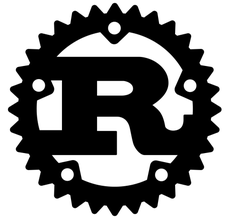
INTRO TO MODERN C++

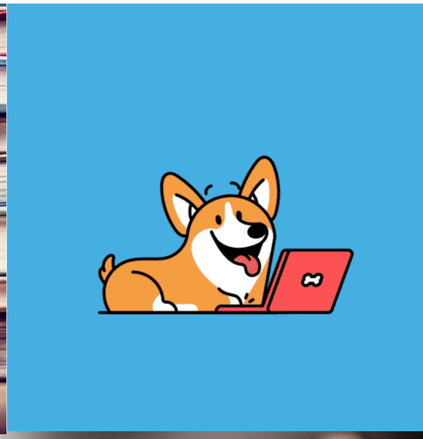
ABOUT ME

- Ahmad Beirkdar
- C++ Enthusiast
- Compilers, language design, Modern C++
- Corgis! Coffee! Star Trek!

- Reach me @:
  - Github: @ahmadbeirkdar
  - Twitter: @ahmadbeir
  - Blog: http://ahmad.ltd
  - Discord: @1337#0001



julia



OUTLINE

WORKSHOP OUTLINE

Part 1:

- What? Why? Modern C++?
- Types, and variables
- Some containers: vector, array
- Control flow
- Objects, Namespaces, Enums

Part 2:

- Templates
- Memory Management
- Constant Expressions

PREREQUISITES

WHAT I ASSUME YOU KNOW OR HAVE

- Control flow - If statements, loops, etc..
- Functions - Functions & Subroutines
- Object Programming
- Ryerson: CPS109, & a small part of CPS209
- Text Editor
- C++ Compiler
- For now, https://repl.it

WHAT!? WHY!? MODERN C++!?

WHAT?

"C++ the newest old language" - Matt Godbolt

- General purpose programming language
- Multi-paradigm: procedural, OO, generic, functional, etc..
- Features: OO, template meta programming, compile time, etc..
- Standardized by ISO, see: https://isocpp.org
- Wide range of standard library facilities, see: https://en.cppreference.com/w/

WHY?

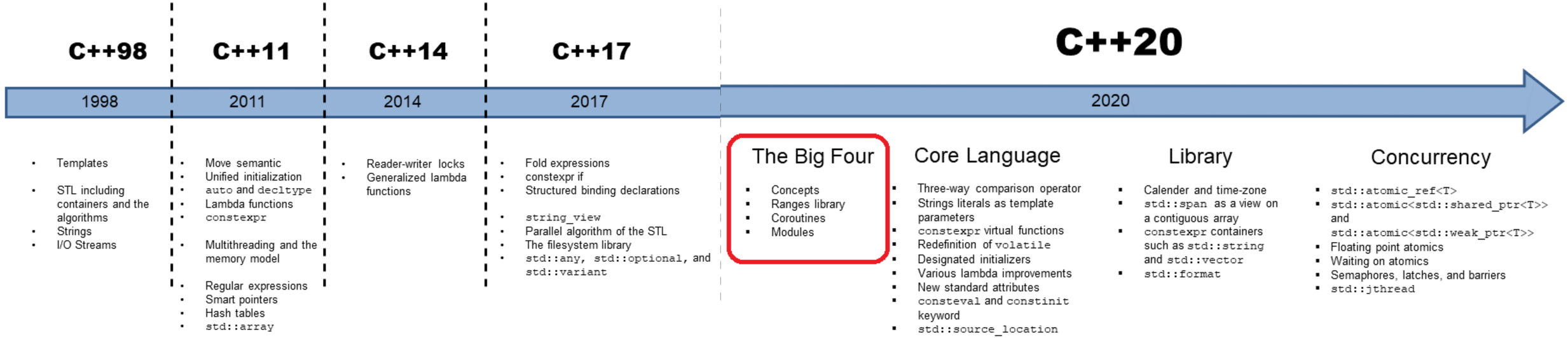
"The standard library saves programmers from having to reinvent the wheel." — Bjarne Stroustrup

- Used everywhere: AI, machine learning, Data Science, GPU programming, video games, cloud, etc...
- Allows for reusable, clean and efficient code
- Fast runtime, and improving

MODERN C++?

"The standard library saves programmers from having to reinvent the wheel." — Bjarne Stroustrup

- Refers to C++11 standard and beyond, (C++11,C++14,C++17,C++20)



Source: Rainer Grimm

C++ CORE GUIDELINES

"The standard library saves programmers from having to reinvent the wheel." — Bjarne Stroustrup

- Use Modern C++ effectively
- Tried methods, for reusability, safety and longevity.
- See: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

TYPES AND VARIABLES

BASICS

"The most important single aspect of software development is to be clear about what you are trying to build." — Bjarne Stroustrup

- Statically typed language
- Each variable must have a type
- Types are determined at compile time

- Integers: short, int, long. Unsigned ex, unsigned int.
- Real numbers: float, double
- Characters: char, unsigned char
- Boolean: bool

AUTO

"The most important single aspect of software development is to be clear about what you are trying to build." — Bjarne Stroustrup

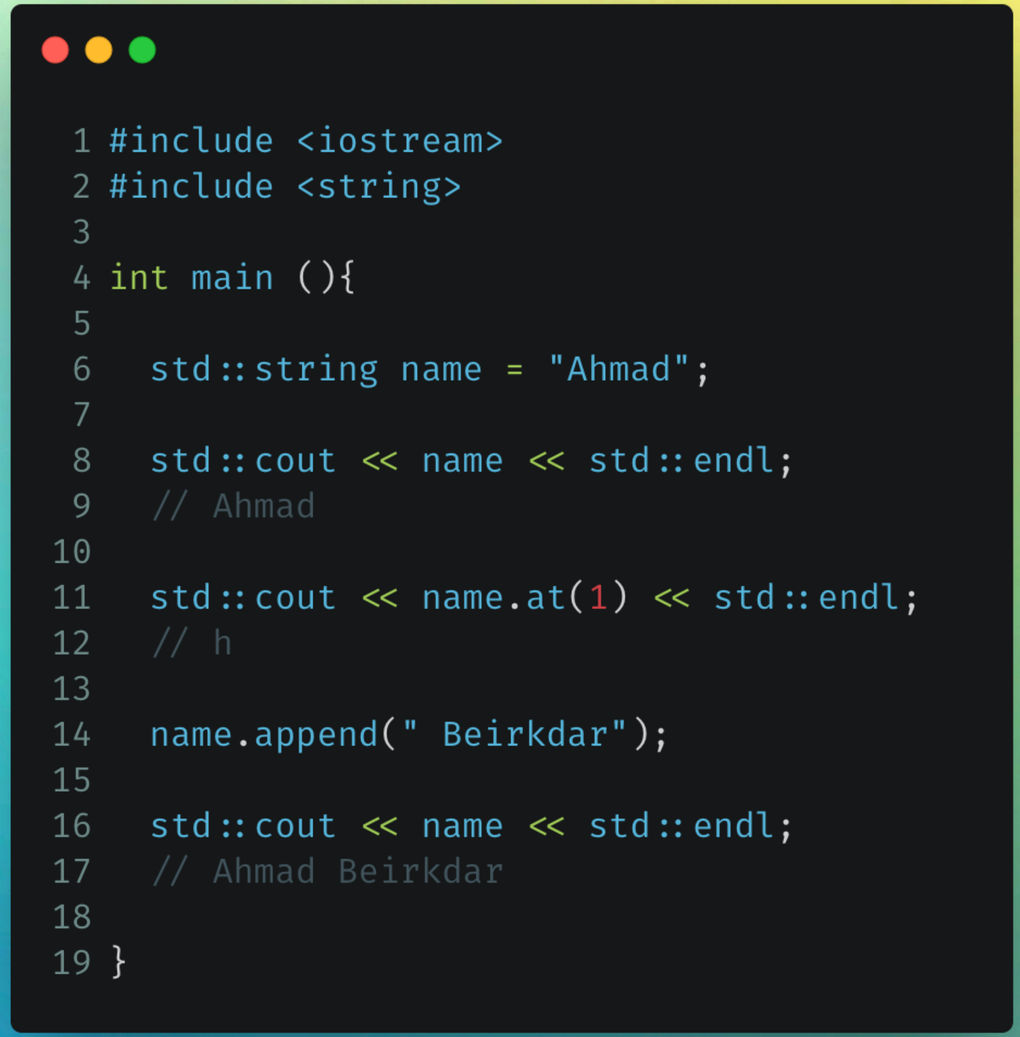
- Asking the compiler to determine the type. Aka, type deduction
- Occurs during compile time
- Used for "ugly" types

```cpp
#include <iostream>

int main(){

  int a = 123;
  int b = 123.923;

  std::cout << "a = " << a << " b = " << b << std::endl;
  // a = 123 b = 123


  double c = 255.123;
  bool d = true;
  char e = 'e';
  std::cout << "c = " << c << " d = " << d << " e = "<< e << std::endl;
  // c = 255.123 d = 1 e = e

}
```

```cpp
auto a = 1123; // Deduced to an integer
auto b = 123.123; // Deduced to a floating point number
auto flag = true; // Deduced to a boolean
auto c = 'c'; // Deduced to a char
```

STRINGS

- Not a fundamental type
- Included in the standard library, in string
- See: https://en.cppreference.com/w/cpp/string/basic_string



```cpp
#include <iostream>
#include <string>

int main (){

  std::string name = "Ahmad";

  std::cout << name << std::endl;
  // Ahmad

  std::cout << name.at(1) << std::endl;
  // h

  name.append(" Beirkdar");

  std::cout << name << std::endl;
  // Ahmad Beirkdar

}
```

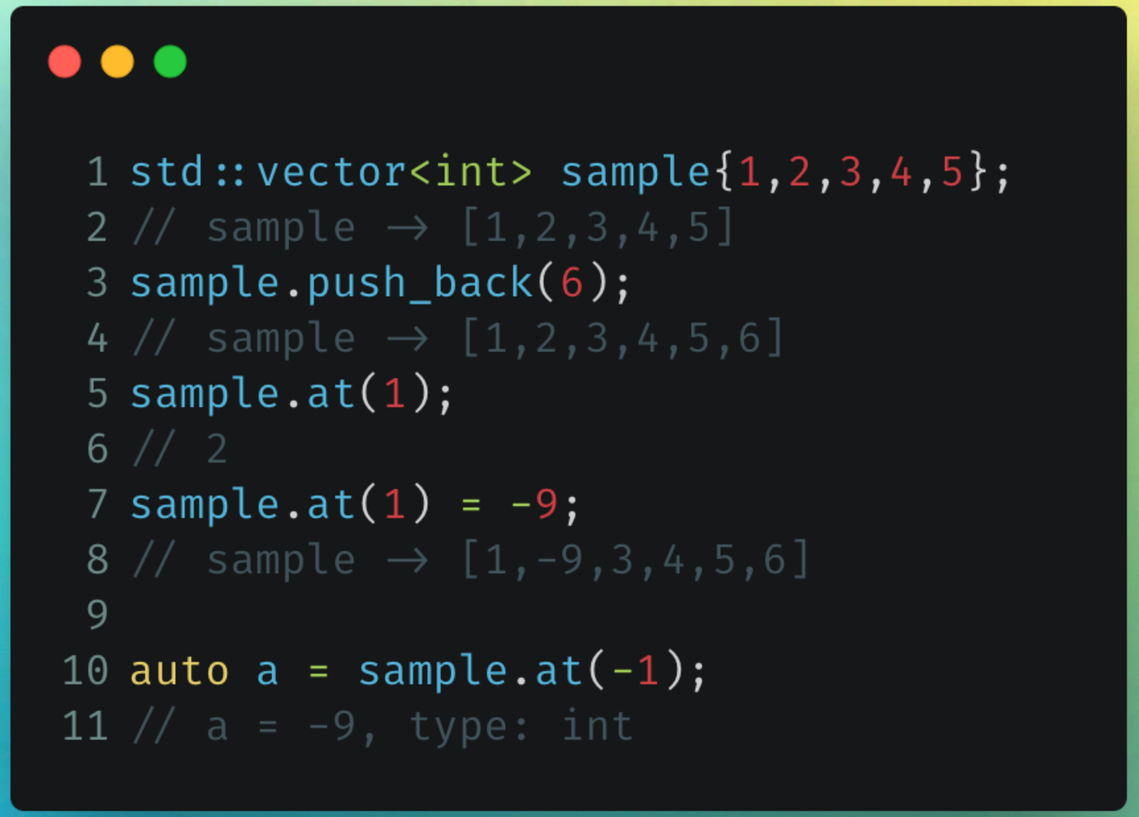
CONTAINERS

CONTAINERS

- Included in the standard library
- Work with standard algorithms beautifully
- Self managed
- “std::vector and std::array are your friends”
- See: https://en.cppreference.com/w/cpp/container

VECTORS

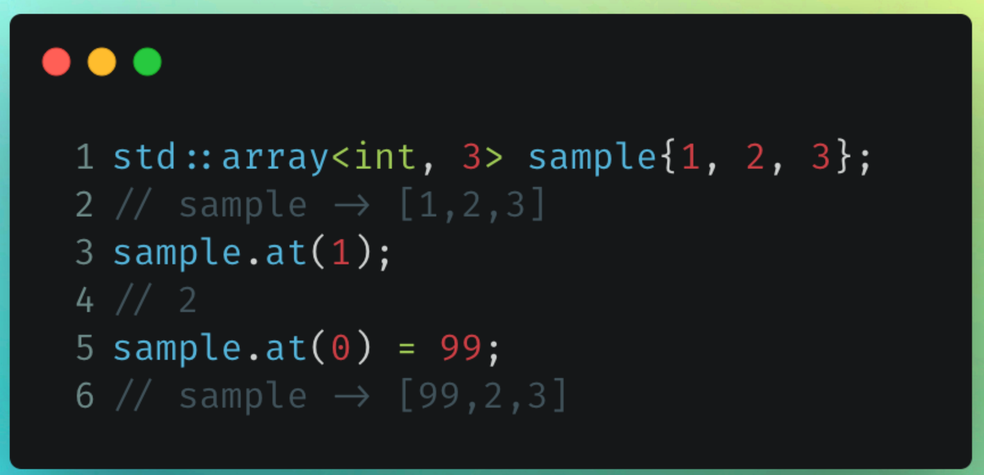
- Dynamic array
- Random access
- Elements are consecutive in memory
- Include vector to use, “#include <vector>”
- See: https://en.cppreference.com/w/cpp/container/vector



```cpp
std::vector<int> sample{1,2,3,4,5};
// sample → [1,2,3,4,5]
sample.push_back(6);
// sample → [1,2,3,4,5,6]
sample.at(1);
// 2
sample.at(1) = -9;
// sample → [1,-9,3,4,5,6]

auto a = sample.at(-1);
// a = -9, type: int
```

ARRAYS

- Static Array, can never grow
- Random access
- No dynamic memory allocation
- Include array to use, “#include < array >”
- Usage: std::array<type, size>
- See: https://en.cppreference.com/w/cpp/container/array


std::array<int, 3> sample{1, 2, 3};
// sample → [1,2,3]
sample.at(1);
// 2
sample.at(0) = 99;
// sample → [99,2,3]

CONTROL FLOW

IF STATEMENTS

```
1 if(a == 1){
2   // a is equal to 1
3 }
4 else if(b == 2) {
5   // b is equal to 2
6 }
7 else{
8  // Both conditions above are false
9 }
```

“RAW” LOOP

```cpp
for(auto i = 0; i < 9; i++){
 std::cout << i;
}
// 012345678
```

```cpp
std::vector<int> sample{1,2,3,4,5,6};
for(auto i = 0; i < sample.size(); i++){
  std::cout << sample.at(i);
}
// 123456
```

RANGE FOR LOOP

- More expressive, added safety
- Used for containers included in the standard library, or anything with iterators

```cpp
std::vector<int> sample{1,2,3,4,5,6};
for(int i : sample){
  std::cout << i;
}
// 123456
```

```cpp
std::vector<int> sample{1,2,3,4,5,6};
for(const auto& i : sample){
  std::cout << i;
}
// 123456
```

```cpp
std::vector<int> sample{1,2,3,4,5,6};
for(auto i : sample){
  i = 1;
}
// 123456
```

```cpp
std::vector<int> sample{1,2,3,4,5,6};
for(auto& i : sample){
  i = 1;
}
// 111111
```

FUNCTIONS

```cpp
int add(int a, int b){
  return a + b;
}
// or
auto add(int a, int b) -> int {
  return a + b;
}

// Better form
audo add(const int& a, const int& b) -> int {
  return a + b;
}
```

```cpp
void fun1(int a){
  a += 3;
}

void fun2(int& a){
  a += 3;
}

int main(){
  int x = 9;

  fun1(x);
  std::cout << x << std::endl;
  // 9
  fun2(x);
  std::cout << x << std::endl;
  // 12
}
```

OBJECTS

CLASSES

```cpp
class Sample {
  public:
    // Public methods and variables
  private:
    // Private methods and variables
}
```

```cpp
class Person {
  public:
    // Public methods and variables
    int get_age() const {
      return age;
      // return this->age;
    }
    void set_age(const int age_in){
      age = age_in;
    }

    std::string get_name() const {
      return name;
    }
    void set_name(const std::string name_in){
      name = name_in;
    }

  private:
    // Private methods and variables
    std::string name;
    int age;
}
```

CONSTRUCTOR

- “Constructs” the object

```cpp
class Person {
  public:
    // Person() implictly exists here
  private:
    // Bad Practice
    // std::string name;
    // int age;

    // Good Practice
    std::string name{};
    int age{};
}

auto a = Person();
```

```cpp
class Person {
  public:
    Person(const std::string& name_in, const int& age_in)
      : name(name_in), age(age_in) {
            // You can do stuff here if you please.
        };
    // Person() does not exist anymore as a
    //    another constructor has been defined.
    // Do we still want a default constructor?
    // So we do this:
    // Person()=default;
  private:
    // Bad Practice
    // std::string name;
    // int age;

    // Good Practice
    std::string name;
    int age;
}

auto a = Person(); // Not valid no more.
auto b = Person("Ahmad",20);
```

NAMESPACES

NAMESPACES

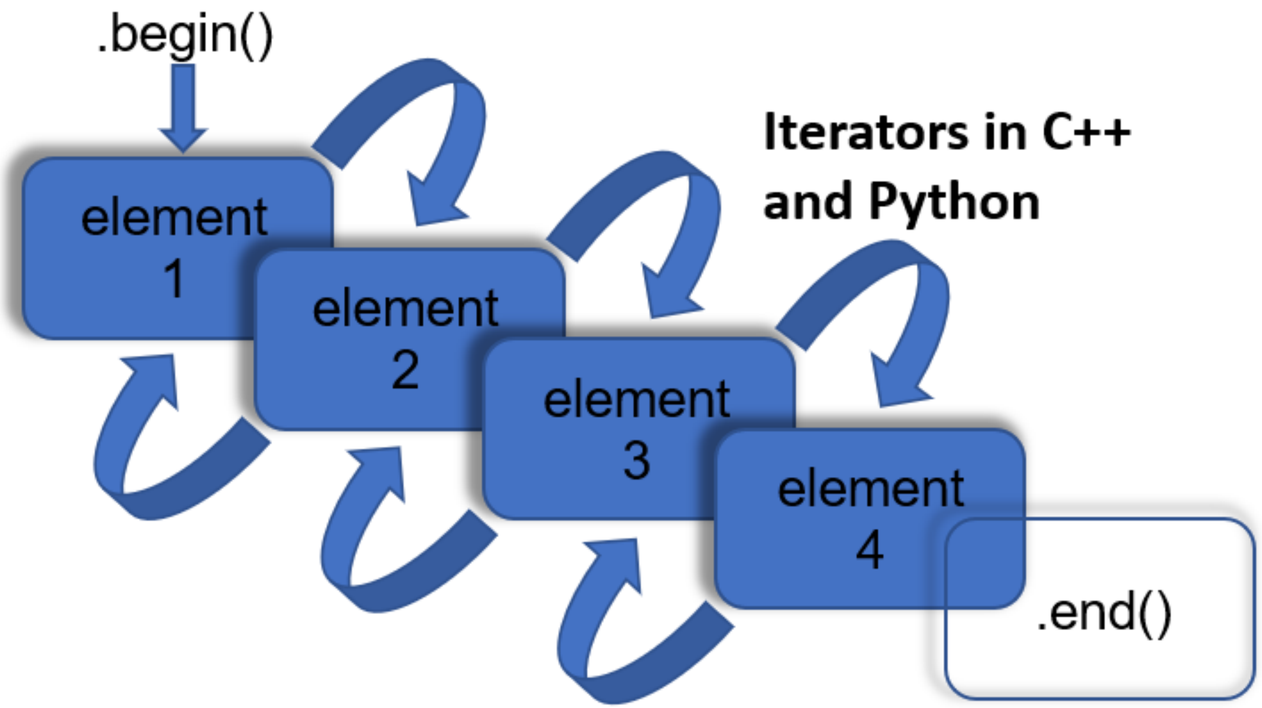
- Prevents name collision
- Ex, std::

```cpp
namespace util {
  auto remove_all_occurrences(std::string& str, std::string& sub_str) -> void {
    size_t m = 0;

    while((n = str.find(sub_str, n)) =/= std::string::npos)
      str.erase(n, sub_str.size());

  }
}

auto str = "Wow Modern C++ is SOOO NICE!";
util::remove_all_occurrences(str," ");
// str -> "WowModernC++isSOOONICE!"
```

ITERATORS

ITERATORS

- Crucial for connecting standard algorithms with the standard containers.
- Want to make your own container? Implement iterators!



```cpp
std::vector<int> vec{1,2,3,4,5,6,7,8};

auto it = vec.begin();
std::cout << *it << std::endl;
// 1

it++;
std::cout << *it << std::endl;
// 2

it--;
std::cout << *it << std::endl;
// 1
```

```cpp
std::vector<int> vec{1,2,3,4,5,6,7,8};

auto it = vec.begin();
while(it != vec.end()){
  std::cout << *it;
  it++;
}
// 12345678
```

LET'S SOLVE SOME LC!

977. SQUARES OF A SORTED ARRAY

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of **the squares of each number** sorted in non-decreasing order*.

Example 1:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].
```

Example 2:

```
Input: nums = [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```

4. MEDIAN OF TWO SORTED ARRAYS

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Example 3:

```
Input: nums1 = [0,0], nums2 = [0,0]
Output: 0.00000
```

Example 4:

```
Input: nums1 = [], nums2 = [1]
Output: 1.00000
```

Example 5:

```
Input: nums1 = [2], nums2 = []
Output: 2.00000
```