

CMPT 125: Introduction to Computing Science and Programming II

Spring 2023

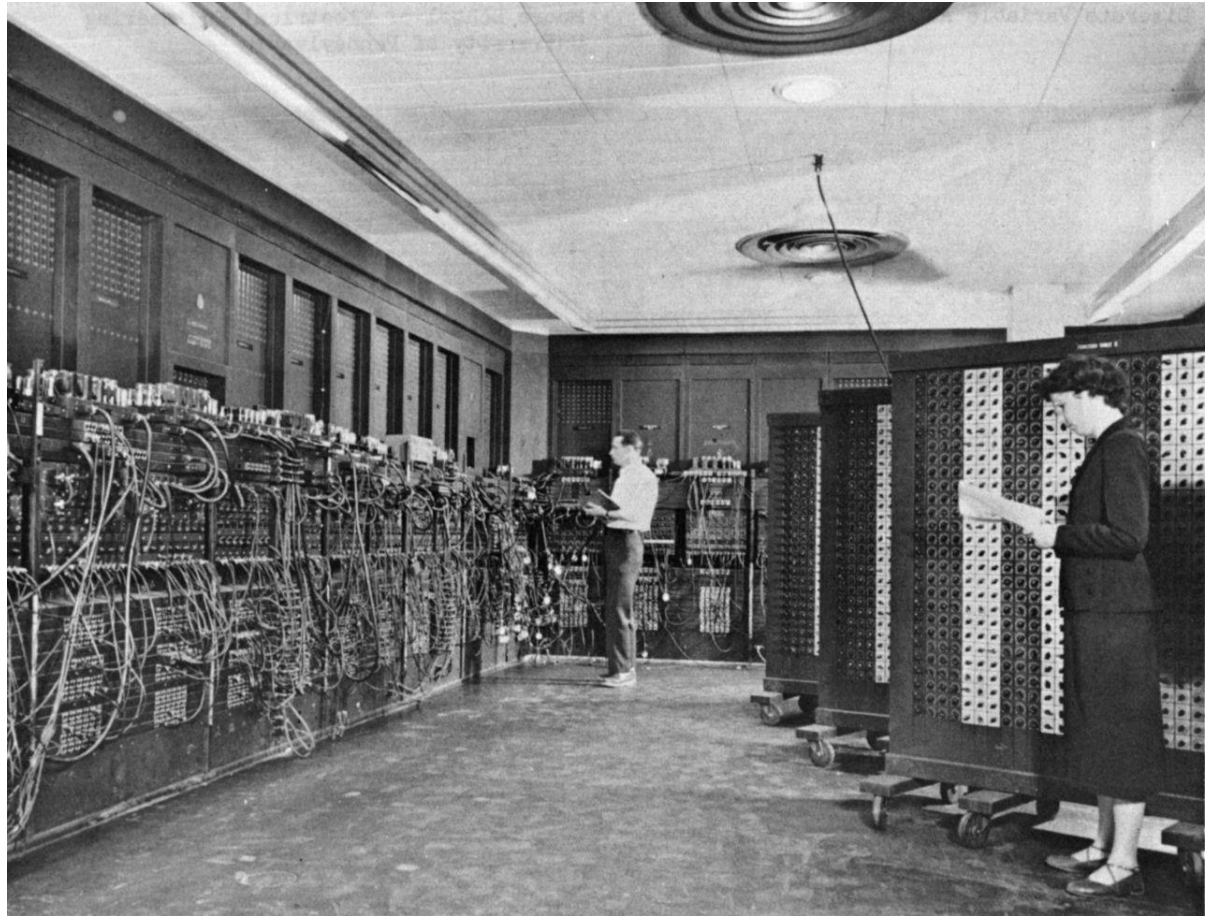
Week 2: More arrays, constants, and stdin

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Fact of the day

The first electronic computer ENIAC was built in 1945 and weighed more than 27 tons



By Unknown author - U.S.Army Photo, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=55124>

Recap from Last Lecture (I)

- Built-in (aka primitive) types in C
 - Different pre-defined types with different required sizes in memory, and different ways (format) to print them to screen
- Pointer variables
 - A type of variables that store memory location (address) of other variables
 - * for declaration or dereference, & to get address of a variable
 - Allow functions to modify parameters, help with space efficiency, enable use of arrays and strings in C

Recap from Last Lecture (2)

- More arrays
 - A way to use 1 name and indexes to store multiple values of the same type (taking up contiguous memory space)
 - Essentially a pointer variable (constant so you can't change it) storing the address of the first element
- Constant variables
 - Used to indicate the variables' values cannot be changed throughout the lifetime of the program
- Strings
 - Sequences of characters
 - Represented in C by character (char) arrays that ends has a null character `\0` place at the end

Review from Last Lecture (I)

- What will be the output of these code pieces?

```
int i;  
int array[5] = {0, 1, 8, 2, 18};  
printf("array[5] = %d\n", array[5]);
```



random!

```
array[5] = 67108865
```

```
int x = 5;  
int* ptr1 = &x;  
x = 8;  
printf("*ptr1 = %d\n", *ptr1);
```



```
*ptr1 = 8
```

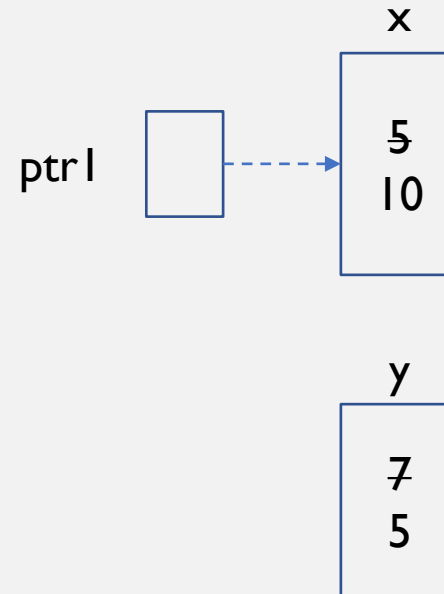
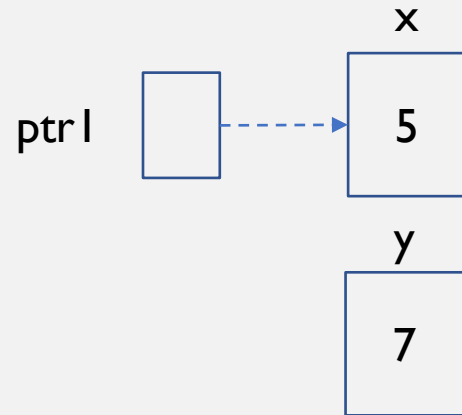
```
int x = 5;  
int y = 7;  
int* ptr1 = &x;  
  
y = *ptr1;  
*ptr1 = 10;  
  
printf("x = %d\n", x);  
printf("y = %d\n", y);
```



```
x = 10  
y = 5
```

Review from Last Lecture (2)

- A closer look at code piece 3



```
int x = 5;
int y = 7;
int* ptr1 = &x;

y = *ptr1;
*ptr1 = 10;

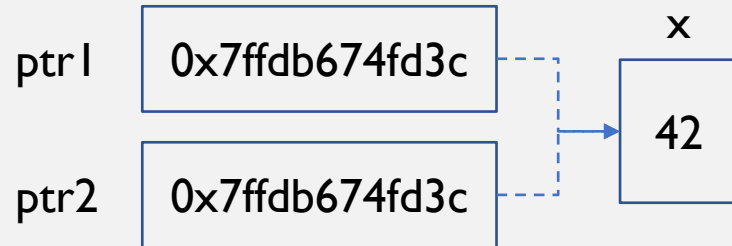
printf("x = %d\n", x);
printf("y = %d\n", y);
```



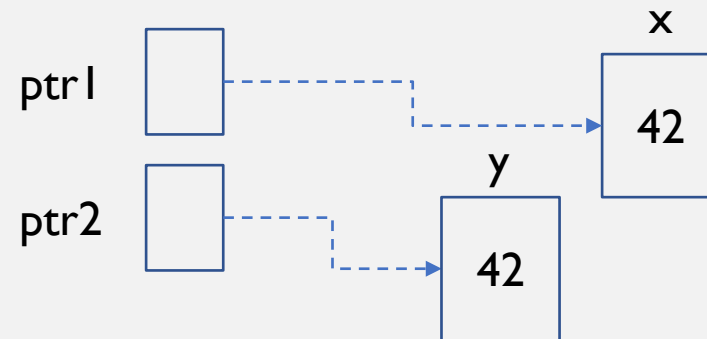
```
x = 10
y = 5
```

Review from Last Lecture (3)

- Suppose ptr1 and ptr2 are both pointers, what is the different between
 - ptr1 = ptr2;
 - The ptr1 variable is now having the same address stored by ptr2, i.e., they are now pointing to the same address
 - *ptr1 = *ptr2;
 - The variable pointed to by ptr1 is now having the same value stored by the variable pointed to by ptr2



ptr1 = ptr2

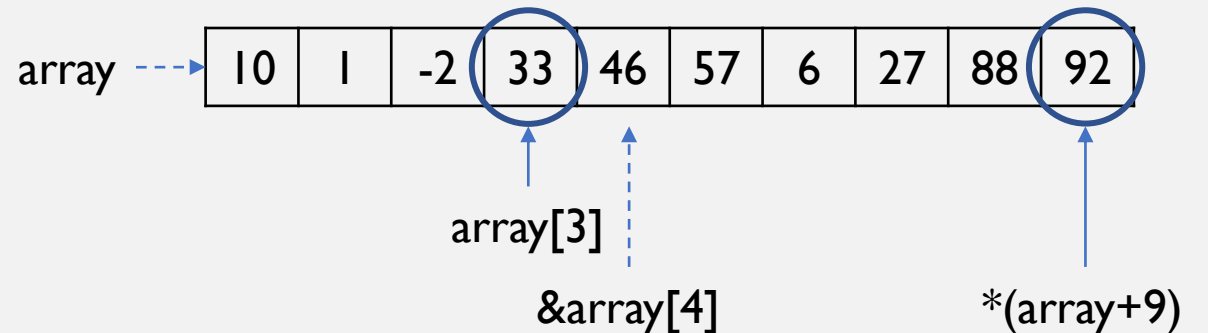


*ptr1 = *ptr2

it is also possible for ptr1 and ptr2 to be pointing to the same address

Review from Last Lecture (4)

- Describe what each of these expressions mean, if array is an int array of size 10, or explain if it is incorrect:
 - array[3] refers to the value stored as the 4th element
 - array[10] refers to the value stored as the 11th element, which is invalid
 - *(array+9) refers to the value stored as the 10th element
 - &array[4] refers to the address of the 5th element
 - array++ is invalid as it is trying to increment the value store in array, which is a const pointer



Review from Last Lecture (5)

- What's wrong with the following code pieces?
 - Left: the function foo is expecting an int pointer type variable (int *), but when it is called the address of a constant int (const int) variable is passed to it (type mismatch)
 - Right: the const_ptr variable is expecting an int pointer whose value can be modified, but it is assigned to a const int variable (type mismatch)

```
void foo(int* a) {  
    //do something  
}  
  
int main() {  
    const int x = 5;  
    foo(&x);  
}
```

```
const int x = 1;  
int* const const_ptr = &x;
```

Today

- Strings and string functions
- Immutable strings
- Reading user input
 - The scanf function
- 2D arrays
 - Syntax and usage

The string.h Library

- Since we have a standardized way to store strings in C, we also have a few standard functions to handle strings
 - `strlen` – returns the number of characters in the C string (excluding the null character `\0`)
 - `strcpy` – copies the C string into another array pointed to by a char pointer
 - `strcat` – appends a copy of the C string to another C string
 - `strcmp` – compares two C strings and returns an integer indicating if they have the same sequence of characters
 - `strstr` – returns a pointer to the first occurrence of a C string within another C string
 - ...etc.
- All these functions **assume that** the parameters are properly created C strings (character arrays terminated by the null character `\0`)

The **strlen** Function – `strlen(const char* str)`

- Returns the length (number of characters) of the string, excluding the null character `\0`
- Knows where to stop counting when it sees the null character `\0`
- Does not always mean the size allocated to the actual char array

```
//this means if used to store a Cstring it can have at most 19 characters  
char str1[20] = "Hello World!";  
printf("str1 is %s has a length of %d\n", str1, (int)strlen(str1));  
  
str1[7] = '\0';  
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));
```



```
str1 is Hello World! has a length of 12  
str1 is Hello W and has a length of 7
```

The **strcpy** Function – strcpy(char* dest, const char* src)

- A convenient way to duplicate C strings (length of the destination C string must be sufficient to copy)
 - Knows when to stop copying when it sees the null character \0 (will also copy that over)

```
char str1[] = "Hello";  
char str2[40]; //make sure there is enough space  
char str3[40];  
  
strcpy(str2, str1);  
printf("%s\n", str2); //prints Hello  
strcpy(str3, "World");  
printf("%s\n", str3); //prints World
```

- A variant of this function is the strncpy function that only copies a part of the C string from the source

The **strcat** Function – `strcat(char* dest, const char* src)`

- A way to create a new C string by combining 2 C strings (length of the target C string must be sufficient to copy)
 - Knows where to begin and when to stop appending when it sees the null character `\0` (will also copy that over)

```
//this means if used to store a Cstring it can have at most 19 characters
char str1[20] = "";
printf("str1 is %s has a length of %d\n", str1, (int)strlen(str1));

char str2[] = "Hello";
strcat(str1, str2);
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));

strcat(str1, " World!");
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));
```



```
str1 is  has a length of 0
str1 is Hello and has a length of 5
str1 is Hello World! and has a length of 12
```

- A variant of this function is the `strncpy` function that only copies a part of the C string from the source

The **strcmp** Function – strcmp(const char* str1, const char* str2)

- When comparing two C strings, it seems to make sense to write the code `str1 == str2`, but it is wrong!

```
char* password = "ABBBAC";  
char* guess = "ABC";  
  
// WRONG approach!  
if (password == guess) {  
    printf("Guessed it right!\n");  
}
```

- Instead, use **strcmp**

```
int strcmp ( const char * str1, const char * str2 );
```

In some C versions they just return -1, 0, or 1

- Returns 0 if contents of both C strings are equal
- Returns non-zero upon the first mismatch: < 0 if the mismatch character in str1 has a lower value than in str2, >0 otherwise
- With this information you can sort C strings in alphabetical order

The **strstr** Function – `strstr(const char* str1, const char* str2)`

- Looks for the presence of a C string in another C string
 - Stops at the first occurrence and returns the pointer to it

```
char str1[] = "This is a very very important message.";
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));

char *replace = strstr(str1, "very");

strncpy(replace, "VERY", 4); //indicate only need to copy 4 characters over
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));
```

strcpy doesn't work...
can you tell why?

str1 is This is a very very important message. and has a length of 38
str1 is This is a VERY very important message. and has a length of 38

- A variant of this function is the `strchr` function that locates the first occurrence of a character (including `\0`)

The strstr Function (Cont'd)

- What if a string is not found inside another string when using strstr?
 - Returns a **null pointer** if not found (null is a special address value that means “nothing”)

```
char str1[] = "This is a very very important message.";
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));

char *key = "super";
char *match = strstr(str1, key);

if (match == NULL) {
    | printf("The string %s is not found in %s\n", key, str1);
} else {
    | printf("There is at least 1 occurrence of the string %s in %s\n", key, str1);
}
```

```
str1 is This is a very very important message. and has a length of 38
The string super is not found in This is a very very important message.
```

Tips on Using C Strings

- Remember, they are just character arrays with the null character `\0` to indicate end of the sequence
- Include the `string.h` library when using the standard functions in C
 - Make sure the C strings you pass to those functions are properly formed (there is a `\0` at the end)
 - Make sure the C strings you pass to those functions that change them have enough space for the characters
 - Check the return values of these functions, some will give you insight of the function
- Look up the reference for the `string.h` library, there are many other useful functions

Immutable Strings

- We have learned that C handles strings using character arrays with a null character automatically added to the end
- There are 3 main ways to create a C string
 - `char str1[5] = {'w','o','r','d','\0'};` //array of 5 initialized with values
 - `char strSize10[10] = {'w','o','r','d','\0'};` //only initialized 5 values, but can be used to store up length ≤ 9
 - `char str2[] = {'w','o','r','d','\0'};` //same as above with size 5 implicitly set
 - `char* str3 = "word";` //using a string literal
- The third way is however different in how it is stored in the memory as an immutable string
 - `str3[3] = 'k';` //causes a segmentation fault that crashes the program due to attempt to modify a read-only memory location
 - `printf("%c\n", str3[3]);` //OK because it is just reading the value from the memory location

Quick Check

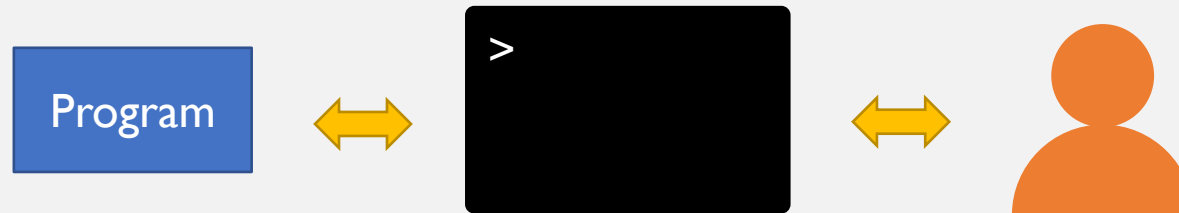
- Suppose we have 3 C strings:

```
char* word1 = "Hello";  
char word2[6] = {'H', 'e', 'l', 'l', 'o', '\0'};  
char word3[] = "Hello";
```

- Which of the following work?
 - strcpy(word1, "hey");
 - strcmp(word1, word2);
 - word3[3] = 'p', word3[4] = '\0';
 - word2[0] = "h";

Interactive Programs

- To be interactive, a program needs some way to communicate with the user
- This means the program needs to be able to get input from the user and provide output to the user
 - In this course we are mostly using the console (i.e., display+keyboard) as the medium



- We have learned how to provide output to the user via the function `printf`:

```
char str1[] = "This is a very very important message.";
printf("str1 is %s and has a length of %d\n", str1, (int)strlen(str1));
```


Getting Input Using scanf

- Reads formatted input from standard in (stdin), default is the keyboard

```
int scanf(const char *format, ...)
```

- The `format` determines the text to be read, with optional embedded format specifiers indicating how each value should be read, puts the value into locations pointed by the additional arguments

```
int x = 0, y = 0;
printf("Type 2 integers separated by a space, then press enter ");
scanf("%d %d", &x, &y);
printf("I have just read 2 integers: %d and %d\n", x, y);
```

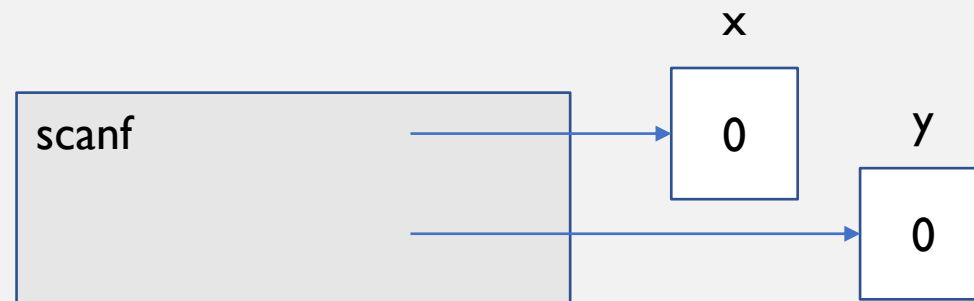


```
Type 2 integers separated by a space, then press enter
5 10
I have just read 2 integers: 5 and 10
```

Why Does scanf need the Addresses?

- Recall the **call-by-value** and **call-by-reference** behaviours
 - `scanf` needs to be able to **modify the variable directly** in order to let its caller to use the value

```
int x = 0, y = 0;  
printf("Type 2 integers separated by a space, then press enter ");  
scanf("%d %d", &x, &y);  
printf("I have just read 2 integers: %d and %d\n", x, y);
```



Another scanf Example

- You can use different **format specifiers** in scanf to read in different types, just like in printf for printing different types

```
char name[20];
int age;

printf("Enter your name: ");
scanf("%s", name); //&name[0]

printf("Enter your age: ");
scanf("%d", &age);

printf("%s is %d years old\n", name, age);
```

```
Enter your name:
Ben
Enter your age:
3
Ben is 3 years old
```

In some environments a
newline is automatically added
for scanf, but not always!

scanf Gone Crazy??

- Try this common way of getting multiple user inputs: input a few non-zero positive numbers, then the letter 'a'

```
int inputNum = 0;
do {
    printf("Give me a positive number (0 to quit): ");
    scanf("%d", &inputNum);
} while(inputNum != 0);
```

- This happens because scanf expects a digit and when it reads in a non-digit, it leaves the non-digit at the stream
 - next time the loop repeats the same thing happens
- One way to fix this is to make use of the return value of scanf, which shows how many values are successfully read

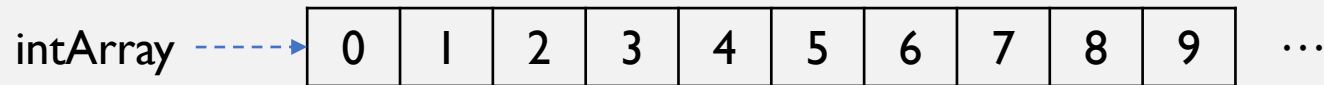
use a variable to store the return
value and compare to 0

```
printf("Give me a positive number (0 to quit): ");
//scanf_ret = scanf("%d", &inputNum);
} while(scanf("%d", &inputNum) != 0 && inputNum != 0);
```

Arrays (from Last Lecture)

- We started using 1 variable to store 1 value at a time, works ok when the program is small, but:
 - Gets overwhelming when we have lots of values to store
 - In many cases a bunch of variables are related (e.g., height of the class, names of songs in an album)
- Arrays allows us to use 1 name to store a bunch of related values, called elements (hence the same type)

```
int anInt;           // a single integer  
int intArray[100];  // an array of 100 integers
```



An array of size 100 with the name **intArray**

2D Arrays (I)

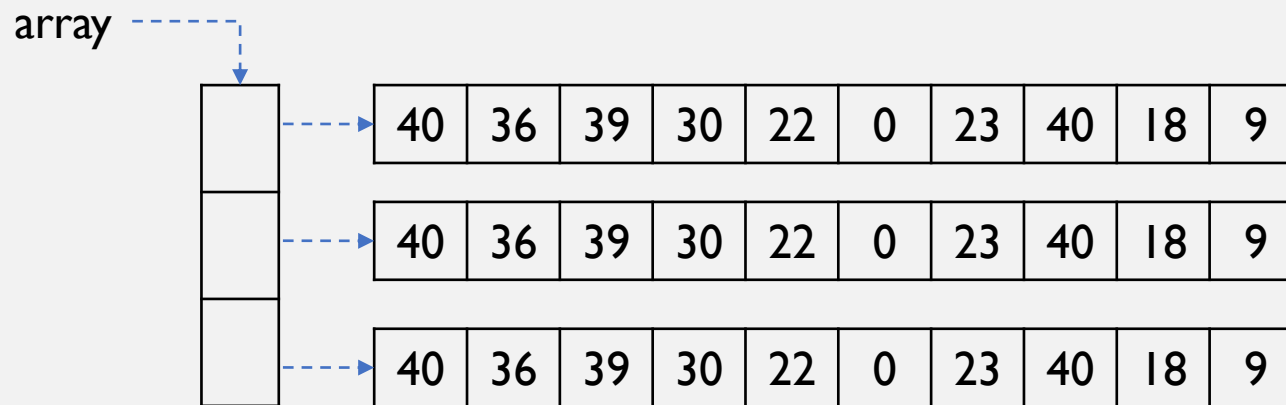
- The array in the previous page is called 1D array because the elements are stored in a linear manner
 - It makes sense in many cases, like marks for all students in an assignment, but what if we have more than 1 assignment?

assignment1marks	----->	40	36	39	30	22	0	23	40	18	9	...
assignment2marks	----->	50	44	30	24	0	10	43	40	28	36	...
⋮												

- There is a way to group arrays of the same type (often same size) under the same name: 2D array
 - Examples include: pixels in a 2D image, temperatures through a day in different places, 2D matrix representation

2D Arrays (2)

- Essentially, a 2D array is a **pointer to an array of pointers** (recall a 1D array is a pointer to a group of elements)



A 2D array with 3 rows and 10 columns (3-by-10) with the name **array**

2D Arrays (3)

- Similar to 1D arrays, the syntax is type, name, and size (in 2 dimensions, first row, then column)


```
int anInt; //a single integer
int intArray[100]; //a 1D array of 100 integers
int int2DArray[4][25]; //a 2D array of 100 integers also, in 4 rows of 25
```

- To initialize a 2D array with values, use a “layered” approach
 - For small arrays: `int shortIntArray[2][3] = {{0, 0, 0}, {0, 0, 0}};`
 - For large arrays use a nest-ed for-loop:

```
int longIntArray[20][30];
for (int r=0; r<20; r++) {
    for (int c=0; c<30; c++) {
        longIntArray[r][c] = 0;
    }
}
```

2D Arrays (4)

- To access an element in a 2D array, first indicate which row you are referring to, then which column
 - Remember indexes all start with 0 and end with `length - 1`

`a[3][4]` 

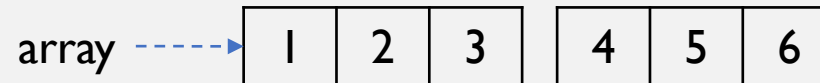
<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

- *note: this row-first-then-column notation is what we call a “row-oriented 2D array”, you might also learn about the “column-oriented 2D array” where you specify which column first

Passing 2D Arrays into Functions

- Similar to 1D arrays, you can pass a 2D array as a parameter into function, pass-by-reference style
- You need to specify at least the column dimension of the array (row dimension is optional)
 - This allows the program to calculate how far to jump to reach the correct row, e.g., if you have a 2-by-3 2D array:

```
void reduceByOne(int arr[][3], int r) {  
    for(int i=0; i<r; i++) {  
        for(int j=0; j<3; j++) {  
            arr[i][j] -= 1;  
        }  
    }  
}
```



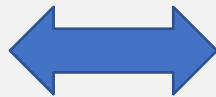
if looking for arr[1][X],
jump 1*3 steps

This way allows the function to work with any 2D arrays that have 3 columns. Look for other ways that doesn't have this constraint! Hint: requires putting the dimensions as the first 2 parameters

Matrices And 2D Arrays

- In mathematics, **matrices** group numbers/symbols/expressions into rectangular arrays to outline their relationships
 - This grouping can be represented in 2D arrays of integers/characters/strings

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$



0	1
2	3
4	5

- Matrix operations such as addition, multiplications, row reductions, can be performed on 2D arrays using for/nested-for-loops

Today's Review

- Strings and string functions
 - Ways to create strings in C, some are created as immutable
 - The null pointer (used in C strings to represent “not found”)
- Reading user input
 - The scanf function reads in user input in a given format
- 2D arrays
 - A way to store multiple 1D arrays

Homework!

- Continue to read the sections in Ch. 2 & Ch. 5 of the Effective C book from last lecture
- Investigate if this statement is valid:
 - `char* str5 = {'w', 'o', 'r', 'd', '\0'};`
- Another way to access a 2D array is to consider the fact that it is simply a pointer to an array of pointers, investigate what this expression is accessing in a 4-by-5 2D array:
 - `(*(array+2))[4]`
- Like 1D arrays, 2D arrays can also be passed to a function as parameters. Investigate how to do that
- Look up the function `scanf` provided by `stdio.h`
<http://www.cplusplus.com/reference/cstdio/scanf/>