

# Lecture 6: Character Operations and Arrays

Sarah Nadi

[nadi@ualberta.ca](mailto:nadi@ualberta.ca)

Department of Computing Science  
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]



## Agenda

- char type
- Character operations & character-handling functions
- One-dimensional arrays
- Multi-length arrays
- Variable-length arrays
- Quick intro to string functions

## Readings

- Textbook Chapter 7.3
- Chapter 8

# Characters & their Operations

# Character Types

```
char ch;  
int i;  
ch = 'A'; //variable ch is assigned the value of A  
i = ch; //variable i has the value 65 (ascii value of A)
```

- You will be dealing with character types in Assignment 1
- Characters are treated as integers, which means that all operations on integers can be done with characters
- Values of type `char` are machine dependent, because there are different character set. ASCII is the most popular character set. See <http://www.asciitable.com/>
- For reading/printing characters, use the conversion specifier `%c`

# Character-handling Functions

- Convert case using `toupper` (need to `#include <ctype.h>`) :

```
ch = toupper(ch);
```

- `toupper(char)` simply implements:  

```
if (ch >= 'a' && ch <= 'z')  
    ch = ch - 'a' + 'A';
```

# Reading and Writing Characters

- Take care of white space (`scanf ("%c", &ch);` vs. `scanf (" %c", &ch);`)
- Read a single character from the keyboard using `ch = getchar();`
- Write a single character to the screen by `putchar(ch);`

demo: read\_char.c

# Idioms to Skip the Rest of the Line

```
do {  
    scanf("%c", &ch);  
} while (ch != '\n');
```

```
while ((ch = getchar()) != '\n') ;
```

```
do {  
    ch = getchar();  
} while (ch != '\n');
```

```
while (getchar() != '\n') ;
```

# Reading char from a Stream

```
/* fgetc example: money counter */
#include <stdio.h>
int main ()
{
    FILE * pFile;
    int c;
    int n = 0;
    pFile=fopen ("myfile.txt","r");
    if (pFile==NULL) fprintf (stderr, "Error opening file\n");
    else
    {
        do {
            c = fgetc (pFile); //If we are the end of file, fgetc returns EOF
            if (c == '$') n++;
        } while (c != EOF);
        fclose (pFile);
        printf ("The file contains %d dollar sign characters ($).\n",n);
    }
    return 0;
}
```

Adapted from <http://www.cplusplus.com/reference/cstdio/fgetc/>



# Arrays

# Variables

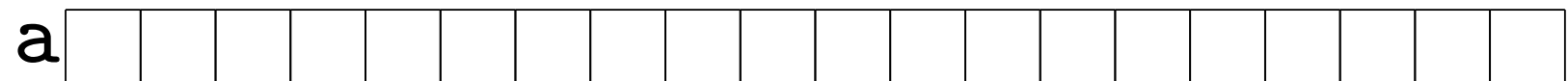
- *Scalar* variables hold a single data item (e.g., `int x; float y; char c`)
- `Aggregate` variables store a collection of values
- C has two kinds of aggregates:
  - ▶ arrays
  - ▶ structures

# Variables

- *Scalar* variables hold a single data item (e.g., `int x; float y; char c`)
- `Aggregate` variables store a collection of values
- C has two kinds of aggregates:
  - ▶ arrays
  - ▶ structures

# One-dimensional Arrays

- Data structure that contains a number of data values **of the same type**
- Each value in the array is called an *element*
- Each element can be accessed by its position within the array.
- Declaring an array of type int with 20 elements: `int a[20];`



# One-dimensional Arrays

*Cont'd*

- The number of elements in an array is called its *length*
- Length must be a **constant** integer (e.g., 10, 1+4) or a macro that gets preprocessed to an integer (e.g., LENGTH where we have `#define LENGTH 10`)
- Conceptually, elements of an array are arranged consecutively in memory
- Index/subscript starts at 0
- `a[2]` is:
  - ▶ an lvalue (i.e., object stored in computer memory)
  - ▶ the 3rd element of array `a`
  - ▶ has type `int` and can be treated as an `int` value

# Array Examples

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 0; i < N; i++)
        a[i] = 0;

    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    int sum = 0;

    for (i = 0; i < N; i++)
        sum += a[i];

    return 0;
}
```

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 1; i <= 2 * N; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i,
               a[i]);
    }

    return 0;
}
```

# Array Examples

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 0; i < N; i++)
        a[i] = 0;

    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    int sum = 0;

    for (i = 0; i < N; i++)
        sum += a[i];

    return 0;
}
```

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 1; i <= 2 * N; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i,
               a[i]);
    }

    return 0;
}
```

**Is there anything wrong with these examples?**

# Array Examples

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 0; i < N; i++)
        a[i] = 0;

    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    int sum = 0;


    for (i = 0; i < N; i++)
        sum += a[i];

    return 0;
}
```

```
#define N 20;
int main (void) {
    int a[N], i;
    for (i = 1; i <= 2 * N; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i,
               a[i]);
    }

    return 0;
}
```

**goes beyond the array bounds!**



**Is there anything wrong with these examples?**



# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```


Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```



**remaining  
elements  
initialized to 0**

Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```

← **all elements initialized to 0**

← **remaining elements initialized to 0**

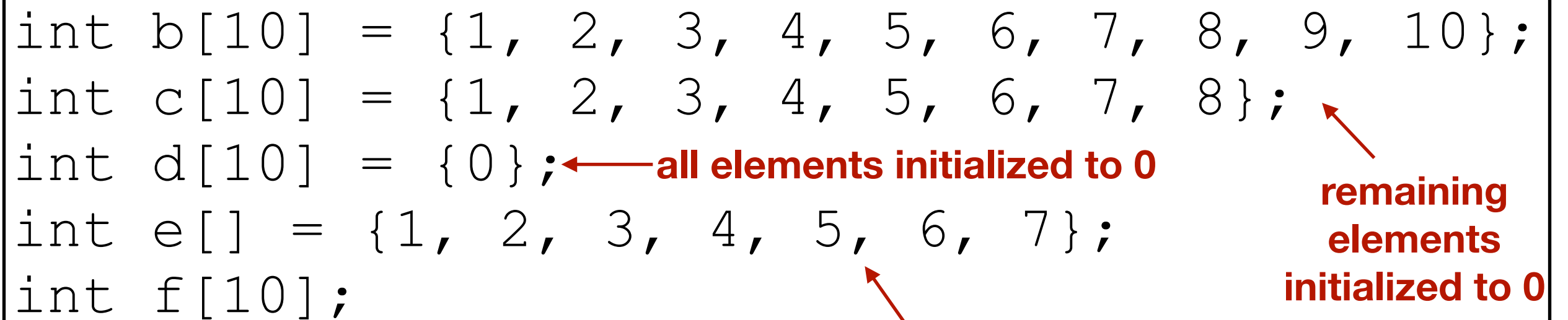
Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```



The diagram illustrates array initialization rules with red arrows pointing to specific parts of the code:

- An arrow points from the text **all elements initialized to 0** to the `{0}` in the declaration of `d[10]`.
- An arrow points from the text **remaining elements initialized to 0** to the closing brace of the initializer list for `c[10]`.
- An arrow points from the text **compiler uses the length of the initializer to determine array length** to the number `5` in the initializer list for `e[]`.

Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```

The diagram shows five lines of C code. Red arrows point from explanatory text to specific parts of the code: one from 'uninitialized array...' to 'int f[10];', one from 'all elements initialized to 0' to '{0}', one from 'remaining elements initialized to 0' to the trailing part of '{1, 2, 3, 4, 5, 6, 7}', and one from 'compiler uses the length of the initializer...' to the list of numbers in '{1, 2, 3, 4, 5, 6, 7}'.

**uninitialized array. Cannot make any assumption about the element values**

**compiler uses the length of the initializer to determine array length**

Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};  
int d[10] = {0};  
int e[] = {1, 2, 3, 4, 5, 6, 7};  
int f[10];
```

The diagram shows five lines of C code. Red arrows point from explanatory text to specific parts of the code: one from 'uninitialized array...' to 'f[10]'; one from 'compiler uses the length of the initializer...' to the number '7' in 'e[] = {1, 2, 3, 4, 5, 6, 7}'; one from 'all elements initialized to 0' to the '0' in 'd[10] = {0}'; and one from 'remaining elements initialized to 0' to the closing brace '}' in 'c[10] = {1, 2, 3, 4, 5, 6, 7, 8}'.

**uninitialized array. Cannot make any assumption about the element values**

**compiler uses the length of the initializer to determine array length**

Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

The diagram shows a line of C code. A red arrow points from the word 'designators' to the '[2]' in the first designated initializer '[2] = 2'.

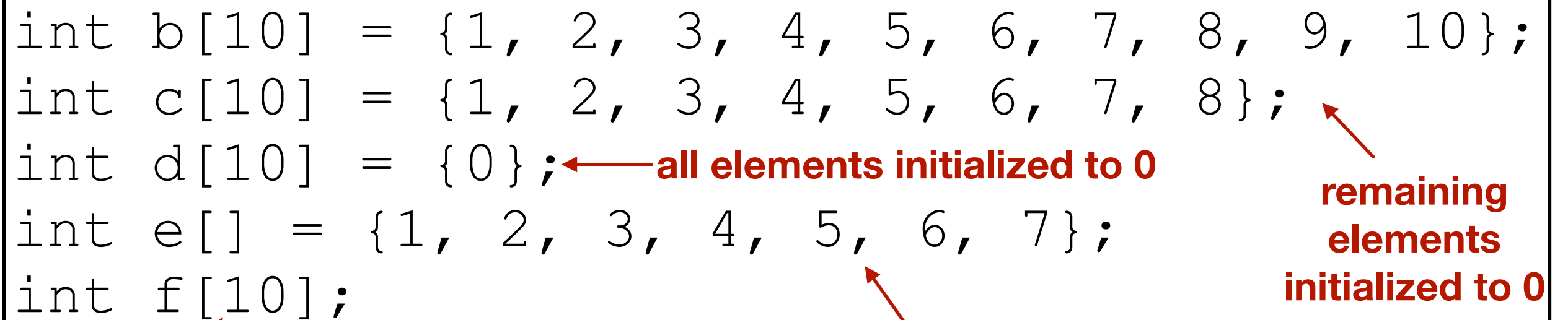
**designators**

**(Specifies the array index)**

# Array Initialization

Can initialize array when it is declared:

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8};
int d[10] = {0};
int e[] = {1, 2, 3, 4, 5, 6, 7};
int f[10];
```



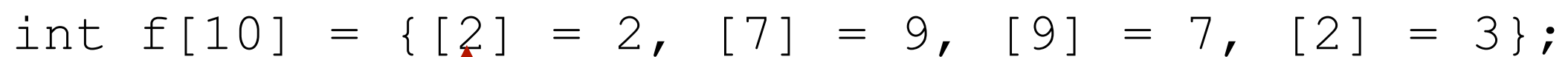
The diagram shows five lines of C code. Red arrows point from text annotations to specific parts of the code: one from 'uninitialized array...' to 'f[10]'; one from 'all elements initialized to 0' to the '0' in '{0}'; one from 'remaining elements initialized to 0' to the closing brace of '{1, 2, 3, 4, 5, 6, 7}'; and one from 'compiler uses the length of the initializer...' to the '7' in '{1, 2, 3, 4, 5, 6, 7}'.

**uninitialized array. Cannot make any assumption about the element values**

**compiler uses the length of the initializer to determine array length**

Can use designated initializers (only c99):

```
int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```



The diagram shows a line of C code using designated initializers. A red arrow points from the text 'designators' to the '[2]' in the first initializer '[2] = 2'.

**designators**  
**(Specifies the array index)**

**Same rules as above apply for unspecified indices**

# Example of Using Arrays

## (variation of p166)

Write a program that counts the number of repeated digits in a given number as follows:

```
Enter a number: 3456787
7 is repeated 2 times
```

```
Enter a number: 97585788
5 is repeated 2 times
7 is repeated 2 times
8 is repeated 3 times
```

```
Enter a number: 9758
No repeated digit
```

demo: find\_repeated.c



# Multi-dimensional Arrays

- An example would be a 2-d array (a.k.a a matrix in mathematical terminology): `int a[5][9];`
  - ▶ 5 rows, indexed from 0
  - ▶ 9 columns, indexed from 0
  - ▶ each element in the array has the same type. In this case, `int`.

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

# Multi-dimensional Arrays

- An example would be a 2-d array (a.k.a a matrix in mathematical terminology): `int a[5][9];`
  - ▶ 5 rows, indexed from 0
  - ▶ 9 columns, indexed from 0
  - ▶ each element in the array has the same type. In this case, `int`.

`a[2][1];`

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

# Multi-dimensional Arrays

- An example would be a 2-d array (a.k.a a matrix in mathematical terminology): `int a[5][9];`
  - ▶ 5 rows, indexed from 0
  - ▶ 9 columns, indexed from 0
  - ▶ each element in the array has the same type. In this case, `int`.

`a[2][1];`

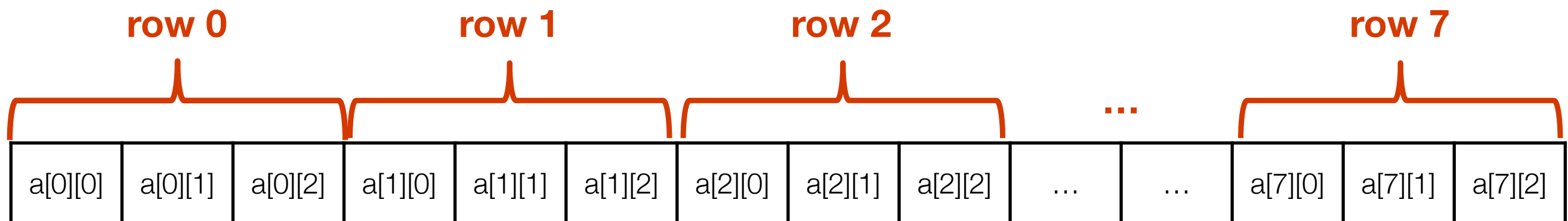
`a[3][7];`

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

# Memory Representation of Multi-dimensional Arrays

- Although we visualize 2-d arrays as a table/matrix, this is not how they are represented in memory. Multi-dimensional arrays in C are stored in row-major order.

```
int a[8][3];
```



# Initializing a 2-D Array

- Use several 1-D initializers, one per row, or designated initializers in C99. Unspecified values will be initialized to 0.

```
int a[5][9] = {{1, 5, 1, 7, 9, 0, 1, 1, 1},  
               {0, 3, 1},  
               {1, 1, 0, 1, 1, 1},  
               [3][2] = 10,  
               [4][8] = 20};
```

- Constant arrays can be declared with the keyword `const`. This means that the program cannot change the value of any element. Useful when you want a “dictionary” to look up things.

```
const char card_suits[] = {'D', 'H', 'C', 'S'};
```

# Accessing Each Element in a 2D Array

- Nested loops are ideal for this task
- For 2-d arrays, you want one outer loop that steps through every row index and an inner loop that steps through every column index

# Randomly Dealing a Hand of Cards

```
Enter a number of cards in hand: 5  
Your hand: 7C 2S 5D aS jH
```

Things to think of:

- how to pick cards randomly?
- how to avoid repeated cards?

demo: deal\_hand.c

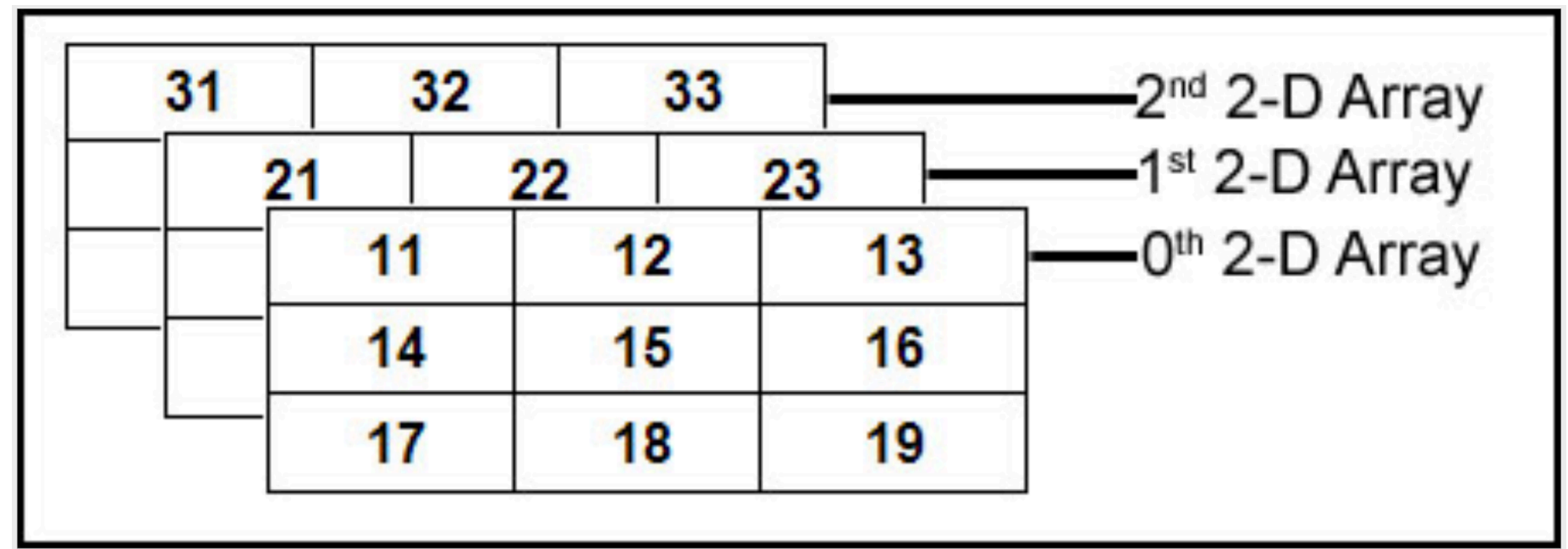
# Higher-dimensional Arrays

- A 2D array is an array of arrays. A 3D array is an array of arrays of arrays, and so on.
- The same concepts of initialization and memory layout apply to higher-dimensional arrays.



# Example of 3-D Array

```
int arr[3][3][3]=
{
    {
        {11, 12, 13},
        {14, 15, 16},
        {17, 18, 19}
    },
    {
        {21, 22, 23},
        {24, 25, 26},
        {27, 28, 29}
    },
    {
        {31, 32, 33},
        {34, 35, 36},
        {37, 38, 39}
    },
};
```



<----- 0 <sup>th</sup> 2D Array ----->									<----- 1 <sup>st</sup> 2D Array ----->									<----- 2 <sup>nd</sup> 2D Array ----->								
11	12	13	14	15	16	17	18	19	21	22	23	24	25	26	27	28	29	31	32	33	34	35	36	37	38	39
1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1024	1026	1028	1030	1032	1034	1036	1038	1040	1042	1044	1046	1048	1050	1052	1054

[Source: <https://owlcation.com/stem/How-to-work-with-Multidimensional-Array-in-C-Programming>]

# Variable-length Array ( only with `-std=c99` )

- Allows the use of an expression rather than a constant to specify the length of the array

```
int main(void) {  
    int i, n;  
  
    printf("Enter the length of array: ");  
    scanf("%d", &n);  
  
    int a[n]; /* declare a length-n array a */  
  
    for (i = 0; i < n; i++) {  
        printf("a[%d] has an initial value %d\n", i, a[i]);  
    }  
  
    return 0;  
}
```

# Variable-length Array ( only with `-std=c99` )

- Allows the use of an expression rather than a constant to specify the length of the array

```
int main(void) {
    int i, n;

    printf("Enter the length of array: ");
    scanf("%d", &n);

    int a[n]; /* declare a length-n array a */

    for (i = 0; i < n; i++) {
        printf("a[%d] has an initial value %d\n", i, a[i]);
    }

    return 0;
}
```

**Note that we are using an uninitialized value. We cannot guarantee what this value will be!**

# Multi-dimensional Variable

## Length Arrays

```
#include <stdio.h>
```

```
int main() {
```

```
    int rows, columns;
```

```
    printf("Enter num of rows: ");  
    scanf("%d", &rows);
```

```
    printf("Enter num of columns: ");  
    scanf("%d", &columns);
```

```
    int ages[rows][columns];
```

```
    for (int i = 0; i < rows; i++)  
        for(int j = 0; j < columns; j++)  
            ages[i][j] = i + j;
```

```
    for (int i = 0; i < rows; i++) {  
        for(int j = 0; j < columns; j++)  
            printf("%d\t", ages[i][j]);
```

```
        printf("\n");
```

```
    }
```

```
    return 0;
```

```
}
```

# String Functions

(Covered in much more detail later)

# Strings in C

- Strings in C are basically character arrays, with a null character ' \0 ' after the last letter of the string.

- Example:

```
char name[10];  
printf("What is your name?");  
scanf("%s", name); //same idea for fscanf
```

# Strings in C

- Strings in C are basically character arrays, with a null character ' \0 ' after the last letter of the string.

- Example:

```
char name[10];  
printf("What is your name?");  
scanf("%s", name); //same idea for fscanf
```

**Use %s to read a string**

# Strings in C

- Strings in C are basically character arrays, with a null character `'\0'` after the last letter of the string.

- Example:

```
char name[10];  
printf("What is your name?");  
scanf("%s", name); //same idea for fscanf
```

**Use %s to read a string**

**note that there is no & before the array name**



# Strings in C

- Strings in C are basically character arrays, with a null character '`\0`' after the last letter of the string.

- Example:

```
char name[10];  
printf("What is your name?");  
scanf("%s", name); //same idea for fscanf
```

**Use %s to read a string**

**note that there is no & before the array name**

<b>name</b>	a	l	i	c	e	\0				
-------------	---	---	---	---	---	----	--	--	--	--

# Some Notes about Strings

- Reading a string with `scanf` will add the null character for you at the end of the string it reads
- However, remember that `scanf` ignores whitespace and the newline character is a whitespace so `scanf` will just ignore the newline entered after the string. It will also stop at the first space. This means that `scanf` can read only one word at a time
- If you want to read a whole line that might have multiple words, use the `fgets` function:  
`fgets(name, 9, stdin);`
  - `name` is the character array you are reading the string into
  - 9 is the maximum number of characters to read (will stop when 8 characters are read or it reads a newline)
  - `stdin` is the stream you are reading from (can be a `FILE*` variable)
  - In general, reading input lines with `fgets` is preferred as we will see later.
- If you want to compare two strings (i.e., null-terminated character arrays), use the `strcmp` function from the `string.h` library:

```
if (strcmp(argv[1], "-m") == 0) { ... }
```