

# Arrays

October 26, 2021

## 1 Arrays

### 1.1 Topics

- C-array introduction
- static and dynamic arrays
- similarity between arrays and pointers
- passing arrays to functions
- aggregate operations on arrays
- C-string - array of characters
- buffer overflow security flaw
- sorting data
- 2-d array and tic-tac-toe game

### 1.2 Array

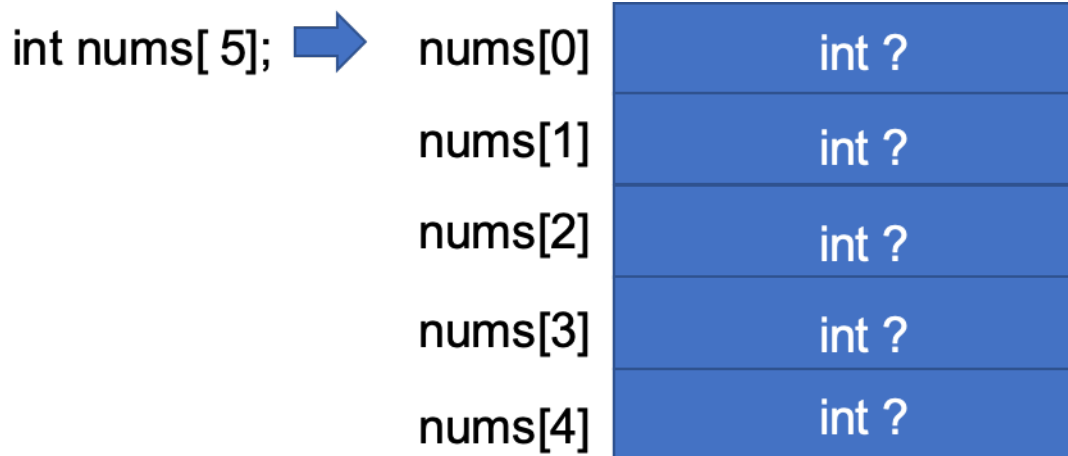
- dictionary definition of array is a range of a particular type of thing
- we've used single variable to store single data/value
- large programs typically deal with a large number of data values that must be stored in memory e.g. sorting data values.
- NOT practical to declare a large number of variables to store a large number of values
- array is a container used to store a large number of same type values under one name
- array we're learning about in this chapter is C-array
- since C++11 standard, C++ provides **array** and **vector** types under STL (standard template library)
  - these advanced types (array and vector) are similar to C++ string type
- understanding the C-array helps understand many C++ concepts and data structures that rely on C-array
  - plus, a large no. of legacy C++ codebase and libraries specially developed before C++11 may be still using C-array
- C++ vector is a better choice among C++ **array** and C-array, if your compiler supports it
  - vector takes care of all the common operations one would do in an array
  - similar to C-string (more below) vs C++ string
- array in this notebook refers to C-array
- there are two types of array:
  1. static array
  2. dynamic array

### 1.3 Static array

- the size of the array is determined during compile-time and is fixed
- local static array is stored on stack memory segment
- syntax to declare a static array

`type arrayName[size];`

- `size` tells the compiler how many of similar type of values can be stored in the `arrayName`
- `size` must be a positive integer (`size_t` type) - literal or variable
- the following figure depicts computer memory when an array of `int` is declared



## Fig. C++ Array Declaration

- each member of the array is called an element
- each element has same type and share same array name but different index
- index also called offset ranges between 0 to size-1

#### 1.3.1 Visualize array using [pythontutor.com](https://pythontutor.com)

```
[1]: #include <iostream>
#include <string>

using namespace std;

[2]: // nums array to store 5 integers
int nums[5];
```

#### 1.3.2 Accessing member elements

- members can be accessed and used ONLY one element per operation
- no aggregate operation is allowed on the array variable as a whole

- e.g. copy one array into another; printing the whole array, etc.
- only aggregate operation allowed is during array initialization

```
[3]: // access and store values into each element
nums[0] = 10;
nums[1] = 20;
nums[2] = 30;
nums[3] = 40;
nums[4] = 50;
```

```
[4]: // access some element
cout << nums[0];
```

10

```
[5]: // each element can be used like a single variable
nums[1] = nums[2] + nums[3];
```

```
[6]: // traverse an array
for(int i=0; i<5; i++) {
    cout << i << " -> " << nums[i] << endl;
}
```

0 -> 10  
1 -> 20  
2 -> 30  
3 -> 40  
4 -> 50

```
[7]: // declaring and initializing an array
// size is optional; will be determined with the no. of values it's initialized
    ↪with
float grades[] = {90.5f, 34.5f, 56, 81, 99, 100, 89.9};
```

```
[8]: grades
```

```
[8]: { 90.5000f, 34.5000f, 56.0000f, 81.0000f, 99.0000f, 100.000f, 89.9000f }
```

## 1.4 Member functions

- C-array is so primitive that it doesn't come with any useful operations or member functions
- implementing any array operation falls under programmer's responsibility!
- e.g. how can you quickly tell the size or length of an array?

```
[9]: // finding the size of the array
size_t arr_size = sizeof(grades)/sizeof(float);
```

```
[10]: cout << "array's size or length = " << arr_size;
```

array's size or length = 7

```
[11]: cout << "last grade = " << grades[arr_size-1] << endl;
```

last grade = 89.9

#### 1.4.1 Array size is fixed!

- one has to know how many elements will be stored in a given array
- what happens when the array is full?

```
[12]: // grades doesn't have index 7 as the size is 7
      grades[7] = 67;
```

```
input_line_22:3:1: warning: array index 7 is past the
end of the array (which contains 7 elements) [-Warray-bounds]
grades[7] = 67;
~      ~
```

```
input_line_14:4:1: note: array 'grades' declared
here
float grades[] = {90.5f, 34.5f, 56, 81, 99, 100, 89.9};
~
```

### 1.5 Array and Pointers

- there's a lot of similarities on how array and pointers work!
  - they can be used interchangeably as desired

```
[13]: int ids[] = {100, 200, 300, 400};
```

```
[14]: // copy the base address of array
      // which is the address of element at index 0; which is &ids[0];
      int * ptr = ids;
```

```
[15]: // print the base memory addresses
      cout << ptr << " equals to " << &ids[0] << " equals to " << ids;
```

0x10de0df80 equals to 0x10de0df80 equals to 0x10de0df80

```
[16]: // print the data located at the base memory addresses
      cout << *ptr << " equals to " << ids[0] << " equals to " << *ids;
```

100 equals to 100 equals to 100

```
[17]: // using pointers to traverse array
      // point to the second element
```

```
ptr++;
```

```
[18]: // dereference the value at that location
cout << *ptr << endl;
```

200

```
[19]: ptr = ids; // copy the base address
for(int i=0; i<4; i++) {
    cout << i << "-> " << *(ptr+i) << " == " << ptr[i] << " == " << ids[i] <<
    endl;
}
```

0-> 100 == 100 == 100

1-> 200 == 200 == 200

2-> 300 == 300 == 300

3-> 400 == 400 == 400

## 1.6 Dynamic array

- array size can be determined during run time (program execution)
  - once the size is set, it's fixed
- local dynamic array is allocated on the heap memory segment using pointer and **new** operator
- syntax to declare dynamic array:

```
type * arrayName = new type[size];
```

- size can be a variable determined or assigned during program execution
- once the dynamic array is declared, using dynamic array is same as using static array
- dynamic memory must be deallocated to prevent memory leak
- syntax:

```
delete[] arrayName;
```

### 1.6.1 Visualize dynamic array in [pythontutor.com](http://pythontutor.com)

```
[20]: size_t capacity;
```

```
[21]: cout << "How many integers would you like to enter? ";
cin >> capacity;
```

How many integers would you like to enter?

2

```
[22]: int * some_array = new int[capacity];
```

```
[23]: // prompt user to store capacity number of integers and store them into array
for(int i=0; i<capacity; i++) {
    cout << "Enter a number: ";
    cin >> some_array[i];
}
```

Enter a number:

10

Enter a number: 20

```
[24]: // output some values
cout << capacity << " " << some_array[0] << " " << some_array[capacity-1];
```

2 10 20

## 1.7 Aggregate operations on arrays

- some commonly used aggregate operators are (=, math operators (+, \*, etc.), comparison operators (>, ==, etc.)
- array doesn't allow any aggregate operations as a whole
  - e.g. copy one array into another; printing the whole array, etc. are aggregate operations
  - it doesn't make sense to compare two arrays (compare with what elements' values?)
  - Input/Output needs to be done one element at a time

### 1.7.1 shallow copy with = operator

- both dynamic and static arrays CAN'T be copied to another array using = operator
- both dynamic and static array can be assigned to another dynamic array
  - however, it doesn't actually copy the data (shallow copy)
- copying one array into another by its name copies only the base address
  - thus creating two aliases pointing to the same memory location
  - if one is modified, the other is modified as well

### 1.7.2 Visualize shallow copy using [pythontutor.com](http://pythontutor.com)

```
[25]: int * copy_array = new int[arr_size];
```

```
[26]: // try to copy some_array into copy_array as a whole
copy_array = some_array;
```

```
[27]: // let's see some values
cout << some_array[0] << " == " << copy_array[0];
```

10 == 10

```
[28]: // let's update some_array
some_array[0] = 100;
```

```
[29]: // now, let's see the value of copy_array[0]  
cout << some_array[0] << " == " << copy_array[0];
```

100 == 100

### 1.7.3 Deep copy

- deep copy refers to the actual copy of the data
- data from one array must be copied to another array element by element
- must write your own function or code to achieve the deep copy
- Couple of notes:
  - destination array type must match the source array type
  - destination array size must be at least as big as the source array size

```
[30]: // let's copy some_array created above  
// let's create an empty array to deep copy data to  
int * deep_copy = new int[capacity];
```

```
[31]: // let's deep copy  
for(int i=0; i<capacity; i++)  
    deep_copy[i] = some_array[i];
```

```
[32]: // if one array is modified it doesn't affect the other array  
deep_copy[0] *= 2; // update the first element with twice its value
```

[32]: 200

```
[33]: // let's print the copied data side by side  
for(int i=0; i<capacity; i++) {  
    cout << i << " -> " << deep_copy[i] << " " << some_array[i] << endl;  
}
```

0 -> 200 100

1 -> 20 20

```
[34]: deep_copy
```

[34]: @0x7ffee62f7520

## 1.8 Passing array to function

- arrays (both static and dynamic) can be passed to a function
- array provides an efficient way to pass a large number of similar values without copying them
  - pass-by reference is by default and the only way!
  - arrays can't be passed by value

```
[35]: // since actual size of the array is not easy to determine,  
// size of the array is typically passed as an argument
```

```
void updateArray(int array[], int size) {
    for(int i = 0; i < size; i++) {
        array[i] *= 2; // simply double the value of each element
    }
}
```

```
[36]: // print array function; notice passing pointer
void printArray(int * array, int size) {
    cout << "{";
    for(int i=0; i<size; i++)
        cout << array[i] << ", ";
    cout << "}\n";
}
```

```
[37]: printArray(some_array, arr_size);

{100, 20, -929235295, 1073743866, 3, 0, -1, }
```

```
[38]: updateArray(some_array, arr_size);
```

```
[39]: printArray(some_array, arr_size);

{200, 40, -1858470590, -2147479564, 6, 0, -2, }
```

## 1.9 Returning array from function

- since aggregate assignment operator = is not allowed on array, returning a local static array is not possible
- returning dynamic array is possible but not the best practice!
  - details as to why it's a bad practice is left for your own research and exploration
  - Hint: it has to do with the ownership and memory management (deleting memory, etc.)
  - quick demo of returning dynamic array can be visualized at [pythontutor.com](http://pythontutor.com)
- best practice is to pass an empty array (pass-by reference) and get it filled inside the function
  - a technique to get the data/result out of the function without explicitly returning it from a function

## 1.10 C-string

- C language doesn't have a type defined to work with string like in C++
- now that we understand pointer and C-array, let's revisit C-string
- C-string is an array of characters that ends with a NULL character '\0' (ASCII 0)

```
[40]: // declaration and initialization is easier
// NULL character is automatically added at the end!
char name[] = "John Smith";
```

```
[41]: // once declared; working with C-string is a little cumbersome
// you've to work one character at a time!
```



```
char f_name[10];
```

```
[42]: f_name[0] = 'J';  
f_name[1] = 'a';  
f_name[2] = 'k';  
f_name[3] = 'e';  
f_name[4] = '\0';
```

```
[43]: // C-strings must end with null-character '\0'  
cout << f_name;
```

Jake

### 1.11 Library Functions to work with C-string

- <http://www.cplusplus.com/reference/cstring/>

### 1.12 Array of strings

- we can declare array of any type (fundamental and advanced)

```
[44]: #include <iostream>  
#include <string>  
  
using namespace std;
```

```
[45]: // array of C++ string  
string names[] = {"John", "Jake", "Dave", "Jenny"};
```

```
[46]: // first element and first character of first element  
cout << names[0] << " first char = " << names[0][0];
```

John first char = J

### 1.13 Array of char \*

- array of C-string (char \*)
- similar to array of C++ string conceptually; harder to work with however!
- a parameter for **main( int argc, char\* argv[] )** is always an array of char\*

```
[47]: // create array of char * that stores 4 C-strings  
char * stuff[4];
```

```
[48]: char val1[] = "ball";
```

```
[49]: char val2[] = "test";
```

```
[50]: stuff[0] = val1;
      stuff[1] = val2;
      stuff[2] = "cat";
      stuff[3] = "dog";
```

input\_line\_63:4:12: **warning:** ISO C++11 does not allow

conversion from string literal to 'char \*' [-Wwritable-strings]

```
stuff[2] = "cat";
      ^
```

input\_line\_63:5:12: **warning:** ISO C++11 does not

allow conversion from string literal to 'char \*' [-Wwritable-strings]

```
stuff[3] = "dog";
      ^
```

### 1.13.1 Passing array of char \* to function

```
[51]: // write a function similar to main
      int my_main(int argc, char* argv[]) {
          cout << "argc = " << argc << endl;
          for(int i=0; i< argc; i++) {
              cout << argv[i] << " " << endl;
              if (string(argv[i]) == "test")
                  cout << " test is found in argv[]\n";
          }
          return 0;
      }
```

```
[52]: my_main(4, stuff);
```

```
argc = 4
ball
test
  test is found in argv[]
cat
dog
```

### 1.14 Buffer Overflow

- C-string is also called buffer
- if C-string is not used correctly, it'll lead to buffer overflow security flaw
- if data is copied to C-string buffer without checking the bounds, it may overflow!
- one of the most dangerous security flaws that lets hackers completely control the vulnerable program and computer
- in-depth study of buffer overflow and exploitation is beyond the scope of the course

### 1.14.1 Demo programs for buffer overflow

- buffer overflow can be used to overwrite existing data or corrupt memory
  - a simple overflow demo is found at [demos/arrays/buffer\\_overflow1/](#)
- buffer overflow can be used to change the flow of execution; read other part of memory
  - a more intuitive demo is found here: [demos/arrays/buffer\\_overflow2/](#)
- buffer overflow can be exploited to execute arbitrary code
  - for details see: <https://github.com/rambasnet/EthicalHacking>

### 1.15 Sorting data

- sorting is a very important operation done to solve a large number of problems
- all the data must be stored in memory in order to sort
- e.g., sort students' records based on grades, ids, names, etc.
- there are many algorithms to sort data
  - one of the highly studied topics in algorithm courses
- you should learn these algorithms and implement on their own to sort data
- an easy and efficient way to sort data is using library
- `<algorithm>` header library has many commonly used algorithms implemented
  - more: <https://en.cppreference.com/w/cpp/header/algorithm>
- `sort(begin, end)` function sorts the data given a sequence that has `begin( )` and `end( )`
  - by default it sorts data in ascending order
  - can be customized to sort data in descending order

```
[53]: // let's declare an array of float
float stu_grades[] = {100, 99.6, 55, 100, 65, 15.5};
```

```
[54]: #include <algorithm> // sort()
#include <iterator> // begin() and end()
```

```
[55]: // sort stu_grades in ascending order
sort(begin(stu_grades), end(stu_grades));
```

```
[56]: // now let's see the sorted values
stu_grades
```

```
[56]: { 15.5000f, 55.0000f, 65.0000f, 99.6000f, 100.000f, 100.000f }
```

```
[57]: // let's sort stu_grades in descending order
// pass greater<type> function template that is used to compare the data
// with greater value towards the beginning
sort(begin(stu_grades), end(stu_grades), greater<float>());
```

```
[58]: stu_grades
```

```
[58]: { 100.000f, 100.000f, 99.6000f, 65.0000f, 55.0000f, 15.5000f }
```

```
[59]: // sort array of strings
string words[] = {"zebra", "yoyo", "x-ray", "ball", "apple"};
```

```
[60]: // sort in ascending order
sort(begin(words), end(words));
```

```
[61]: words
```

```
[61]: { "apple", "ball", "x-ray", "yoyo", "zebra" }
```

## 1.16 Bubble sort

- bubble sort repeatedly compares and swaps two adjacent elements if they're not in order
- see animation here: [https://en.wikipedia.org/wiki/Bubble\\_sort](https://en.wikipedia.org/wiki/Bubble_sort)
- step through the algorithm here: <https://opensa-server.cs.vt.edu/ODSA/Books/CS3/html/BubbleSort.htm>
- one of the worst performing algorithms; but used to demonstrate a quick and easy way to write your own sort algorithm for a small number of elements
  - because of its poor performance, bubble sort should not be used in real-world applications

```
[62]: #include <iostream>
#include <string>

using namespace std;
```

```
[63]: template<class T>
void printArray(T * arr, int size) {
    cout << "{";
    for(int i=0; i<size; i++)
        cout << arr[i] << ", ";
    cout << "}\n";
}
```

```
[64]: template<class T>
void bubbleSort(T * array, int size) {
    bool swapped;
    for(int pass=0; pass<size; pass++) {
        swapped = false;
        // let's print array before every pass
        // TODO: comment out the the following debugging info...
        //cout << "pass # " << pass << ": ";
        //printArray<T>(array, size);
        for(int i=0; i<size-1-pass; i++) {
            // sort in ascending order; check out of order?
            if (array[i] > array[i+1]) {
                swap(array[i], array[i+1]);
                swapped = true;
            }
        }
    }
}
```

```

    }
    // check if the elements are sorted; i.e. not single pair was swapped
    // let's print array after each pass; uncomment the following statement
    //printArray<T>(array, size);
    if (!swapped)
        break;
}
}

```

```
[65]: int numbers[] = {100, 99, 55, 100, 65, 15};
```

```
[66]: bubbleSort<int>(numbers, 6);
```

```
[67]: numbers
```

```
[67]: { 15, 55, 65, 99, 100, 100 }
```

```
[68]: float values[] = {7.9, 3.5, 5.5, 6.5, 7.5, 7.6};
```

```
[69]: bubbleSort<float>(values, 6);
```

```
[70]: values
```

```
[70]: { 3.50000f, 5.50000f, 6.50000f, 7.50000f, 7.60000f, 7.90000f }
```

## 1.17 Two-dimensional array

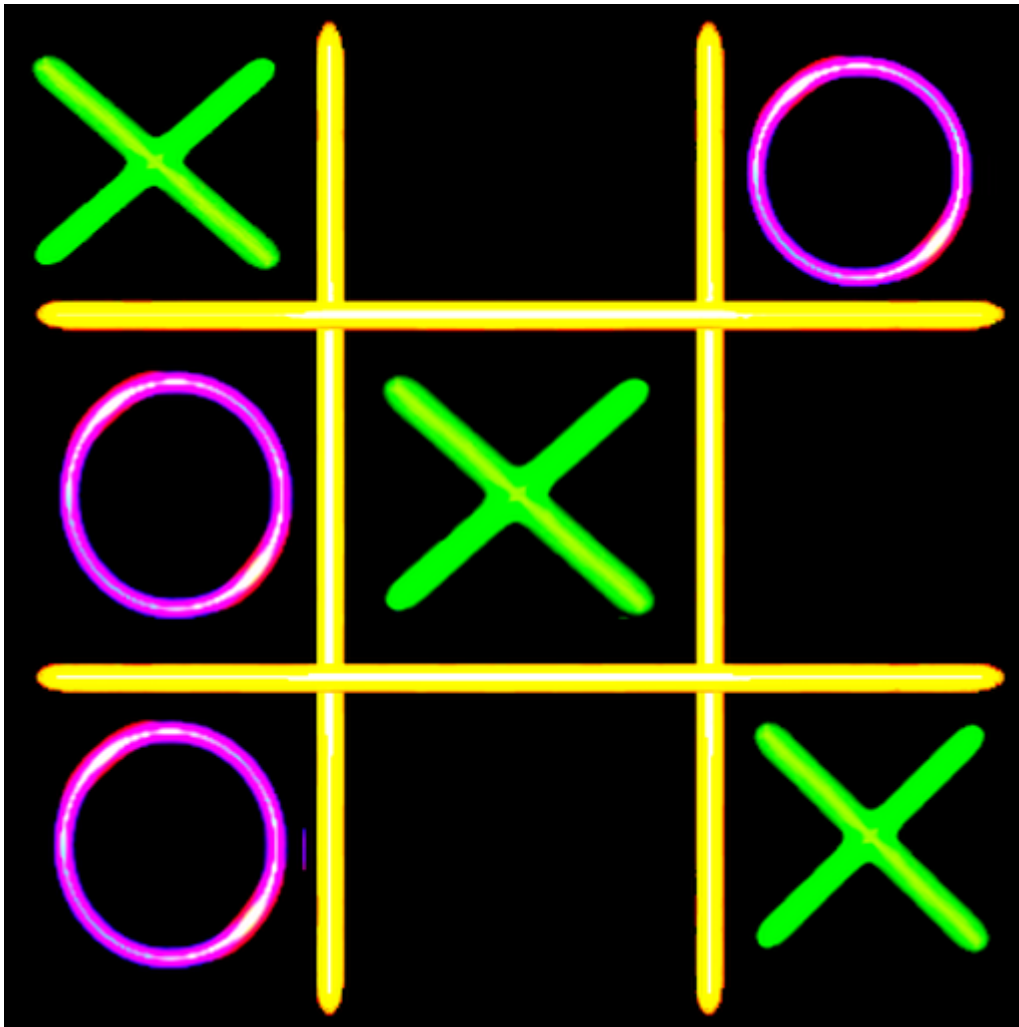
- two dimensional array is a useful construct to store data of 2-D in nature
  - table with row and column (representing 2-d board games), cartesian coordinates, etc.
- storing 3-D and beyond is also possible
  - 3-D array is used in video games to store graphics information
- syntax to declare 2-D array:

```
type arrayName[rowSize][colSize];
```

- 2-D array can be both static and dynamic

### 1.17.1 Tic-tac-toe game application

- represent 2-D tic-tac-toe board



```
[71]: #include <iostream>
#include <iomanip>
#include <string>

using namespace std;
```

```
[72]: // declare a 2-d tic-tac board;
// tic_tac_toe[0][0] represents top left box
char tic_tac_toe[3][3];
```

```
[73]: // define a function to initialize empty tic_tac_toe board
// Note: must provide the column_width inside []
void initTicTacToe(char board[][3], int row) {
    for(int i=0; i<row; i++)
        for(int j=0; j<3; j++)
            board[i][j] = ' '; // space represents empty box
}
```

```
[74]: void printTicTacToe(char board[][3], int row) {
    cout << endl << setfill('-') << setw(14) << " " << endl;
    for(int i=0; i<row; i++) {
        cout << "| ";
        for(int j=0; j<3; j++)
            cout << tic_tac_toe[i][j] << " | ";
        cout << endl << setfill('-') << setw(14) << " " << endl;
    }
}
```

```
[75]: // let's initialize our board
initTicTacToe(tic_tac_toe, 3);
```

```
[76]: // let's print the empty board
printTicTacToe(tic_tac_toe, 3);
```

```
-----
|   |   |   |
-----
|   |   |   |
-----
|   |   |   |
-----
```

```
[77]: // let's fill Xs and Os as shown in the above figure
// assuming a game play
tic_tac_toe[0][0] = 'X';
tic_tac_toe[0][2] = 'O';
tic_tac_toe[1][0] = 'O';
tic_tac_toe[1][1] = 'X';
tic_tac_toe[2][0] = 'O';
tic_tac_toe[2][2] = 'X';
```

```
[78]: printTicTacToe(tic_tac_toe, 3);
```

```
-----
| X |   | O |
-----
| O | X |   |
-----
| O |   | X |
-----
```

```
[79]: // let's determine winner!
char findWinner(char board[][3], int row) {
```

```

char winner; // is it O or X?
bool won;
// check 3 rows
for(int i=0; i<row; i++) {
    winner = board[i][0]; // whatever symbol is at the first box, that
    ↪ should appear in other columns
    won = true;
    // check the rest of the columns
    for(int j=1; j<3; j++) {
        if (winner != board[i][j]) {
            won = false;
            break;
        }
    }
    if (won) // we've a winner
        return winner;
}
// #FIXME: check columns FIXME#
// check diagonals
// top left to bottom right
if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) return
    ↪ board[0][0];
// #FIXME: check the other diagonal

return '-'; // return '-' if it's a tie
}

```

```
[80]: char winner;
```

```
[81]: winner = findWinner(tic_tac_toe, 3);
```

```
[82]: if (winner == '-')
    cout << "Oops! it's a tie...\n";
else
    cout << "Congrats " << winner << "! You win!!\n";
```

Congrats X! You win!!

## 1.18 Labs

1. The following lab demonstrates the usage of an array data structure and some operations on arrays.
  - use partial solution `array.cpp` in `labs/arrays` folder
  - use Makefile to compile and debug program
  - fixe all the FIXMEs and write `#FIXED#` next to each FIXME once fixed



## 1.19 Exercises

1. Write a function that takes an array and finds and returns the max value in the array.
  - write at least 3 automated test cases

```
[83]: #include <cassert>
```

```
[84]: template<class T>
T max(T * array, int size) {
    assert(size >= 1); // make sure array is not empty!
    T curr_max = array[0];
    for(int i=1; i<size; i++) {
        // if the value at i is larger than curr_max; update it with the new max
        if (curr_max < array[i])
            curr_max = array[i];
    }
    return curr_max;
}
```

```
[85]: void test_max() {
    assert(max({1, 2, 3} == 3));
    assert(max({10, -5, -30} == 10));
    assert(max({-10, -5, -30, 0, -100} == 0));
    cerr << "all test cases passed for max()\n";
}
```

```
[86]: test_max();
```

all test cases passed for max()

2. Write a function that takes an array and finds and returns the min value in the array.
  - write at least 3 automated test cases
3. Write a complete C++ program that computes some statistical values on any given number of numbers
  - prompt user to enter a bunch of numbers
  - find and display the max and min values
  - find and display the average or mean
  - find and print the range (max - min) in the array
  - find and display the mode or modal (the number with largest frequency)
  - program continues to run until the user wants to quit
4. Write a search function that checks if a given value is found in an array.
  - write 3 automated test cases

## 1.20 Kattis problems

- a large number of difficult problems require to store data in 1 or 2-d arrays and manipulate the data
- solve the following Kattis problems writing at least 3 automated test cases for each function used as part of the solution

1. Falling Apart - <https://open.kattis.com/problems/fallingapart>
2. Statistics - <https://open.kattis.com/problems/statistics>
3. Line Them Up - <https://open.kattis.com/problems/lineup>

### 1.21 Summary

- learned about array and types of arrays
- passing array to functions
- similarity between array and pointers in terms of using memory addresses
- methods or member functions or lack there of
- array of C++ strings and C-strings
- went over a quick intro to buffer overflow security vulnerability
- sorting using <algorithm> and writing our own bubble sort
- 2-d array and it's application on tic-tac-toe game

[ ]: