Reading 7

Parts of Program Memory and Space:

- Pointers provide a level of indirection to accessing program state.
- Dynamic Memory Allocation allows a program to adjust to changes in size and space needs as it runs, allocating more space as it needs and freeing space it no longer needs.

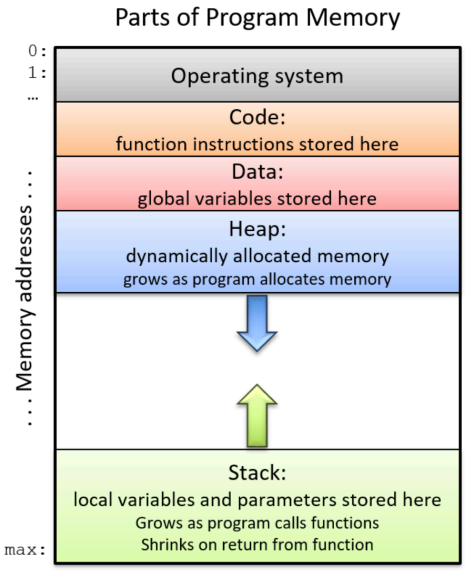
- A variables scope defines when its name has meaning (can be used).
- Global Variables remain permanently in scope.
  - ↳ only use when needed!!

- Local variables and parameters are only in scope inside the function in which they are declared.

- A programs address space (or memory space) represents storage locations for everything it needs in its execution, namely storage for its instructions and data.
  - ↳ The address space can be thought of as an array of addressable bytes; each used address in the program's address space stores all or part of a program instruction or data value (or some additional state necessary for the program's execution).

- Parts of memory diagram!

↳ Note: this diagram is usually drawn more idiomatically with the stack on top!!!



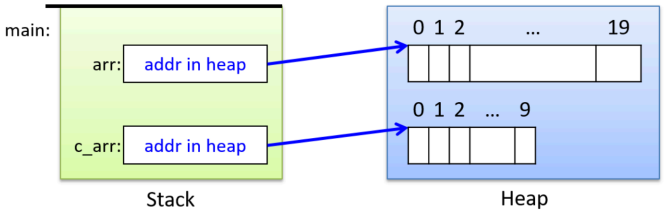
Dynamic Memory Allocation

- heap memory is anonymous memory.
  - ↳ addresses in the heap are not bound to variable names.

- malloc() returns a void * type
  - ↳ or NULL if an error occured.
  - ↳ you should always check if the pointer is NULL before dereferencing it.

- When you free(), it's good practice to set the pointer to NULL too.
- (Ex)!

```c
int *arr;
char *c_arr;

// allocate an array of 20 ints on the heap:
arr = malloc(sizeof(int) * 20);

// allocate an array of 10 chars on the heap:
c_arr = malloc(sizeof(char) * 10);
```



- Because the pointer variable's value represents the base address of the array in the heap, we can use the same syntax to access elements in dynamically allocated arrays as we use to access elements in statically declared arrays

  d_array[i]

- A call to malloc() not only allocates the requested memory bytes, but it also allocates a few additional bytes right before the allocated chunk to store a header structure.
  - ↳ the header stores metadata (such as size) and thats why the call to free only requires the pointer.

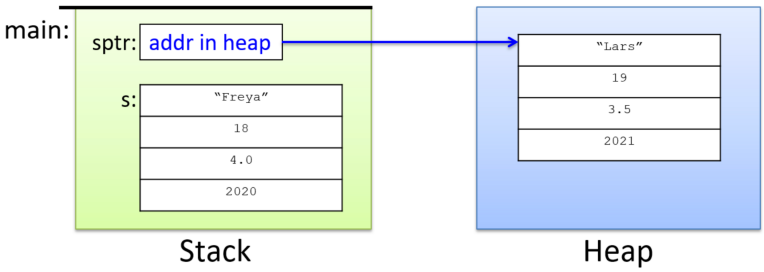
- Since static and dynamic arrays decompose to an address when passed to a function, the same function definition can handle both dynamically allocated and statically declared arrays.

C Structs

- Passing a struct to a function, the parameter gets a copy of the structs value (a copy of all bytes from the argument).
  - ↳ Consequently, changes to the parameter's field values do not change the argument's value.
  - ↳ But arrays do get modified in the argument because you are passing an Address.

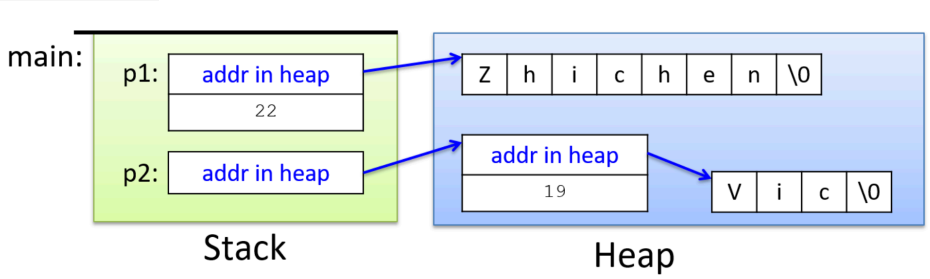
- Remember! (*sptr).grad_yr is equivalent to sptr → grad_yr

- Ex Struct on stack and heap:



- Structs can also be defined to have pointer types as field values. Example:

```c
struct personT {
    char *name;     // for a dynamically allocated string field
    int  age;
};

int main(void) {
    struct personT p1, *p2;

    // need to malloc space for the name field:
    p1.name = malloc(sizeof(char) * 8);
    strcpy(p1.name, "Zhichen");
    p1.age = 22;


    // first malloc space for the struct:
    p2 = malloc(sizeof(struct personT));

    // then malloc space for the name field:
    p2->name = malloc(sizeof(char) * 4);
    strcpy(p2->name, "Vic");
    p2->age = 19;
    ...

    // Note: for strings, we must allocate one extra byte to hold the
    // terminating null character that marks the end of the string.
}
```



- Since p2 is simply a pointer to a struct on the heap, that means you must use (*p2).age or p2→age. p2.age will NOT work.

- There can also be arrays of structs or arrays of pointers to structs:

```c
struct studentT classroom1[40];   // an array of 40 struct studentT

struct studentT *classroom2;      // a pointer to a struct studentT
                                  // (for a dynamically allocated array)

struct studentT *classroom3[40];  // an array of 40 struct studentT *
                                  // (each element stores a (struct studentT *))
```

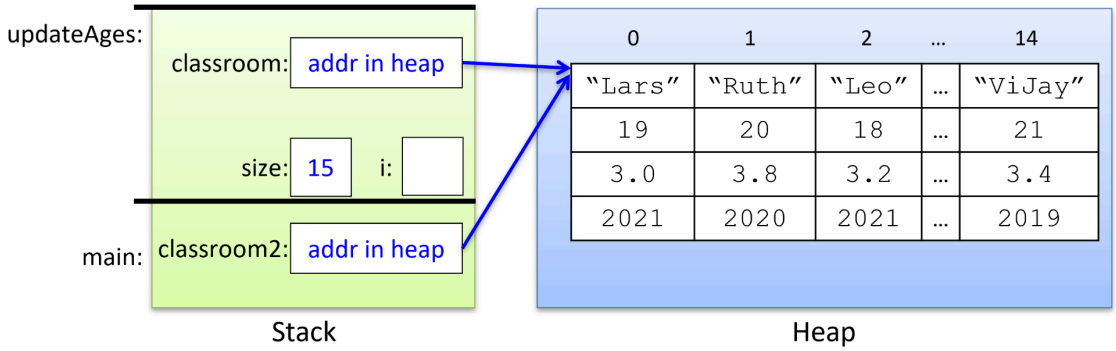
```c
// classroom1 is an array:
//    use indexing to access a particular element
//    each element in classroom1 stores a struct studentT:
//    use dot notation to access fields
classroom1[3].age = 21;

// classroom2 is a pointer to a struct studentT
//    call malloc to dynamically allocate an array
//    of 15 studentT structs for it to point to:
classroom2 = malloc(sizeof(struct studentT) * 15);

// each element in array pointed to by classroom2 is a studentT struct
//    use [] notation to access an element of the array, and dot notation
//    to access a particular field value of the struct at that index:
classroom2[3].year = 2013;

// classroom3 is an array of struct studentT *
//    use [] notation to access a particular element
//    call malloc to dynamically allocate a struct for it to point to
classroom3[5] = malloc(sizeof(struct studentT));

// access fields of the struct using -> notation
// set the age field pointed to in element 5 of the classroom3 array to 21
classroom3[5]->age = 21;
```

- A function that takes an array of type `struct StudentT` as a parameter might look like this:

```
void updateAges(struct studentT *classroom, int size) {
    int i;

    for (i = 0; i < size; i++) {
        classroom[i].age += 1;
    }
}
```

```
updateAges(classroom1, 40);
updateAges(classroom2, 15);
```

↑ Again you can pass static or dynamic arrays.



- As always, the parameter gets a copy of the value of its argument (the memory address of the array in heap memory). Thus modifying the array's elements in the function will persist to its argument's values. Both the parameter and the argument refer to the same array in memory).

Self Referential Structs

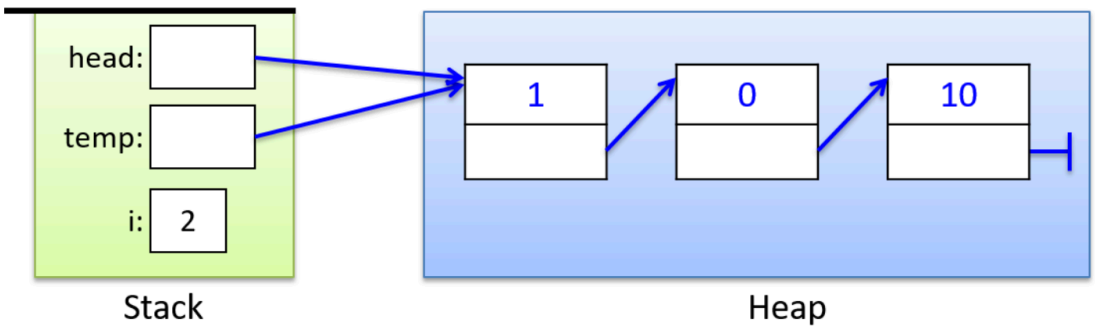
- A struct can be defined with fields whose type is a pointer to the same `struct` type.
  - ↳ These self-referential `struct` types can be used to build linked implementations of data structures, such as linked lists, trees, and graphs.

- A Node struct for nodes is a singly linked list:

```
struct node {
    int data;           // used to store a list element's data value
    struct node *next;  // used to point to the next node in the list
};
```

- Example: Singly-Linked List of 3 elements:

```c
struct node *head, *temp;
int i;

head = NULL;  // an empty linked list

head = malloc(sizeof(struct node));  // allocate a node
if (head == NULL) {
    printf("Error malloc\n");
    exit(1);
}
head->data = 10;    // set the data field
head->next = NULL;  // set next to NULL (there is no next element)

// add 2 more nodes to the head of the list:
for (i = 0; i < 2; i++) {
    temp = malloc(sizeof(struct node));  // allocate a node
    if (temp == NULL) {
        printf("Error malloc\n");
        exit(1);
    }
    temp->data = i;     // set data field
    temp->next = head;  // set next to point to current first node
    head = temp;        // change head to point to newly added node
}
```



Memory API

- Stack memory is sometimes called automatic memory.
- malloc() takes a single parameter of size_t which simply describes how many bytes you need.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

prints 8, since pointers are 8 bytes.

```
int x[10];
printf("%d\n", sizeof(x));
```

prints 40 since the compiler has enough static information to know 40 bytes have been allocated.

- To free allocated memory call free(). free() takes in one parameter: the pointer returned by malloc().

- Ex:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

  - Alternatively, you could use strdup() to make life easier.

- A memory leak occurs when you forget to free memory.
  - ↳ If program is long, you could run out of memory.
  - ↳ Even in modern languages (i.e. Python), the garbage collector only free's chunks of memory if you don't have a reference to it.

- Good practice to free() every byte you allocate.

- If you free memory before you are finished using it, you have made a mistake called a dangling pointer.

- For short-lived programs you can get away with not calling free(), since the OS reclaims all memory used by a process after its done running.
  - ↳ THIS IS BAD PRACTICE, YOU SHOULD ALWAYS FREE!!!

- If you free() memory twice, the result is undefined, so only free once!!

- Other functions!
  - ↳ calloc() → allocates and initializes everything to 0
  - ↳ realloc() → makes a larger memory region and copies the existing data into it.