CSC 212: Data Structures and Abstractions

Dynamic (Growing/Resizing) Arrays

Jonathan Schrader

[credit Marco Alverez]

Department of Computer Science and Statistics
University of Rhode Island

Fall 2022

Quick notes

- **Review** Pointers and Dynamic Memory Allocation
- Programming Assignment 1
  - ✓ Due tonight (9/16)
  - ✓ Submit up to 3 days late @10% penalty per day late
- Programming Assignment 2
  - ✓ Recommend learning vector of pairs std::vector<std::pair>>
  - ✓ Introducing at the end of this lecture

Arrays

Arrays

- An array is a **contiguous** sequence of elements of the **same type**
- Each element can be accessed using its **index**

array name: A

array length: n

0	1	2	3			n-1
A[0]	A[1]	A[2]	A[3]	…		A[n-1]

all elements of the same data type

Declaration

```
// array declaration by specifying size
int myarray1[100];

// can also declare an array of user specified
// size (must be const for many compilers!)
int n = 8;
int myarray2[n];

// can declare and initialize elements
double arr[] = { 10.0, 20.0, 30.0, 40.0 };
// compiler figures the right size

// a different way
int arr[5] = { 1, 2, 3 };
// compiler creates an array of length 5 and
// initializes first 3 elements
```

Static arrays

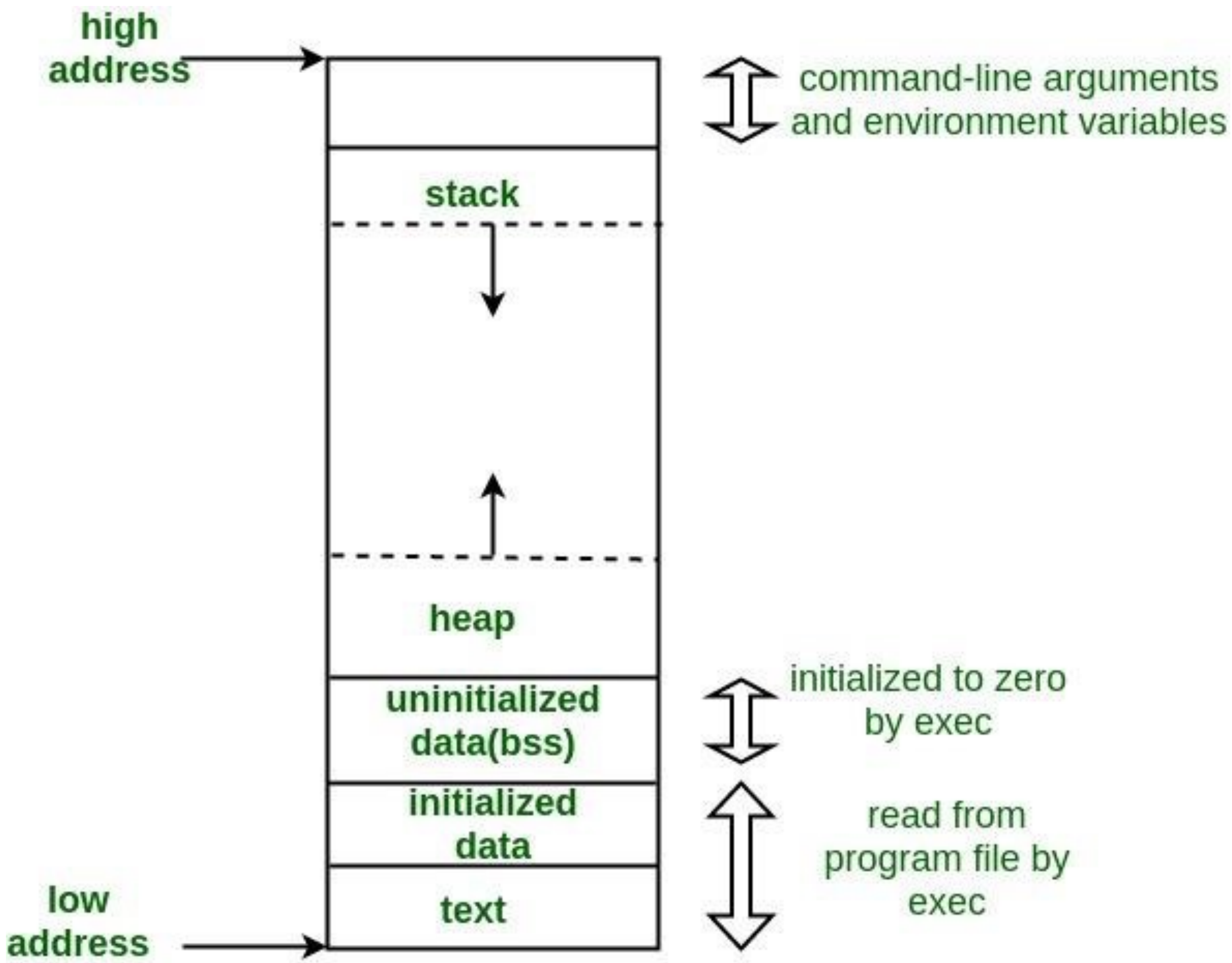
- So far … we have seen examples of arrays, **allocated in the stack** (fixed length)

```
// array declaration by specifying size
int myarray1[100];
```

- You can allocate memory dynamically, **allocated in the heap** (still fixed length)

```
int *myarray = new int[100];
// ...
// work with the array
// ...
delete [] myarray;
```

Memory layout of C/C++ programs



https://www.geeksforgeeks.org/memory-layout-of-c-program/

Live coding demo (static arrays — stack and heap)

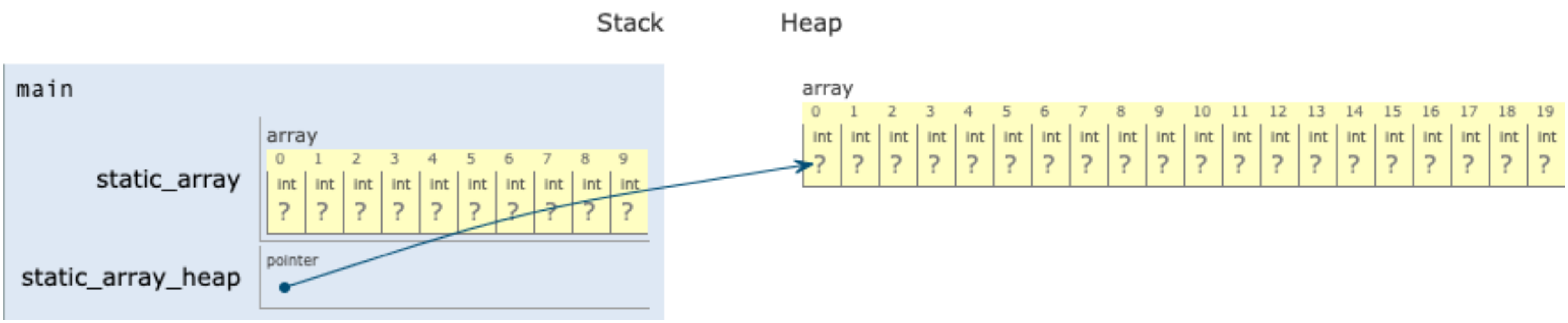
C++ (gcc 4.8, C++11)
EXPERIMENTAL! known limitations

```
int main() {
    float var1;
    double var2;
    int static_array[10];
    int *static_array_heap = new int [20];
    // ...
    // work with the array
    // ...
    delete [] static_array_heap;
}
```

Edit this code

line that just executed
next line to execute

View this code in PythonTutor!



credit: pythontutor.com

A few notes …

- Creating variables in the stack:
  - ✓ variables are automatically created and freed
  - ✓ variables only exist while the function is running
  - ✓ faster and good for small local variables
- Allocating memory in the heap:
  - ✓ memory is allocated at runtime
  - ✓ programmer is responsible for allocating/deallocating memory
  - ✓ variables can be accessed globally (in the program)
  - ✓ memory may become fragmented
  - ✓ slower but good for large variables

What if … ?

- We don't know the max size of an array before running the program
  - ✓ user specified inputs/decisions
  - ✓ e.g. read an image or video and display
- The sequence changes over time (during the execution of the program)
  - ✓ e.g. you develop a text editor and represent the sequence of characters as an array

Which data structure (studied so far) would you use on each case?

Dynamic Arrays (resizing, growing)

Dynamic Arrays

- Dynamically allocated arrays that change their size over time
  - ✓ can **grow** automatically
  - ✓ can **shrink** automatically
- Operations on arrays (we could have more, but these are enough for the purposes of this lecture)
  - ✓ append
  - ✓ remove_last
  - ✓ get — $\Theta(1)$
  - ✓ set — $\Theta(1)$

First try …

- Start with an empty array
- For every append:
  - ✓ increase the size of the array by 1 then write the new element
- For every remove_last:
  - ✓ remove the last element and then decrease the size of the array by 1
- Demo …

Analyzing the cost (grow by 1)

- Count array accesses (reads and writes) of adding first n elements
  - ✓ will ignore the cost of allocating/deallocating arrays

n	append	copy

each row indicates the number of **reads and writes** necessary for appending an element into an **existing array of length n**

$$n + \sum_{i=0}^{n-1} 2i = n + n^2 - n$$

$$\Theta(n^2)$$

Lets try again ...

- If array is **full**, create an array of **twice the size**
  - ✓ **repeated doubling**
- If array is **one-quarter full**, **halve the size**
  - ✓ more efficient
  - ✓ why not halving when array is one-half full?

append - remove - append - remove - append - remove…

- Demo ...

Analyzing the cost (doubling the array)

- Count array accesses (reads and writes) of adding first $n = 2^i$ elements
  - ✓ will ignore the cost of allocating/deallocating arrays

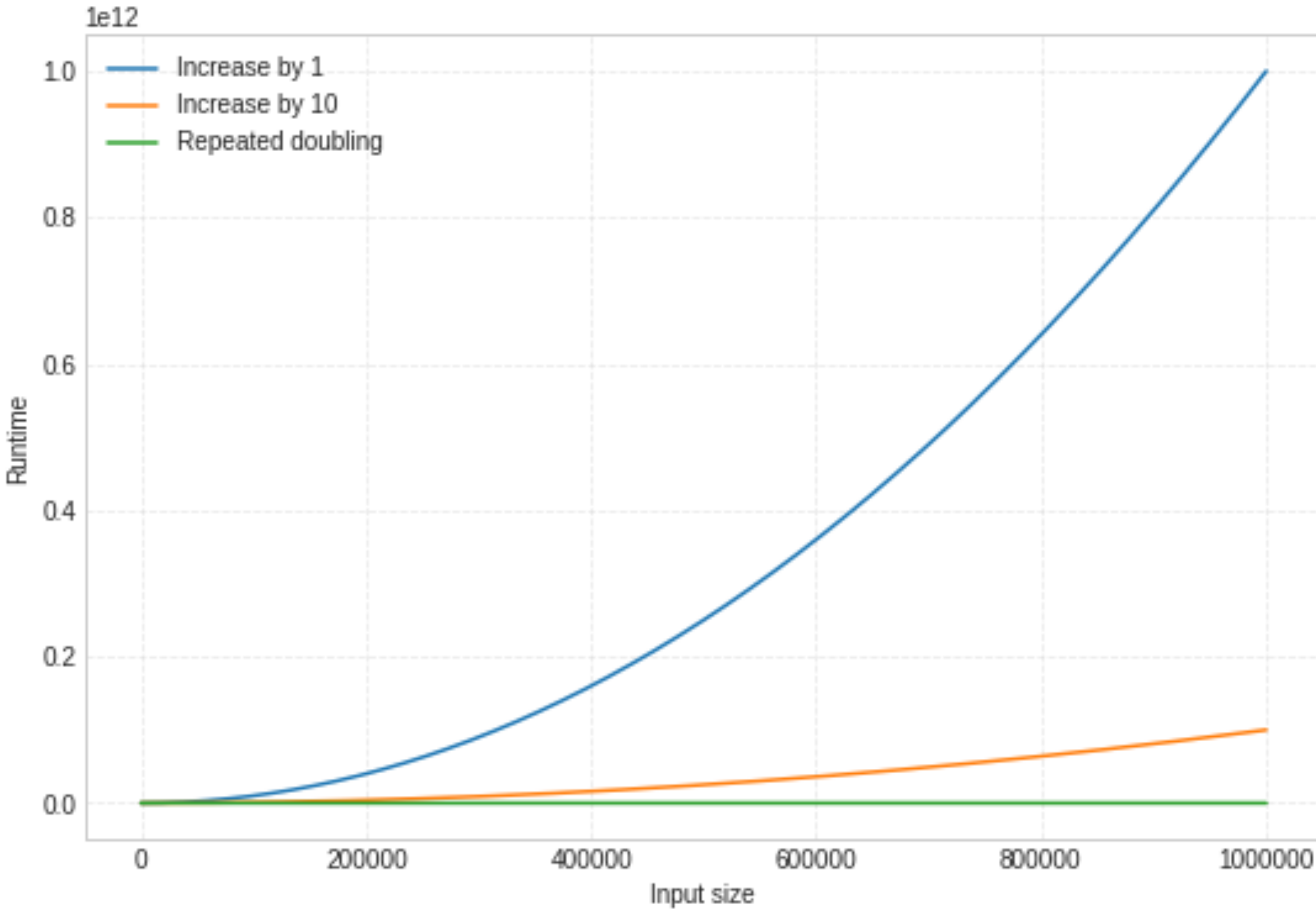
n	append	copy

each row indicates the number of **reads and writes** necessary for appending an element into an **existing array of length n**

$$n + \sum_{i=1}^{\log n} 2^i = n + 2^{\log n + 1} - 1$$

$$\Theta(n)$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}$$


1e12
Increase by 1
Increase by 10
Repeated doubling
1.0
0.8
0.6
0.4
0.2
0.0
Runtime
0
200000
400000
600000
800000
1000000
Input size

Worst-case and average-case

- Analysis for appending **a single element** using increase-by-1

- Analysis for appending **a single element** using repeated doubling