

Arrays

January 18, 2026

1 Arrays, C-Strings, and Stack Overflow

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 External Resources

- YouTube Video - Arrays, C-Strings, and Stack Overflow - <https://youtu.be/mEHEXNaqXEk>
- YouTube Podcast - Arrays, C-Strings, and Stack Overflow - <https://youtu.be/qwwSKXVwUVg>
- NotebookLM learning materials - <https://notebooklm.google.com/notebook/769aba1f-c864-46f0-ab65-a2499454998d>

1.3 Array

- dictionary defintion 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 practice 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

1.4 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

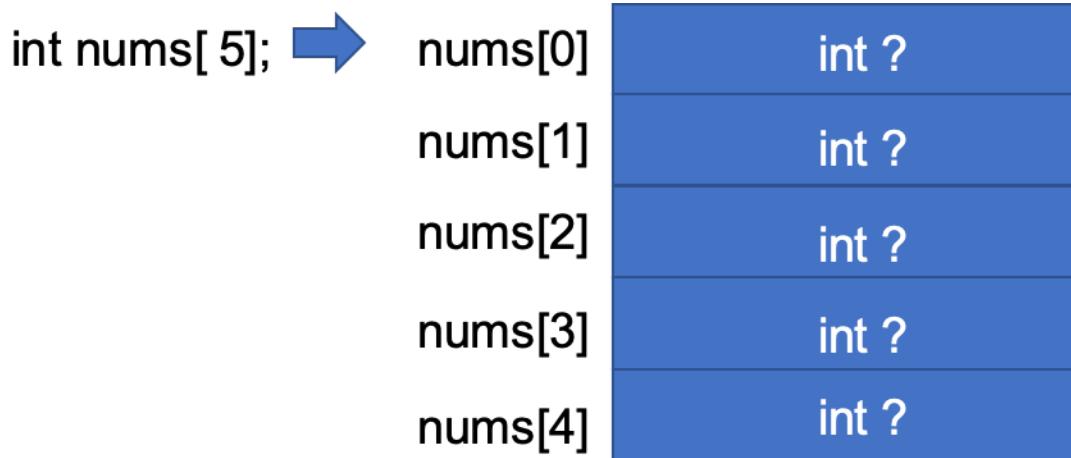


Fig. C++ Array Declaration

- 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

1.4.1 Visualize array using pythontutor.com

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

using namespace std;
```

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

1.4.2 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

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

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

10

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

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

0 -> 10
1 -> 70
2 -> 30
3 -> 40
4 -> 50

```
[ ]: // 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.f, 99.f, 100, 89.9};
```

```
[ ]: cout << grades;
```

0x7b07fbba0050

1.5 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?

```
[ ]: grades.size();  
  
input_line_16:2:8: error: member reference base type 'float [7]' is not a  
structure or union  
grades.size();  
~~~~~^~~~~~
```

Interpreter Error:

```
[ ]: // finding the size of the array  
size_t arr_size = sizeof(grades)/sizeof(float);  
  
[ ]: cout << "array's size or length = " << arr_size;  
  
array's size or length = 7  
  
[ ]: cout << "last grade = " << grades[arr_size-1] << endl;  
  
last grade = 89.9
```

1.5.1 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?

```
[ ]: // grades doesn't have index 7 as the size is 7  
grades[7] = 67;  
  
input_line_20:3:1: 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.f, 99.f, 100, 89.9};  
^
```

1.6 Array and Pointers

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

```
[ ]: int ids[] = {100, 200, 300, 400};  
  
[ ]: cout << ids;  
  
0x7b07fbba0070  
  
[ ]: // copy the base address of array  
// which is the address of element at index 0; which is &ids[0];  
int * ptr = ids;
```

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

0x7b07fbba0070 equals to 0x7b07fbba0070 equals to 0x7b07fbba0070

```
[ ]: // 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

```
[ ]: // using pointers to traverse array
// point to the second element
ptr++;
```

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

200

```
[ ]: 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

1.7 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;
```

1.7.1 Visualize dynamic array in pythontutor.com

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

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

How many integers would you like to enter?

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

```
[ ]: // 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: Enter a number: Enter a number: Enter a number: Enter a number:

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

5 10 8

1.8 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

1.8.1 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 alias pointing to the same memory location
 - if one is modified, the other is modified as well

1.8.2 Visualize shallow copy using pythontutor.com

```
[ ]: int * copy_array = new int[capacity];
```

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

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

10 == 10

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

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

```
100 == 100
```

1.8.3 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

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

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

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

```
[ ]: 200
```

```
[ ]: // 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 -> 10 10  
2 -> 5 5  
3 -> 7 7  
4 -> 8 8
```

```
[ ]: deep_copy
```

```
[ ]: @0x7fff923fe588
```

1.9 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

```
[ ]: // 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
    }
}
```

```
[ ]: // 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";
}
```

```
[ ]: printArray(some_array, capacity);
```

```
{100, 10, 5, 7, 8, }
```

```
[ ]: updateArray(some_array, capacity);
```

```
[ ]: printArray(some_array, capacity);
```

```
{200, 20, 10, 14, 16, }
```

1.10 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
- 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

1.11 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)
- aggregate operations `cin` and `cout` are allowed on C-string
 - `cin` leads to buffer overflow condition corrupting memory
 - * the exploitation of the vulnerability compromises the confidentiality, integrity and availability of the program

- cout may print more than the buffer contents if not delimited with '\0'
 - * the exploitation of the vulnerability compromises the confidentiality of the program

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

```
[ ]: char name[] = {'J', 'o', 'h', 'n'};
```

```
[ ]: cout << "Hello " << name << "!" ;
```

Hello John!

```
[ ]: // once declared; working with C-string is a little cumbersome
// you've to work one character at a time or use functions provided in <cstring> library
char f_name[10];
```

```
[ ]: f_name[0] = 'M';
f_name[1] = 'i';
f_name[2] = 'c';
f_name[3] = 'h';
f_name[4] = 'a';
f_name[5] = 'e';
f_name[6] = 'l';
f_name[7] = '\0';
```

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

Hello Michael!

```
[ ]: f_name[0] = 'J';
f_name[1] = 'o';
f_name[2] = 'h';
f_name[3] = 'n';
// forgot to terminate f_name with NULL character \0
```

```
[ ]: 'n'
```

```
[ ]: cout << f_name;
```

Johnael

1.12 Library Functions to work with C-string

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

1.13 Array of strings

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

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

using namespace std;
```

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

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

```
John first char = J
```

1.14 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*

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

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

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

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

```
input_line_55:4:12: warning: ISO C++11 does not allow conversion from string
literal to 'char *' [-Wwritable-strings]
```

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

```
input_line_55:5:12: warning: ISO C++11 does not allow conversion from string
literal to 'char *' [-Wwritable-strings]
```

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

1.14.1 Passing array of char * to function

```
[ ]: // write a function similar to main
// main is not allowed to be defined in Jupyter Notebook C++
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;
}

```

[]: my_main(4, stuff);

```

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

```

1.15 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

1.15.1 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/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/overflow2/](#)
- buffer overflow can be exploited to execute arbitrary code
 - for details see: <https://github.com/rambasnet/SoftwareSecurity>

1.16 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 them to sort data
 - typically studied in algorithm courses
- 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

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

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

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

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

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

```
[ ]: // 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>());
```

```
[ ]: stu_grades
```

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

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

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

```
[ ]: words
```

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

1.17 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
- step through the algorithm here: <https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BubbleSort.html>
- 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

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

using namespace std;
```

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

```
[ ]: 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 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;
    }
}
```

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

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

```
[ ]: numbers
```

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

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

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

```
[ ]: values
```

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

1.18 Two-dimensional array

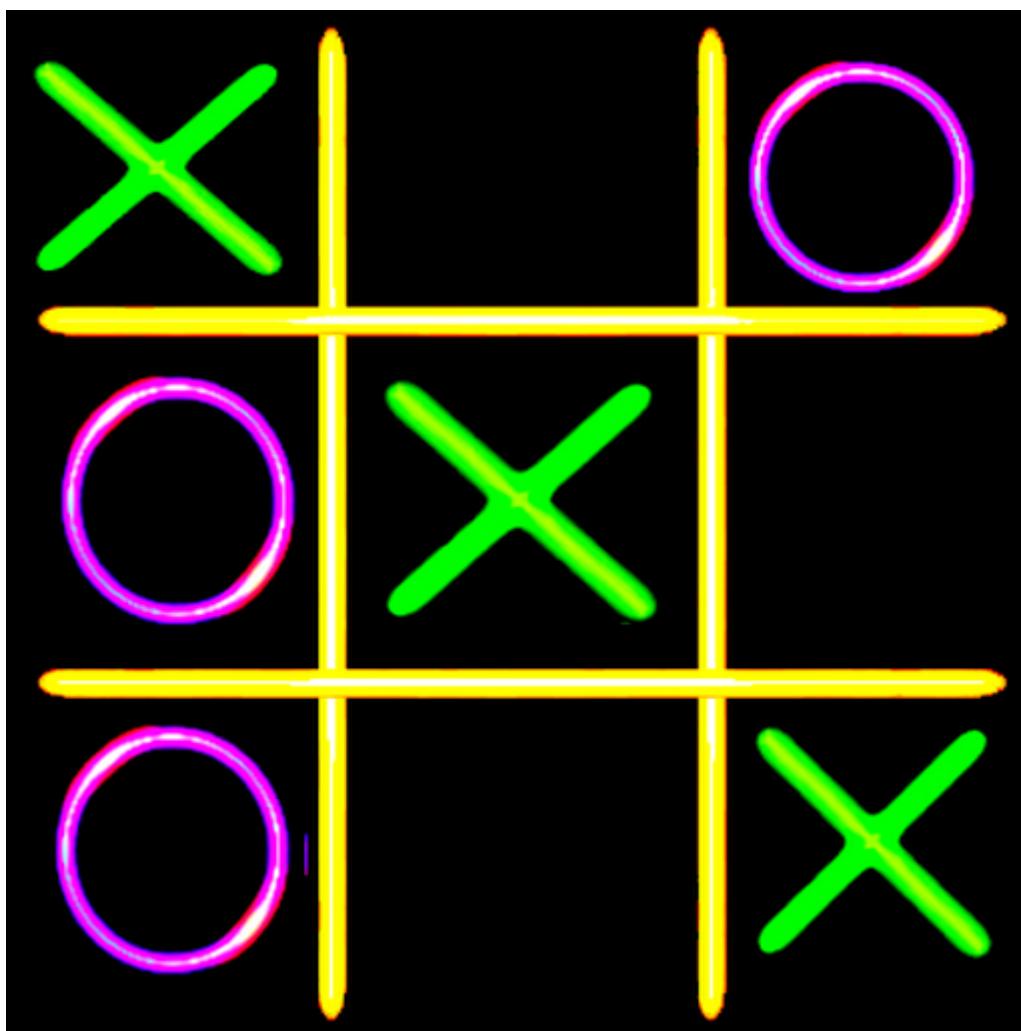
- two dimensional array is a useful construct to store data of 2-D in nature
 - table with row and column (representing 2-d board games), cartesian coordinates, etc.
- storing 3-D and beyond is also possible
 - 3-D array is used in video games to store graphics information
- syntax to declare 2-D array:

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

- 2-D array can be both static and dynamic

1.18.1 Tic-tac-toe game application

- represent 2-D tic-tac-toe board



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

```

using namespace std;

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

[]: // 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
}

[]: 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;
    }
}

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

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

```

| | | |

| | | |

| | | |

```

[]: // 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';

```

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

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

```
[ ]: // 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  
}
```

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

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

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

Congrats X! You win!!

1.19 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

1.20 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

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

```
[ ]: 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;
}
```

```
[ ]: 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";
}
```

```
[ ]: 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

1.21 Kattis problems for demo

- Personality Test - <https://open.kattis.com/problems/personalitytest>
- Boule Judge - <https://open.kattis.com/problems/boulejudge>

1.22 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
 - Falling Apart - <https://open.kattis.com/problems/fallingapart>
 - Statistics - <https://open.kattis.com/problems/statistics>
 - Line Them Up - <https://open.kattis.com/problems/lineup>
 - No Thanks! - <https://open.kattis.com/problems/nothanks>
 - Connect-N - <https://open.kattis.com/problems/connectn>
 - Hint: 2-D Array - simply check 4 winning ways from each B or R char - just like in tic-tac-toe
 - Milk Mystery - <https://open.kattis.com/problems/milkmystery>
 - Hint: max of all consecutive subset sums
 - Tic-Tac-Toe Solver - <https://open.kattis.com/problems/tictactoesolver>
 - Hint: 2-D Array - simply check 8 winning ways from each X or O char - just like in tic-tac-toe
 - Sudoku Verify - <https://open.kattis.com/problems/sudokuverify>
 - Hint: 2-D Array - check each row, column and 3x3 sub-grid for numbers 1 to 9
 - Alphabetical Aristocrats - <https://open.kattis.com/problems/alphabeticalaristocrats>
 - Hint: use 2D dynamic array to store the actual name and name to sort and sort them using built-in sort method

1.23 Summary

- learned about array and types of arrays
- passing array to functions
- similarity between array and pointers in terms of using memory addressses
- 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