# Week 6

### Part 1

* Pointers can be copied - The pointer(s) will then point to the **same memory slot** (same value)
  + int \*p = &a;  
    int \*q = p;
* Pointers can be passed as arguments - modification to the pointer value will then be reflected on the original variable
  + int change(int \*x) {  
     \*x = 5; //set value of original var to 5  
    }  
    int var = 3; change(&var); // & get address of var  
    printf(“%d”, var); // 5
* Functions can also return pointers:
  + int\* max(int\* a, int\* b) {  
     if (\*a > \*b) {  
     return a;  
     }   
     return b;  
    }
  + **Important: Never return a pointer to a local variable. All variables are terminated after its scope ends.**

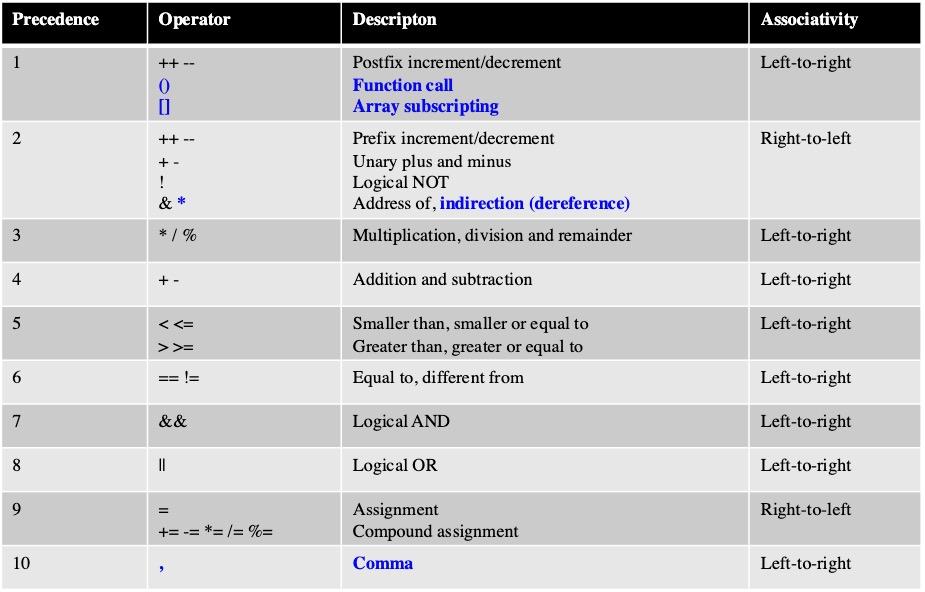
### Part 2 - arrays & pointers combination / arithmetics with pointers

* Pointers can point to array elements e.g. &a[i] points to the i-th element of array a.
  + Note that \* and & terminate each other e.g. \*&a[i] = a[i]
  + Use \* to deference those pointer - you then get the element’s value!
* Supported operations:
  + **Adding** an integer x to a pointer → the address of element x index ahead.
    - int \*p = &a[0];  
      p += 3  
      printf(“%d”, \*p); // value of a[**3**]
  + **Subtracting** an integer x from a pointer → the address of element x index before.
    - int \*p = &a[3];  
      p -= 2  
      printf(“%d”, \*p); // value of a[**1**]
  + **Subtracting two pointers:** You get their distance. You might recognize that this operation make sense only for pointers to the **same array,** since only elements of the **same array** are guaranteed to be **continuous**.
    - int \*p = &a[3];  
      int \*q = &a[1];  
      printf(“%d”, p - q); // 2 - distance between 2 index
* Comparing pointers: >, <, ==, >=, <= can be used to compare pointers of the **same array,** undefined behavior otherwise.
* Example: Looping through array with pointers:
  + int a[10] = {...};  
    int \*p;  
    for (p = a; p <= &a[9]; p++) {  
     printf(“%d”, \*p); // print all elements of a!  
    }

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Week 7

### Precedence of operators



* Pre & post ++, --:
  + ++i: Calculate i = i + 1 first, then use the value of i + 1
  + i++: Use the value of i first, then i = i + 1
* Combining the \* and ++ Operators:
  + \*p++, or can be interpreted as \*(p++) can be used for increasing & -- for decreasing a pointer's address.
* Using array name as a pointer:
  + Name of an array can be used as a pointer to the first element in the array.
  + If a is an array, in general, a + i = &a[i] – Both represent a pointer to element i of a
* Using array name in for loops:
  + for (p = &a[0]; p < &a[N]; p++)   
     sum += \*p;
  + for (p = a; p < a + N; p++)   
     sum += \*p;
* **Important: Never use array names for iteration:**
  + while (\*a != 0)   
     a++; /\*\*\* WRONG \*\*\*/
* Pointer argument as array:
  + We can parse an array int a[] to a function func using pointer:
    - void func(int \*a); // initializing function  
      func(a); // calling function
  + NOTE: It is better to use \*a for array argument than a[]

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

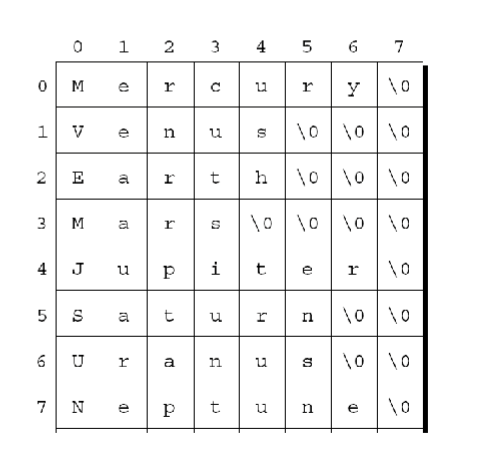
* Segmentation fault:
  + It means your program has attempted to access an area of memory that it is not allowed to access.
  + It attempts to stomp on memory ground that is beyond the limits that the operating system (e.g., Unix) has **allocated** for your **program**.
  + Your program is only allowed to touch memory that belongs to it (**allocated to the variables**).
* Common causes of segfault:
  + Forgetting to use & on the arguments to scanf
  + Forgetting to use & on the arguments to function as pointers
  + Incorrect use of the & (address of) and \* (indirection) operators
  + Failure to initialize a pointer before accessing it

# Week 8 - Strings

### Part 1

* Strings in C:
  + Two types of strings:
    - Constants, initialized using “ ”, ex: “Hello World”
    - Variables
  + There is no string type in C, instead an array of chars with null terminator (‘\0’) marks the end.
* String literals:
  + Sequence of characters enclosed with double quotes, ex: “Hello World”
    - When too long, you can use backlash \ to continue in a new line  
      printf(“Fucking \  
       Dog Shit”);
  + String literals are stored as follow: Ex: “abc” is stored in memory as   
    [ ‘a’ | ‘b’ | ‘c’ | ‘\0’ ]
  + Null character:
    - Character with value zero
    - Represent escape sequence ‘\0’
    - Empty string “” is stored as a **single null character** in memory
  + Use pointer for string literal:
    - String literal is stored as an array
    - char \*a = “String 1”;
      * This should not be modified, can crash or cause undefined behaviors
* String variables:
  + Any one-dimensional array of characters can be used to store a string.
  + char s[] = “Hello World’;
  + The compiler will automatically add a ‘\0’ at the end of the string
  + String variables must have enough space for a **null terminator**. Without a null terminator, our program might cause unexpected behaviors or crash.
* Accessing characters in a string:
  + Array subscripting:
    - char s[] = “Hello World”; // s[4] = ‘o’
  + Pointer arithmetic
    - char s[] = “Hello World” // \*(s + 6) = ‘W’
* Reading and Writing strings:
  + printf: Use %s for writing strings. printf will write out characters until it reaches the **null terminator**.
    - char str[] = "Are we having fun yet?";   
      printf("%s\n", str); // “Are we having fun yet?”
  + scanf: Use %s for reading strings. Ex:
    - char str[20];   
      scanf("%s", str); // Input: “sponsorship”  
      // str will now be “sponsorship\0”
    - Note: str is treated as a pointer, no need for &
  + scanf always store a ‘\0’ at the end of the string.
* **CAUTION FOR SCANF**:
  + scanf only reads characters into an array and **has no way to detect when it’s full.** It may store characters past the end of the array, causing undefined behavior.
  + scanf can be made safer by using the conversion specification %ns instead of %s, n is an integer indicating the maximum number of characters to be stored.
    - scanf("%20s", str);
* **IMPORTANT**: When scanf is called, it skips white space, then reads characters and stores them in str until it encounters a white-space character (‘ ’, ‘\n’)
* **IMPORTANT**: Differences between NULL pointer & null character
  + NULL pointer: a pointer points to nothing, Address = 0x0  
    Ex: int \*p = NULL;
  + Null character: ‘\0’, use for ending a string
* To read a full line of input, implement your own function, for instance the read\_line() function provided in slides.

### Part 2 - String library & Command line arguments (CL args)

* Import: #include <string.h>
* Functions:
  + strlen(s1): return string’s **actual** length (excluding null terminator!)
    - strlen(“abc”); // 3
  + strcpy(\*s1, \*s2): copy s2 to s1, return pointer to s1.
    - strcpy(str2, “abcd”); // str2 = “abcd”  
      strcpy(str1, str2); // str1 = “abcd”
    - Note: You can’t copy directly with = i.e. str1 = str2; does **NOT** work!
  + strcat(\*s1, \*s2): add s2 to the end of s1, return pointer to s1
    - char str1[10] = “abcd”;  
      char str2[10] = “efgh”;  
      strcat(str1, str2); // str1 = “abcdefgh”
  + strcmp(\*s1, \*s2): return **< 0** if s1 < s2, **0** if s1 == s2, and **> 0** if s1 > s2. Usually the absolute difference between the first different characters.
    - strcmp compares **ASCII value** of each character, if all characters match up to a point, it compares the **length**.
* Array of strings
  + Multidimensional arrays: E.g. int a[3][9] declares a 2D array with **3** rows and **9** columns.
    - To access row i, col j, we can type a[i][j]
  + To create an array of strings with initial values, use a 2D array of characters:  
    char strings[][8] = {“Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"}
    - The number of **rows** can be **omitted** (or specified, whatever) but the number of **columns** must be **specified**.
  + We can also create array of string literals, but are **read-only**:  
    char \*strings[] = {“Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"}
  + To access a string, use subscripting as usual e.g. strings[2] → “Earth”

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* To access Command Line Arguments, main() should be written as such:  
  int main(int argc, char \*argv[])

{

...

}

* + argc is the number of arguments
  + argv is the array of arguments. Note that argv[0] always have the name of the program (hence argc is always >= 1)
* How to enter CL args: E.g. ./a.out 8 24 52 81 23 → argv[0] = “a.out”, argv[1] = “8”, argv[2] = “24”, …
  + Note that all arguments are **strings**!