MPATE-GE 2618:
C Programming for Music Technology

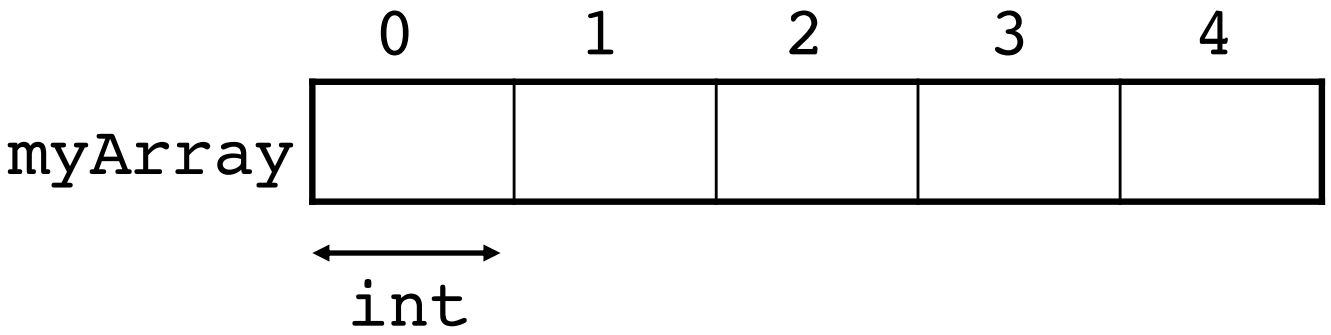
Arrays and Strings

Reading

- Chapters
  - 6 – Working with Arrays
  - 9 – Character Strings, but skip pages 218-226

What is an array?

- An array is a series of elements of the same type that occupies consecutive memory locations
- The array *name* represents the start of the array in memory
- The array *index* is used to access individual elements of the array. *All arrays are indexed from zero!*
- An array containing 5 values of type `int` and called myArray would look like this:



Declaring an array

- Like a regular variable, an array has to be declared before used.
- It requires a type and a length indication
- A typical declaration:

  type `name [numElements];`

  Note: the number between the [] must be a constant (typically an literal integer or a #define value).

- Examples:

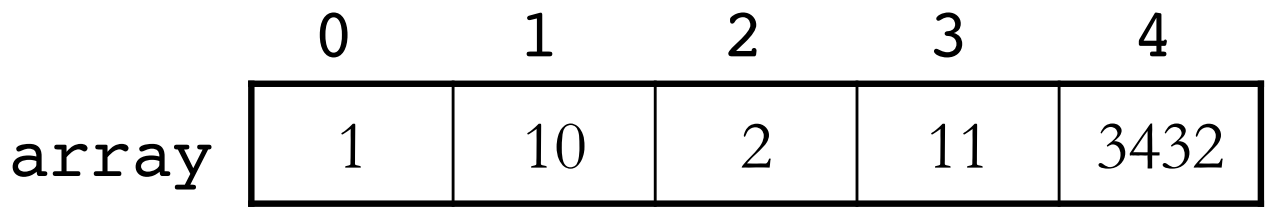
```
#define numElements 10
int arr[10];
float array[numElements];
```

Initializing an array

- When declaring an array, the elements can be explicitly initialized using `{}` notation. For example:

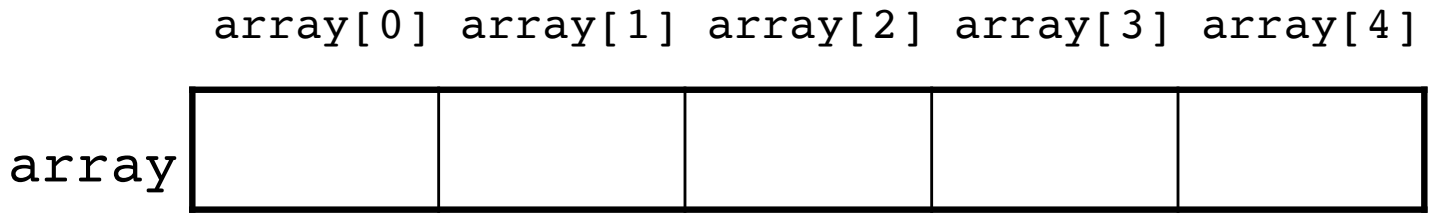
```
int array[5] = { 1, 10, 2, 11, 3432 };
```

- The above declaration would create an array that looks like this:



Accessing the values of an array

- We can access any of the values in the array as if it were a normal variable, meaning we can both read and modify its value.
- Our previously declared array had 5 elements and each element can be referred to like this:

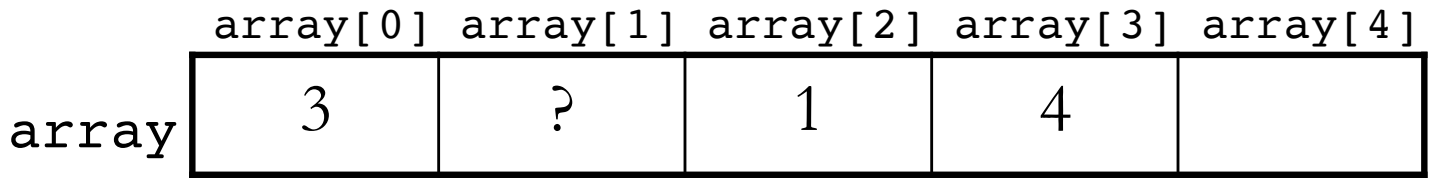


- To store the value 55 in the third element of array, we do this:
```
array[2] = 55;
```
- To set the variable x to the third element of array we do this:
```
x = array[2];
```

Accessing values in an array

- Other valid operations:

```
int a = 3, b;
array[2] = 1;
array[0] = a;
array[a] = 4;
b = array[a-1];
array[array[b]] = array[2] + b;
```



Summary: declaring and defining an array

- How to declare an array:

```
type name[length];
```

- Assigning values:

```
int grades[5]; // declaration of array of 5 ints
grades[0] = 22; // explicitly set elements of array
grades[1] = 0;
grades[2] = 0;
grades[3] = 0;
grades[4] = 98;
```

- Another form of initialization (on declaration only):

```
int grades[5] = { 22, 0, 0, 0, 98 };
```

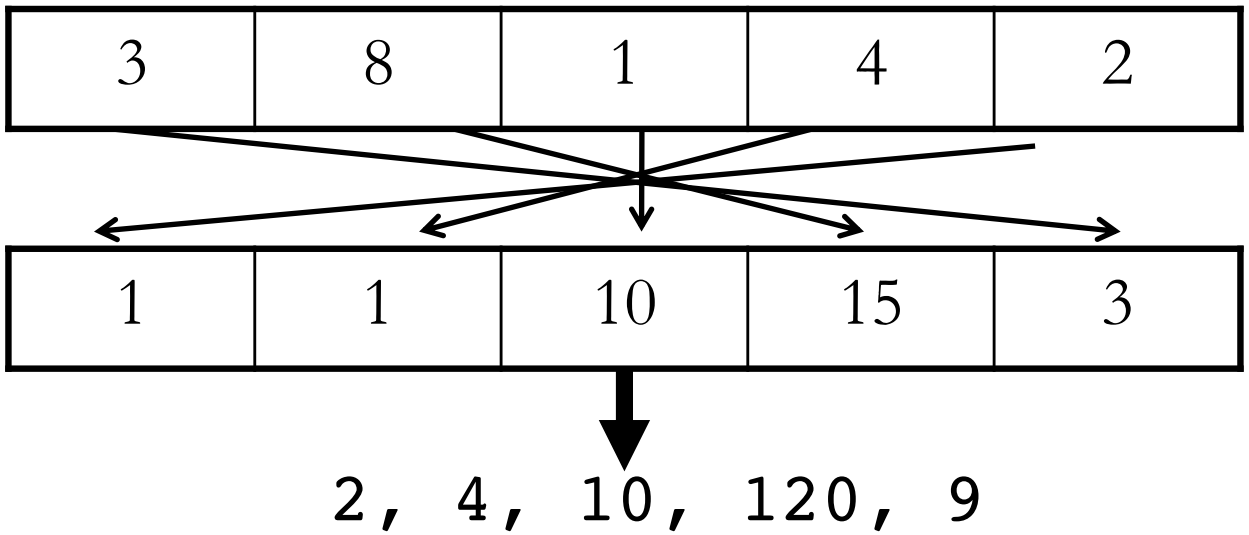
Array example

- Write a program that prompts the user for four grades which are then input into a single array.
- Compute the average of the four grades. Round the results to 1/10th point.

See array1.c for solution.

Array example #2

- Write a program that prompts the user for two sets of N integers and puts the values in two arrays.
- Print the result of the values of the two array multiplied in this manner:
  - the first value of array 1 is multiplied by the last value of array 2
  - the second value of array 1 is multiplied by the second-to-last value of array 2
  - etc.



- See Unit1/array2.c

Two-dimensional arrays

- Easy two-dimensional array
- Conventional array emulates 2-dim array

```
#define NumRows 5
#define NumCols 6
int array[NumRows*NumCols]
int v, i = 2, j = 3;
v = array[i*NumCols + j];
```

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15		
18					
24					29

- This works because C arrays are indexed from zero!
- Matlab (indexed from 1) requires

```
v = array((row-1)*NumCols + col-1)
```

Two-dimensional arrays
Explicit indexing

- We can use explicit indexing to access the same array

```
#define NumRows 5
#define NumCols 6
int v, i = 2, j = 3;
int array[NumRows][NumCols];
v = array[i][j];
```

- This also applies to multi-dimensional arrays

```
int array[N1][N2][N3];
```

Two-Dimensional Arrays Initialization

- All arrays must occupy consecutive memory locations!
- Can initialize as conventional array
  - Exploits consecutive storage of rows

```
int array[NumRows][NumCols] = {
    1,  2,  3,  4,  5,  6,
    7,  8,  9, 10, 11, 12,
   13, 14, 15, 16, 17, 18,
   19, 20, 21, 22, 23, 24,
   25, 26, 27, 28, 29, 30};
```

Alternate Initialization

- Can initialize using structure that conforms to array dimension
  - Structure indicated by use of braces

```
int array[NumRows][NumCols] = {
   { 1,  2,  3,  4,  5,  6},
   { 7,  8,  9, 10, 11, 12},
   {13, 14, 15, 16, 17, 18},
   {19, 20, 21, 22, 23, 24},
   {25, 26, 27, 28, 29, 30}
};
```

Arrays of character: Strings

- Arrays of characters are so useful they are given a special name: *Strings*
- Possible string declarations:

```
char *mystring = "hello";
char mystring[] = "hello";
char mystring[] = {'h', 'e', 'l', 'l', 'o', '\0' };
char mystring[6] = {'h','e','l','l','o','\0'};
```

- Each declaration results is same characters in array
- Last character in string is **"null" ('\0')** character.
- Each string is accessed in the same way
  - **mystring[1]** is **'e'**

Pointers and Strings

- The statement **`char *mystring`** declares **`mystring`** as a *pointer to char*
  - More on pointers later
- This declaration makes it particularly easy to initialize a string

```
char *mystring = "hello";
```

- You still use conventional array indexing to access this string
  - **`mystring[1]`** is **`'e'`**

Null termination of strings

- All strings *must* be null terminated.
  - This is an easy way to know the end of string
  - Strings in C do not explicitly indicate length!
- Null termination is done by default on initializations like this one:

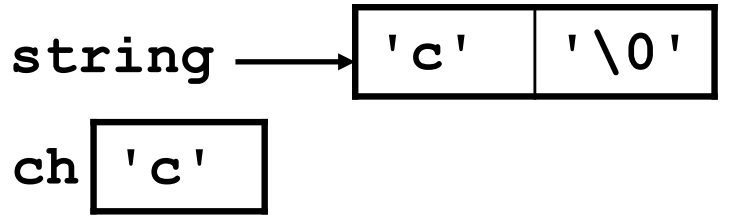
```
char *string = "hello";
```

- The null character is `'\0'` or ASCII value zero

Characters and Strings

- Difference between “c” and ‘c’:

```
char string[] = "c";
char ch =  'c';
```



- Double quote **“c”** is string
- Single quote **‘c’** is a single character

Arrays of Strings

- Consider

```
char *name1 = "John";
char *name2 = "Jill";
char *names[2] = {name1, name2};
```

- `names` is an array of strings
- Example

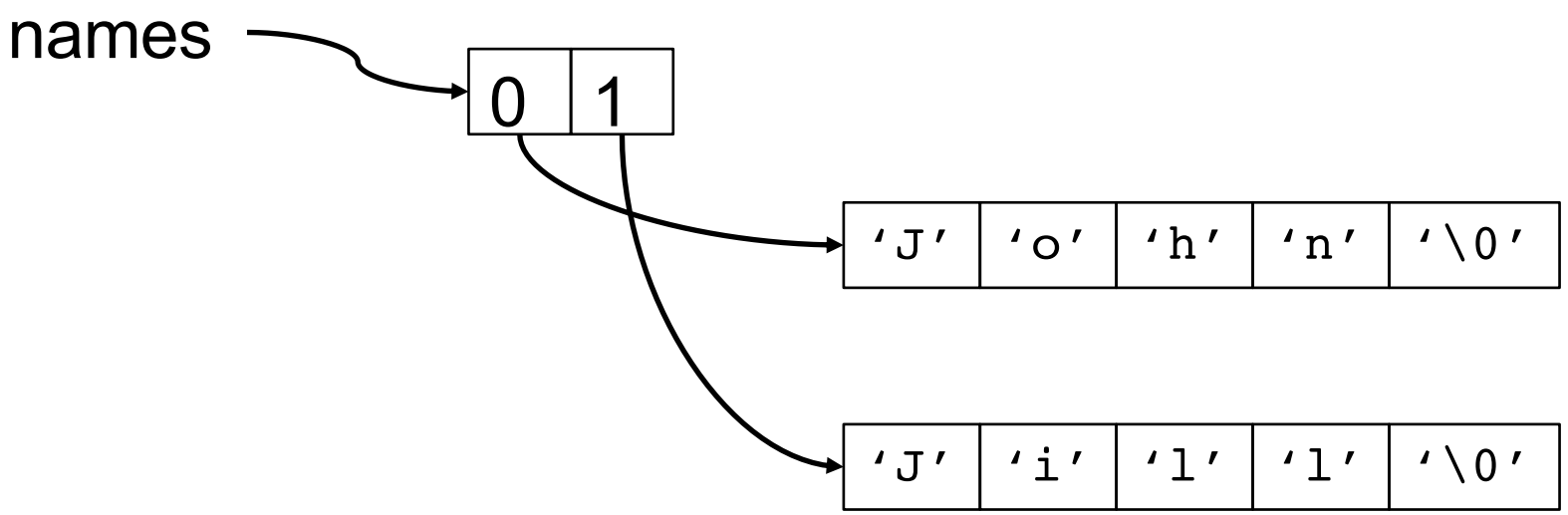
```
printf("%s\n", names[0]);
printf("%s\n", names1);
```

- Both print

```
John
```

Array of Strings in Memory

- The array of strings would have a layout in memory like this



Array of Strings as 2-D Array

- Given

```
char *name1 = "John";
char *name2 = "Jill";
char *names[2] = {name1, name2};
```

- Then

`name1[0]` is `'J'`

`name2[1]` is `'i'`

`names[0][0]` is `'J'`

`names[1][1]` is `'i'`

2D String Array

- Given

```
char *name1 = "John Smith";
char *name2 = "Jill Johnson";
char *names[2] = {name1, name2};
```

- Is names[][] a square 2-D array?
  - No!
  - Length of **`name1`** is 10, **`name2`** is 11
  - And that is OK, since each name string (i.e. row) carries it own length as NULL termination.

More on strings

- Comparison of strings: *this doesn't work:*

```
char *string = "hello";
if (string == "hello") //WRONG!
```

- Assignment of strings: *this doesn't work:*
  - Assignment after declaration

```
char string[80];  //declaration is OK
string = "hello"; //assignment is WRONG!
```

- Printing strings: use `%s` in printf() statement

```
printf("%s", mystring);
```

- See `Unit1/string1.c`

String library: string.h

- To copy and compare strings, use the functions in the `string.h` library

  ```
  #include <string.h>
  ```

- Important functions include
  `strlen, strcmp, strcpy, strcat`

  (for finding string length, comparing strings and copying strings)

- See cppreference.com/strcmp()
  - Note that it indicates: "defined in header <string.h>"
- See file `Unit1/string[23].c` for examples on how to use these functions

String length

- `n = strlen(mystr)` returns length of string *not including terminating NULL character*

```
int i, j, len1, len2;
char *str1 = “Hello”, *str2 = “World”, str3[80];
len1 = strlen(str1);
len2 = strlen(str2);
for (i=0; i<len1; i++)
    str3[i] = str1[i]; //don’t copy NULL
//continue with value of i
for (j=0; j<len2; i++,j++)
    str3[i] = str2[j]; //don’t copy NULL
str3[i] = 0; //add NULL termination (works because i++)
```

Copy and concatenate strings

- `strcpy(dest, src)` copies string `src` to string `dest`
- `strcat(dest, src)` concatenates string `src` to *end* of string `dest` (overwrites NULL)
- Together, they perform same function as `for` loops in previous slide

```
char *str1 = "Hello", *str2 = "World", str3[80];
strcpy(str3, str1);
strcat(str3, str2);
```

- See Unit1/string2.c

Safer functions for copy and concatenate strings

- **`strncpy(dest, src, n)`**
  - copies no more than **`n`** characters of string **`src`** to string **`dest`**
- **`strncat(dest, src, n)`**
  - concatenates no more than **`n`** characters string **`src`** to *end* of string **`dest`**
- They perform same function as **`strcpy()` and `strcat()`** but can control for string overflow
- See Unit1/string3.c

Split a line into words

- In C "white space" separates compiler tokens
  - variables and operators or words and spaces
- White space is: space, tab (\t) and newline (\n)
- Function "string to tokens"

```
tok = strtok(char *line, char *wht_space);
```

- Example
  - First call: `tok = strtok(line, " \t\n");`
  - Subsequent calls: `tok = strtok(NULL, " \t\n");`

String library: ctype.h

- Another important library is **`ctype.h`**. Use functions in this library to deal with single characters.
- Descriptions of these standard C libraries and functions are in Appendix B of your textbook.
- You can also type "**`man 3 ctype`**" in your terminal window to get more info.

Example: ctype.h

```
#include <ctype.h>
char c; //this is user input
switch ( tolower (c) ) {
    case ‘a’:
    <statements>
    break;
}
```

Other functions from <ctype.h>

- Test for a letter

  int isalpha(int c);

- Test for a numeral

  int isdigit(int c);

Dealing with characters and their ASCII values

- As noted, the `ctype.h` library is useful for character-specific operations
  - querying if a string is alpha-numeric,
  - if letters are uppercase or lowercase, etc.
- Let's look at some code that deals with character values: ascii1.c, ascii2.c, and capitalize.c

- http://www.cplusplus.com/reference/cctype/

const Modifier

- `const` can be used to modify any variable type

```
const int max_value = 100;
```

- However, this is not so different from using #define

```
#define MaxValue 100
```

- `const` for arrays
  - compiler checks that values are never altered
- Example: Max values in mid-tread quantizer

```
const int qbits[] = {2, 3, 4, 5};
const int max_values[] = {1, 3, 7, 15};
```

Array bounds overflow

- Going beyond the bounds of an array is *very bad*:

```
int arr[5];
for (i = 0; i < 10; i++) {
    arr[i] = 0;
}
```

- If you run this code, you might get a *memory access fault* or *segmentation fault*.
- See Unit1/array_bounds.c

Review

- Arrays are sequence of values stored in contiguous memory locations
  - Indexed from 0
- Arrays can be multi-dimensional as
  - Implicit: int a[5*6]; Explicit: int a[5][6];
- Strings are arrays of characters with NULL at end

  `char *a = "hello"; //handy dec and init`

  Variable `a` is a pointer to string `"hello"`
- Array of strings:

  `char *names[2]={"n1";"n2"};`

  `name[0]` is a pointer to a string `"n1"`