Lecture Notes for Lecture 5 of CS 5001 (Foundations of CS) for the Fall, 2017 session at the Northeastern University Silicon Valley Campus.

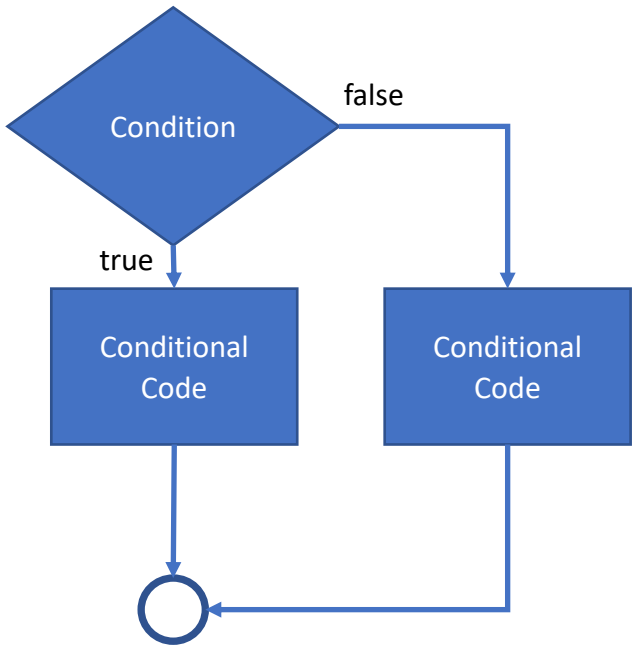
Arrays and Strings

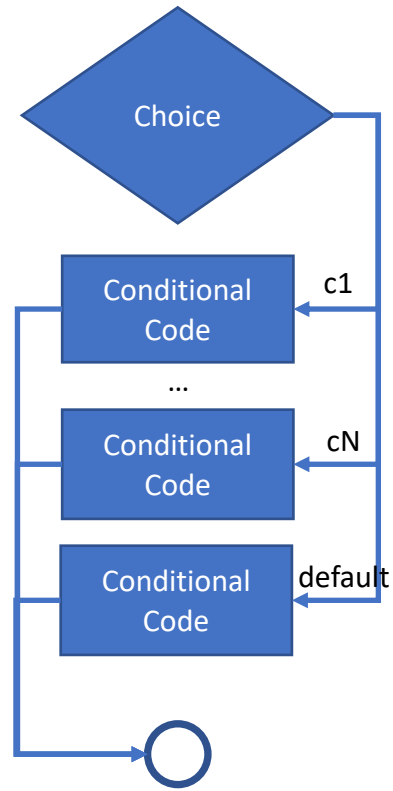
Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 4 Review

- Program logic often depends on evaluating conditions during a computation
- C provides control statements that use results to determine what sequence of statements to perform
- Control statements include:
  - Conditional: “either-or”
    - **if** (*condition*) { *statements* } **else** { *statements*}
  - Choice: “one-of”
    - **switch** (*intval*) { **case** *c1*: … **break**; … **case** *CN*: … **break**; **default**: … **break**; }
  - Repeated: “many”
    - **while** (*condition*) { *statements* }
    - **do** { *statements* } **while** (*condition*)
    - **for** (*init* ; *condition*; *re-init*) { *statements* }
- Control statements can be nested in other control statements

Lecture 4 Review





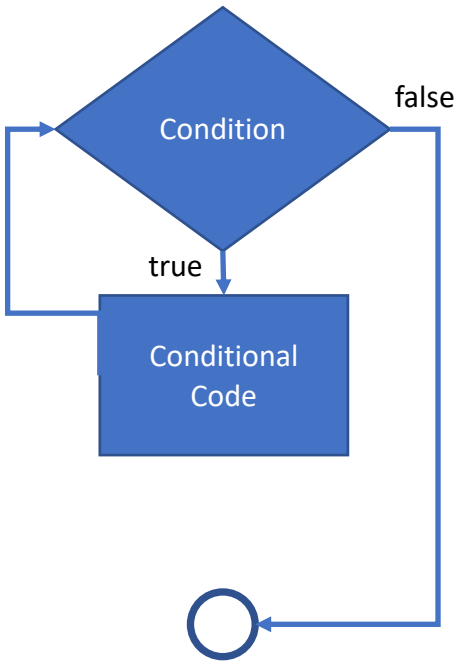
Conditional: "either-or"

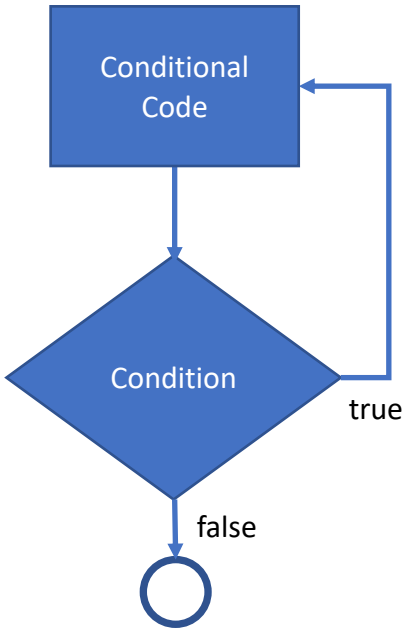
if (*condition*) {
  statements
} **else** {
  statements
}

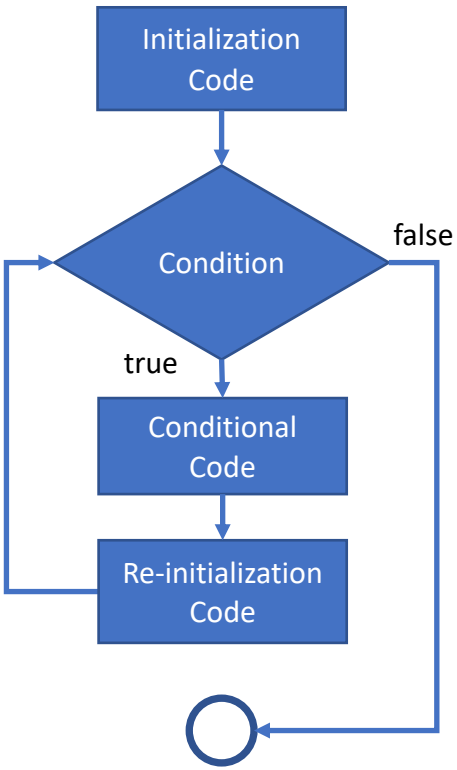
Choice: "one-of"

switch (*intval*) {
  case *c1*: *statements;* **break**; ...
  case *CN*: *statements;* **break**;
  default: *statements;* **break**;
}

Lecture 4 Review







Repeated: "many"

while (*condition*) {
statements
}

Repeated: "many"

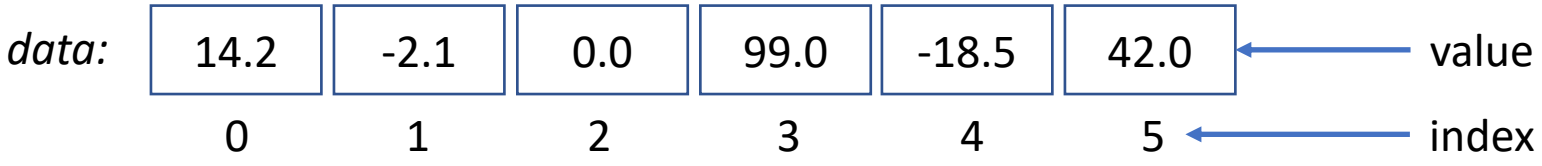
do {
statements
} **while** (*condition*)

Repeated: "many"

for (*int; condition; re-init*) {
statements
}

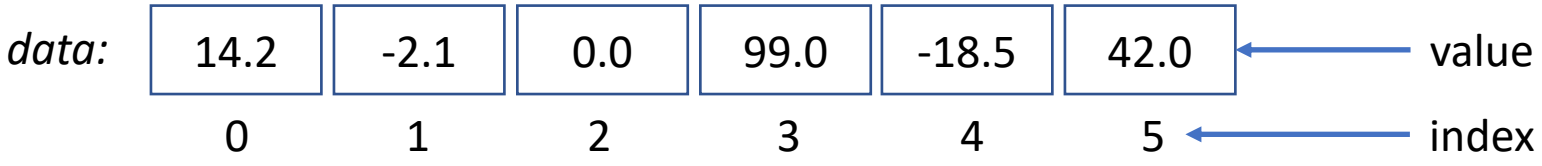
Array

- An *array* is a fixed-size, sequential collection of *elements* of the same type.
- Elements of an array occupy contiguous memory locations
- A non-negative integer is used as an *array index*.
- Here is an array named *data* of six double values:



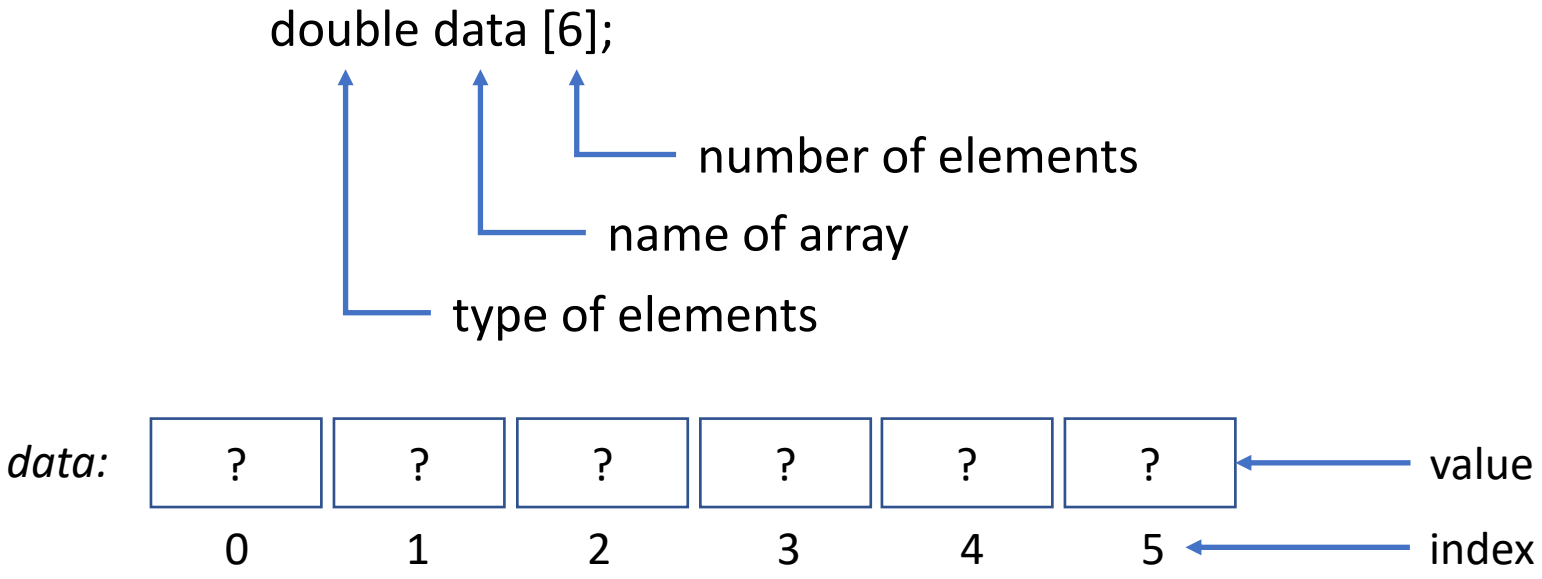
Array

- Size of the array is 6; occupies 6*8 = 48 bytes of memory
- First element is data[0]; last element is data[5]
- Get 2nd element of *data*: double d = data[1];
- Divide 4th element of *data* by 3.0: data[3] /= 3.0; *// 33.0*



Array

- Declare uninitialized array *data* of 6 doubles

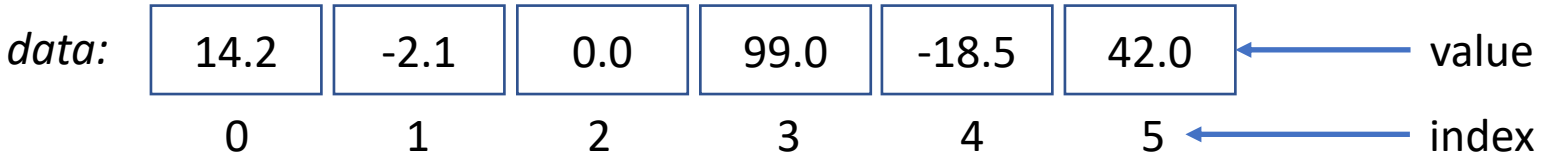


Array

- Declare initialized array *data* of 6 doubles

```
double data [6] = { 14.2, -2.1, 0.0, 99.0, -18.5, 42.0 };
```

Initializer list



Array

- Declare initialized array *data* using size of initializer list

```
double data [ ] = { 14.2, -2.1, 0.0, 99.0, -18.5, 42.0 };
```

6

Initializer list

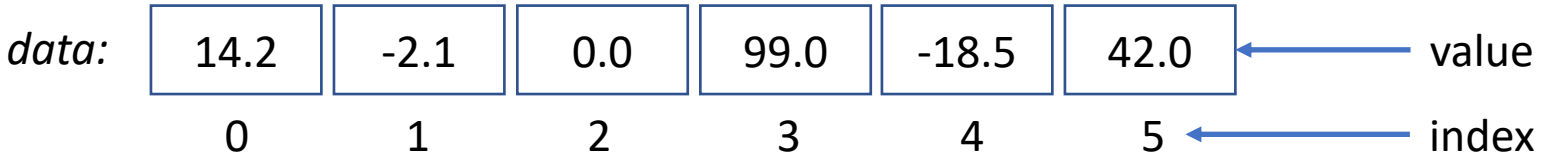
data:	14.2	-2.1	0.0	99.0	-18.5	42.0	value
	0	1	2	3	4	5	index

Array

- Print the array elements:

```
double data [6] = { 14.2, -2.1, 0.0, 99.0, -18.5, 42.0 };

for (int i = 0; i < 6; i++) {
    printf("data[%d] = %.1f\n", i, data[i]);
}
```



Output:

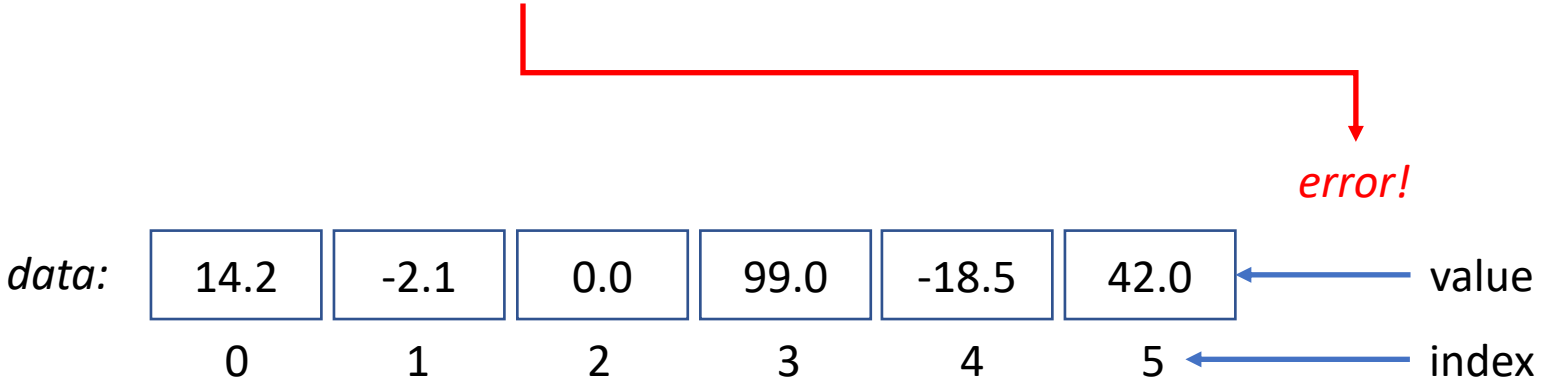
```
data[0] = 14.2
data[1] = -2.1
data[2] = 0.0
data[3] = 99.0
data[4] = -18.5
data[5] = 42.0
```

Array

- C does not check array index out of bounds, so be careful!

```
double data [6] = { 14.2, -2.1, 0.0, 99.0, -18.5, 42.0 };

data[6] = -88;   // overwrites memory location after array!
```



Array

- A 2D array is a fixed-size, sequential collection of *elements* of the same type that represent rows and columns of data
- Elements of a 2D array occupy contiguous memory locations
- A non-negative integer is used as row and column indexes.
- Here is unsigned char array *image* of 3 rows and 3 columns :

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

Array

- Size of the array is 3x3; occupies 3x3x1 = 9 bytes of memory
- First element is image[0][0]; last element is image[2][2]
- Get row 2 column 1 value: unsigned char color = image[1][0];
- Divide row 1 column 2 value by 2: image[0][1] /= 2;

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

Array

- Declare uninitialized array *image* of 3x3 unsigned char

```
unsigned char image[3][3];
```

number of columns

number of rows

name of array

type of elements

image:

	0	1	2
0	?	?	?
1	?	?	?
2	?	?	?

column index

color value

row index

Array

- Declare initialized array *image* of 3x3 unsigned char

```
unsigned char image[3][3] =
  { {197, 12, 0}, {99, 255, 86}, {42, 197, 62} };
```

Initializer lists per row

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

Array

- Declare initialized 3x3 array using size of initializer list

```
unsigned char image[  ][3] =
   { {197, 12, 0}, {99, 255, 86}, {42, 197, 62} };
```

Initializer lists per row

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

Array

- Print 2-D array elements

```
unsigned char image[ ][3] =
   { {197, 12, 0}, {99, 255, 86}, {42, 197, 62} };
for (int r = 0; r < 3; r++) {
   for (int c = 0; c < 3; c++) {
      printf("image[%d][%d] = %d\n", r, c, image[r][c]);
   }
}
```

Output:

```
Image[0][0] = 197
Image[0][1] = 12
Image[0][2] = 0
Image[1][0] = 99
Image[1][1] = 255
Image[1][2] = 86
Image[2][0] = 42
Image[2][1] = 197
Image[2][2] = 62
```

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

Array

- Actual storage in memory is in “row-major” order”

197	12	0	99	255	86	42	197	62
0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2

image:

	0	1	2
0	197	12	0
1	99	255	86
2	42	197	62

column index

color value

row index

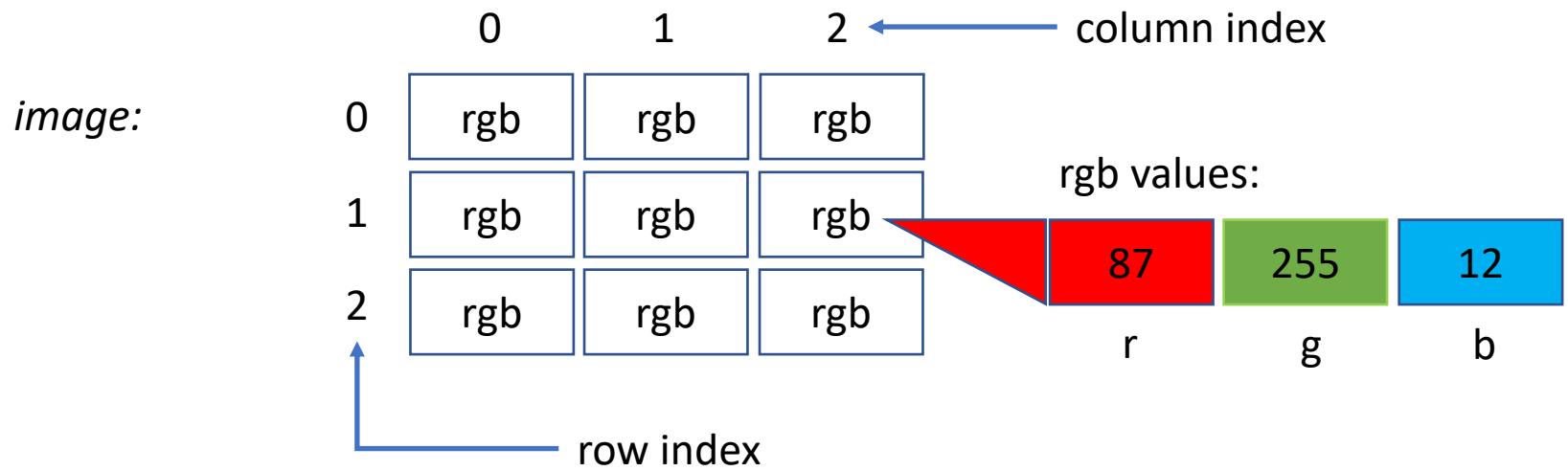
Array

- In reality, C does not have 2-D arrays. A 2-D array is actually an array of 1-D arrays.
- All the 1-D arrays must have the same length, and have elements of the same type.
- The type of each row in the image array is unsigned char[]

	0	1	2 ← array index
image[0]	197	12	0
image[1]	99	255	86
image[2]	42	197	62

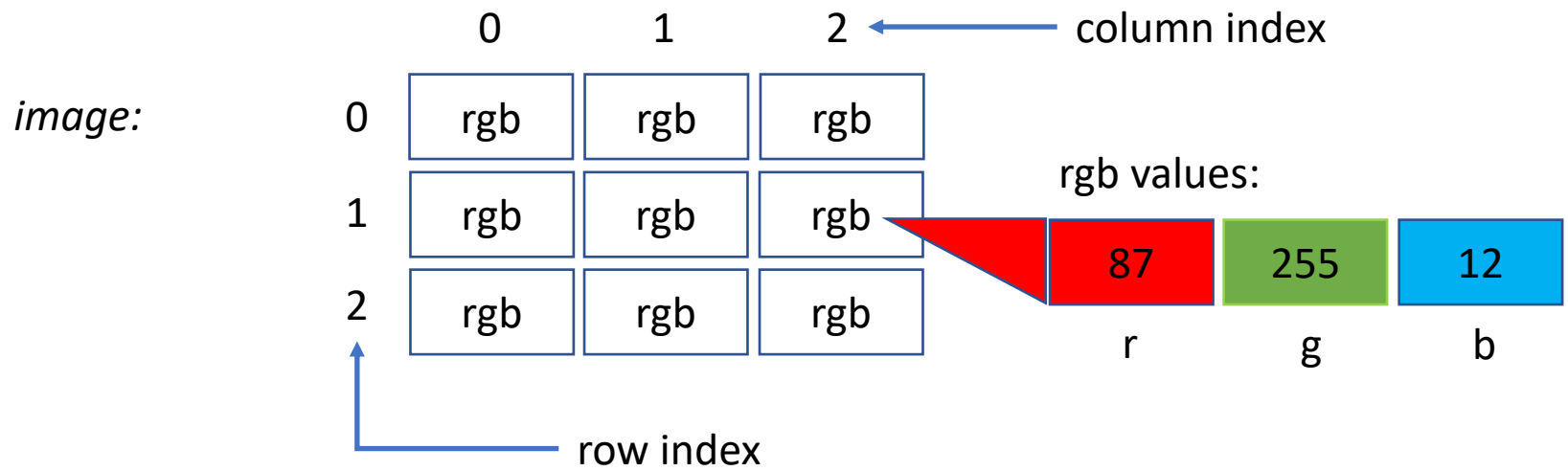
Array

- A 3D array is a fixed-size, sequential collection of *elements* of the same type that represent rows and columns of array data
- Elements of a 3D array occupy contiguous memory locations
- A non-negative integer is used as three indexes.
- Here is an unsigned array *image* of 3 rows and 3 columns x 3 color cells for RGB :



Array

- Size of the array is 3x3x3; occupies 3x3x3x1 = 27 bytes
- First element is image[0][0][0]; last element is image[2][2][2]
- Get green value at row 2 column 3 value:
  unsigned char green_color = image[1][2][1];
- Subtract 10 from row 2 column 3 blue value
  image[1][2][2] -= 10; // value becomes 2



Array

- Printing 3-D array elements

```
unsigned char image[3][3][3] =
   { {197, 12, 0}, {99, 255, 86}, {42, 197, 62} },
   { {12, 0, 197}, {255, 86, 99}, 197, 62, 42} },
   { {0, 197, 12}, {86, 99, 255}, {62, 42, 197} };

for (int r = 0; r < 3; r++) {
   for (int c = 0; c < 3; c++) {
      for (int v = 0; v < 3; v++) {
         printf("image[%d][%d][%d] = %d\n",
                  r, c, v, image[r][c][v]);
      }
   }
}
```

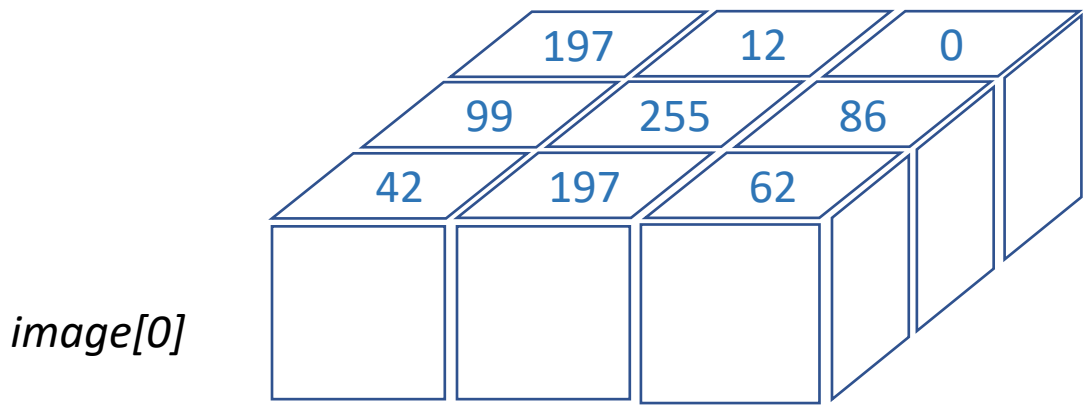
```
image[0][0][0] = 197
image[0][0][1] = 12
image[0][0][2] = 0
image[0][1][0] = 99
image[0][1][1] = 255
image[0][1][2] = 86
image[0][2][0] = 42
image[0][2][1] = 197
image[0][2][2] = 62

image[1][0][0] = 12
image[1][0][1] = 0
image[1][0][2] = 197
image[1][1][0] = 255
image[1][1][1] = 86
image[1][1][2] = 99
image[1][2][0] =197
image[1][2][1] = 62
image[1][2][2] = 42

image[2][0][0] = 0
image[2][0][1] = 197
image[2][0][2] = 12
image[2][1][0] = 86
image[2][1][1] = 99
image[2][1][2] = 255
image[2][2][0] =62
image[2][2][1] = 42
image[2][2][2] = 197
```

Array

- In reality, C does not have 3-D arrays. A 3-D array is actually an array of 2-D arrays.
- All the 2-D arrays must have the same number of rows and columns,, and have elements of the same type.
- The type of each row in the image array is unsigned char[][]



Array

- Printing array from function

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

/**
 * Print array of size n.
 * @param n size of array
 * @param array the array to print
 */
void printArray(int n, double array[]) {
   for (int i = 0; i < n; i++) {
      printf("array[%d] = %.1f\n", i, array[i]);
   }
}

/** Print array and return EXIT_SUCCESS */
int main(void) {
   double data [6] = { 14.2, -2.1, 0.0, 99.0, -18.5, 42.0 };
   printArray(6, data);
   return EXIT_SUCCESS;
}
```

Array

- Printing 2D array from function

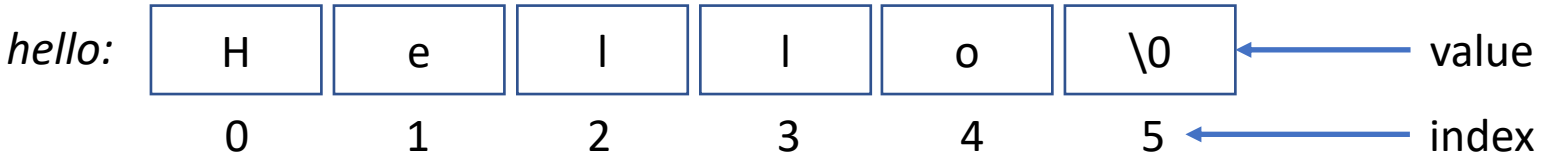
```
#include <stdio.h>
#include <stdlib.h>

/**
 * Print 2D array of size rows x columns
 * @param rows rows in array
 * @param cols columns in the array
 * @array the array to print
 */
void printArray2D(int rows, int cols, float array[][cols]) {
   for (int r = 0; r < rows; r++) {
      for (int c = 0; c < cols; c++) {
         printf("array[%d][%d] = %.1f\n", r, c, array[r][c]);
      }
   }
}

/** Print array and return EXIT_SUCCESS */
int main(void) {
   double data [2][3] = { { 14.2, -2.1, 0.0 }, { 99.0, -18.5, 42.0 } };
   printArray2D(2, 3, data);
   return EXIT_SUCCESS;
}
```

String

- A string is an array of characters terminated by the **null** character '\0'.
- Null character enables string operations to measure the length of the string.
- Length if string is length of array **null** character
- Here is an initialized char array *hello* acting as a string:

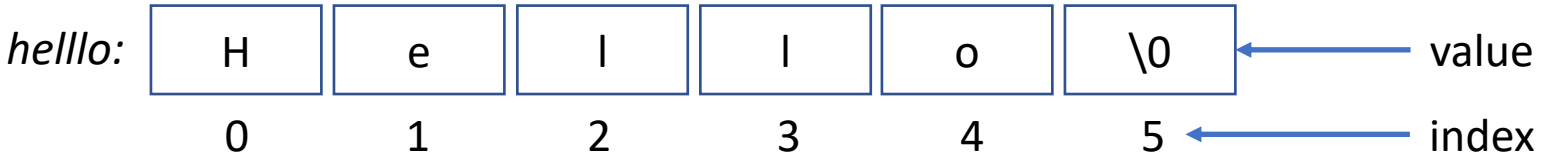


String

- String can be initialized one one of several ways:

```
char  hello[6]  =  {'H',  'e',  'l',  'l',  'o',  '\0'};
char  hello[ ]  =  {'H',  'e',  'l',  'l',  'o',  '\0'};
char  hello[ ]  =  "Hello";  // compiler supplies '\0'

// print message
printf("Greeting message: %s\n", hello);
```



String

- Partial list of C string functions in string.h:

strcpy(s1, s2); Copies string s2 into string s1.
strcat(s1, s2); Concatenates string s2 onto the end of string s1.
strlen(s1); Returns the length of string s1.
strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

String

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/** Example using string functions */
int main (void) {
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   strcpy(str3, str1);    // copy str1 into str3
   printf("strcpy( str3, str1) : %s\n", str3 );
   strcat( str1, str2);     // concatenate str1 and str2
   printf("strcat( str1, str2): %s\n", str1 );
   int len = strlen(str1);   // length of str1 after concatenation
   printf("strlen(str1) : %d\n", len );
   return EXIT_SUCCESS;
}
```

Output:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```