pointers to (and Arrays of) Pointers

Let's model the following $\mathbb{R}^K \mapsto \mathbb{R}^3$ linear system $\mathbf{W}^T\mathbf{x} + \mathbf{b}$.

$$\mathbf{x} \in \mathbb{R}^3, \qquad \mathbf{W} \in \mathbb{R}^{K\times 3}, \qquad \mathbf{b} \in \mathbb{R}^3$$

$\mathbf{x}$ and $\mathbf{b}$ are easy since they're vectors.

```
int K = 1000;
double *x = malloc(K * sizeof(double));
double *b = calloc(K, sizeof(double));
```

But $\mathbf{W}$ requires more care. Same goes for batching m inputs as $\mathbf{X} \in \mathbb{R}^{K\times m}$. You must allocate memory top-down.

```
double **W = malloc(3 * sizeof(double *));
for (int i=0; i<3; i++)
    W[i] = malloc(K * sizeof(double));
```

And afterwards, you should free bottom-up.

```
for (int i=0; i<3; i++) free(W[i]);
```

Dynamic Memory: Persistence on the Heap

Stack memory declared in a scope is only accessible therein (including function calls). Otherwise, what's not caught at compile-time can result in runtime memory errors.

```
int *get_stack_ptr() { int *ptr; return ptr; }
int main() {
    while (1) { int x = 0; } x = 1; // compile error
    int *ptr = get_stack_ptr();
    int y = *ptr; // seg fault
}
```

Heap memory persists after function calls return, to be accessed with pointers.

```
int *get_heap_ptr() { int *ptr = malloc(sizeof(int)); return ptr; }
int main() {
    int *ptr = get_heap_ptr();
    int y = *ptr;
    return 0;
}
```

Dynamic Memory: Exercises (Stack)

How big are these? Where do they live? And until when?

```
void fun1(char c) {  // how big is c? where? until when?
    float f;         // how big is f? where? until when?
}
void fun2(int *i_ptr) {}  // how big is i_ptr? where? until when?
int main() {
    int i = 0;          // how big is i? where? until when?
    int *i_ptr = &i;  // how big is i_ptr? where? until when?

    char s[10] = {'h', 'i'};  // how big is s? where? until when?
    char *s_ptr = s;          // how big is s_ptr? where? until when?

    int is[5] = {4, 5, 2, 5, 1};  // how big is is? where? until when?
    fun2(i);
    return 0;
}
```

Dynamic Memory: Exercises (Stack)

How big are these? Where do they live? And until when?

```
void fun1(char c) {  // 1 on fun1 stack until fun1 returns
    float f;         // 4 on fun1 stack until fun1 returns
}
void fun2(int *i_ptr) {}  // 8 on fun2 stack until fun2 returns
int main() {
    int i = 0;         // sizeof(int) on main stack until program ends
    int *i_ptr = &i;  // 8 on main stack until program ends

    char s[10] = {'h', 'i'};  // 10 on main stack until program ends
    char *s_ptr = s;          // 8 on main stack until program ends

    int is[5] = {4, 5, 2, 5, 1};  // 20 on main stack until program ends
    fun2(i);
    return 0;
}
```

Dynamic Memory: Exercises (Heap)

How about these? `malloc` and `free` make an appearance...

```
void fun1(int **i_ptr_ptr) {  // what about i_ptr_ptr?
    *i_ptr_ptr = malloc(sizeof(int) * 7);  // what about *i_ptr_ptr?
}
int *fun2() {
    int *i_ptr;                         // what about i_ptr?
    i_ptr = malloc(sizeof(int));  // what about *i_ptr?
    return i_ptr;
}
int main() {
    int *i_ptr;  // what about i_ptr?
    fun(&i_ptr);
    free(i_ptr);
    i_ptr = fun2();
    free(i_ptr);
    return 0;
}
```

Dynamic Memory: Exercises (Heap)

How about these? `malloc` and `free` make an appearance...

```
void fun1(int **i_ptr_ptr) {                      // 8 on fun1 stack until fun1
    *i_ptr_ptr = malloc(sizeof(int) * 7);  // 28 on heap until free call
}
int *fun2() {
    int *i_ptr;                          // 8 on fun2 stack until fun2 return
    i_ptr = malloc(sizeof(int));  // 4 on heap until free call
    return i_ptr;
}
int main() {
    int *i_ptr;  // 8 on main stack until program ends
    fun(&i_ptr);
    free(i_ptr);
    i_ptr = fun2();
    free(i_ptr);
    return 0;
}
```

Dynamic Memory: Exercises (Addresses)

Try drawing the memory model of the following code.

```
#include "stdio.h"
#include "stdlib.h"
void init(int *a1, int *a2, int n) {
    for (int i = 0; i < n; i++) { a1[i] = i; a2[i] = 2*i+1; }
}
int main() {
    int nums1[3], *nums2 = malloc(sizeof(int) * 3);
    init(nums1, nums2, 2);
    for (int i = 0; i < 3; i++) printf("%d %d\n", nums1[i], nums2[i]);
    free(nums2);
    return 0; }
```

- Heap: `0x23c` to `0x248`.
- Stack for `init`: `0x454` to `0x470`.
- Stack for `main`: `0x474` to `0x48c`.
- Let `??` represent garbage.

Section	Address	Value	Variable
Heap	0x23c		
	0x240		
	0x244		
...			
Stack frame init	0x454		
	0x458		
	0x45c		
	0x460		
	0x464		
	0x468		
	0x46c		
	0x470		
Stack frame main	0x474		
	0x478		
	0x47c		
	0x480		
	0x484		
	0x488		

Section	Address	Value	Variable
Heap	0x23c	1	
	0x240	3	
	0x244	??	
…			
Stack frame init	0x454	0x474	a1
	0x458	0x474	a1
	0x45c	0x23c	a2
	0x460	0x23c	a2
	0x464	2	n
	0x468	~~0~~, ~~1~~, 2	i
	0x46c		
	0x470		
Stack frame main	0x474	0	nums1[0]
	0x478	1	nums1[1]
	0x47c	??	nums1[2]
	0x480	0x23c	nums2
	0x484	0x23c	nums2
	0x488	~~0~~, ~~1~~, ~~2~~, 3	i

Section 3

Structs

Data is often structured, like in classes in object-oriented programming such as in Java, or relational database schemes like SQL.

In C, we use the `struct`, which is a collection of *members*:

```
struct [structure tag] {
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

- Can be dynamically or statically allocated.
- Can declare arrays of structs, pointers to structs…

Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/structs.zip`

Structures: As a Type

A basic declaration can use an anonymous type.

```
struct { float lon, lat; } a, b;
```

However, commonly a `struct` is declared as an explicit type;

```
struct coordinate { float lon, lat; };
float euclidean(struct coordinate a, struct coordinate b);
```

Additionally, you can use `typedef` to create an alias for it.

```
typedef struct coordinate {
    float lon, lat;
} Coordinate;
double euclidean(Coordinate a, Coordinate b);
float manhattan(Coordinate a, Coordinate b);
short time_zone(Coordinate a, Coordinate b);
```

Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/structs.zip`

Structures: Alignment

As memory addresses implied, pieces of memory are aligned on the smallest granularity of 4 bytes. Members of a `struct` are aligned on their largest member.

What is `sizeof(struct student)`? What if we reorder the members?

```c
struct student {
    char school[21]; // 21
    int student_num; // +4 = 25, round to 28
    char name[21]; // +21 = 49, round to 52
}; // 52
```

Consecutive members of the same size can be packed.

```c
struct student {
    int student_num; // 4
    char name[21], school[21]; // +21+21 = +42 = 46, round to 48
}; // 48
```

Alignment is based on order and size of members. Although there exist compiler optimizations that reorder the members to reduce memory footprint.

Structures: Accessing Members

As seen before, members can be directly accessed using the dot . notation.

```
#include "math.h"
typedef struct coordinate { float lon, lat; } Coordinate;
float manhattan(Coordinate a, Coordinate b) {
    return abs(b.lon - a.lon) + abs(b.lat - a.lat)
}
double euclidean(Coordinate a, Coordinate b) {
    double dlon = (double)b.lon - (double)a.lon;
    double dlat = (double)b.lat - (double)a.lat;
    return sqrt(pow(dlon, 2) + pow(dlat, 2));
}
```

Structures: Composition

A `struct` can contain anything; you can nest `struct`'s...

```
typedef struct box {
    Coordinate c1, c2;
} Box;
```

And access inner members...

```
Coordinate c1 = {1.7, -2.3}, c2 = {3.08, 9.81};
Box b = {c1, c2};
int lon1 = b.c1.lon;
int lat2 = b.c2.lat;
```

Structures: Pointers

Passing struct's by value results in deep copies, which are consume a lot of stack memory. Instead, as with arrays, we can pass struct's by pointers.

```
typedef struct coordinate { float lon, lat; } Coordinate;
typedef struct box {
    Coordinate *c1, *c2;
} Box;
```

- Why are we still storing two float values in Coordinate and not float*?

Then we use pointer operators to access through pointers.

```
double range(Box b) {
    return euclidean(*(b.c1), *(b.c2));
}
```

Commonly, the arrow -> notation is used instead for readability.

```
double area(Box b) {
    return abs(b.c2->lon - b.c1->lon) * abs(b.c2->lat - b.c1->lat);
}
```

Structures: Linked-Lists: A Taster

A very common data structure that maintains ordering with easy insertions/deletions is the linked-list (LL). Here's a sample `struct` implementation.

```
typedef struct llnode {
    struct llnode * next;
    int data;
} LLNode;
```

Each `LLNode` holds a pointer `next` to another `LLNode`, so they can refer to each other. More next week...

Arnold's Code

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/linkedList.zip`

Homework

- Your lab exercise this week will be to implement the `ArrayList`.
- A common interview question is reversing a linked-list. Try this too!