C++ Arrays

Stack Array Implementation

Outline

- Stack array implementation
- Array review
- Arrays in C++

Implementing a Stack

With an Array

Array Implementation

- Record index that represents top of the stack
  - push – increment index
  - pop – decrement index
- Push and pop run in constant time

O(1)

6	1	7			
0	1	2	3	4	5

index of **top** is current size – 1

```
Stack st();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
st.push(13); //top = 4
st.pop(); //top = 3
st.pop(); //top = 2
```

Array Implementation Summary

- Simple array implementation
  - *push* and *pop* performed in constant time
    - Independent of the number of items in the stack
- Once the array is full
  - No new values can be inserted or
  - A new, larger, array is created
    - And the existing items copied to this new array
    - Known as a dynamic array

Implementation decision

How much bigger?

Array Review

Arrays

- Arrays contain identically typed values
  - Which are stored sequentially in main memory
- Values are stored at specific numbered positions in the array called **indexes**
  - The first value is stored at index 0, the second at index 1, the *i*th at index *i*-1, and so on
  - The last item is stored at position *n*-1, assuming that the array is of size *n*
  - Referred to as zero-based indexing

Array Indexing

- `int arr[] = {3,7,6,8,1,7,2};`
  - Creates an integer array with 7 elements
- To access an element, refer to the array name and the index of that element
  - `int x = arr[3];` assigns the value of the fourth array element (8) to x
  - `arr[5] = 11;` changes the sixth element of the array from 7 to 11
  - `arr[7] = 3;` results in an error because the index is out of bounds

In C++ could result in a segmentation fault or logic error

An IDE may raise a debug error after termination

index	value
0	3
1	7
2	6
3	8
4	1
5	11
6	2

Arrays and Main Memory

```
int grade[4];
grade[2] = 23;
```

Declares an array variable of size 4

Assigns 23 to the third element of *grade*

The array is shown as not storing any values – although this isn't really the case

grade is a *constant pointer* to the array and stores the address of the array

But how does the program know where `grade[2]` is?

Memory Addresses

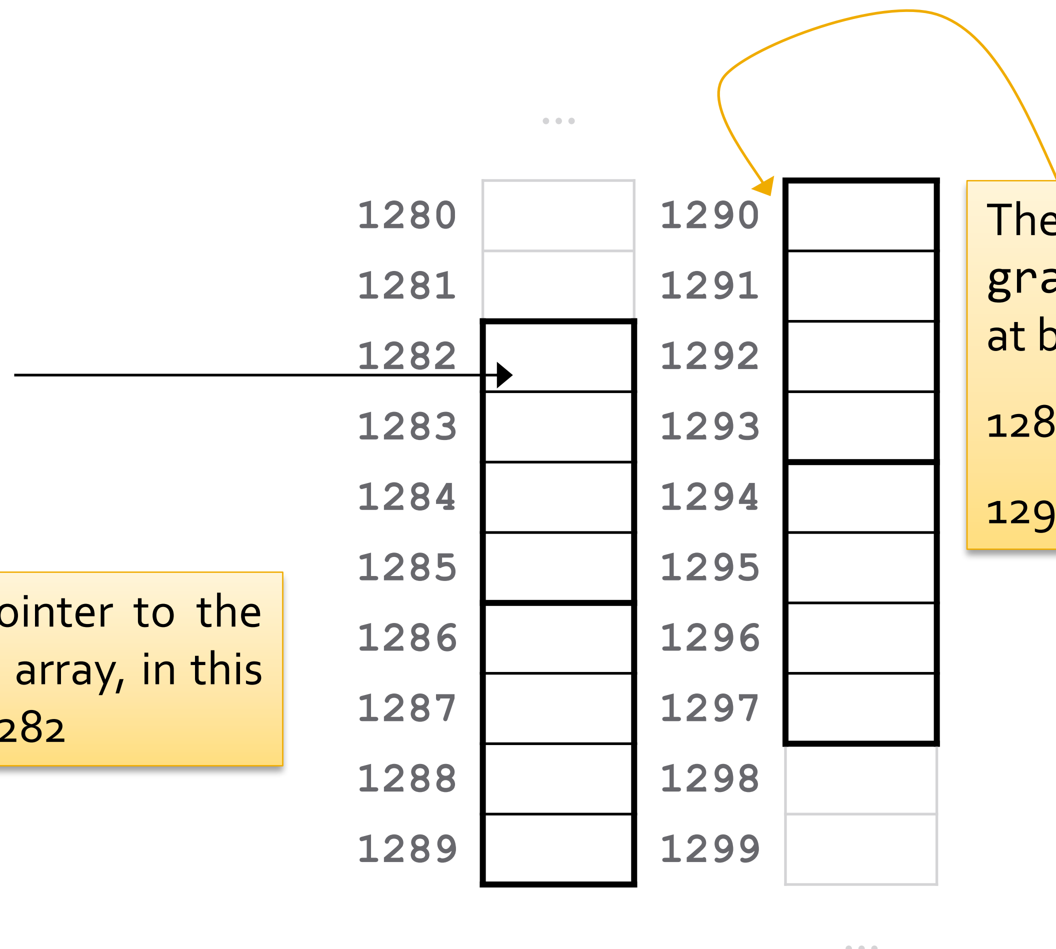
- Access to array elements is very fast
- An array variable refers to the array
  - Stores the main memory address of the first element
  - The address is stored as number, with each address referring to one byte of memory
    - Address 0 would be the first byte
    - Address 1 would be the second byte
    - Address 20786 would be the twenty thousand, seven hundred and eighty seventh byte
    - …

Offset Calculations

- Consider `grade[2] = 23;`
  - How do we find this element in the array?
- What do we know
  - The **address** of the first array element
  - The **type** of the values stored in the array
    - Therefore, the **size** of each of the array elements
  - The **index** of the element to be accessed
- We can calculate the address of the element to be accessed, which equals
  - address of first element + (index * type size)

Offset Example

grade



Stores a pointer to the start of the array, in this case byte 1282

The integer stored at `grade[2]` is located at byte:

1282 + 2 * 4 =

1290

Passing Arrays to Functions

- Array variables are pointers
  - An array variable passed to a function passes the **address** of the array
    - And **not** a **copy** of the array
- Changes made to the array by a function are made to the original (one-and-only) array
  - If this is not desired, a copy of the array should be made within the function

Array Positions

- What if array **positions** carry meaning?
  - An array that is sorted by name, or grade or some other value
  - Or an array where the position corresponds to a position of some entity in the world
    - An array that represents a bookcase
- The ordering should be maintained when elements are inserted or removed

Ordered Array Problems

- When an item is inserted at a given index either
  - Write over the element or
    
  - Move the element, *and all elements after it*, **up** one position
    
- When an item is removed either
  - Leave *gaps* in the array, i.e. array elements that don't represent values or
    
  - Move all the values after the removed value **down** one index
    

Arrays are Static

- The size of an array must be specified when it is created
  - And cannot then be changed
- If the array is full, values cannot be inserted
  - There are, time consuming, ways around this
  - To avoid this, we can make arrays much larger than they are needed
  - However, this wastes space

Array Summary

- Good things about arrays
  - Fast, random access, of elements using a simple offset calculation
  - Very storage space efficient, as little main memory is required other than the data itself
  - Easy to use
- Bad things about arrays
  - Slow deletion and insertion for ordered arrays
  - Size must be known when the array is created
    - Or possibly beforehand
    - An array is either full or contains unused elements

Arrays in C++

Another Review

Declaring (Static) Arrays

- Arrays are declared just like single variables except that the name is followed by []s
- The []s should contain the size of the array which must be a constant or literal integer
  - `int age[100];`
  - `const int DAYS = 365;`
  - `double temperatures[DAYS];`

Some development environments allow the size of arrays to be specified with a variable, but this is not supported by the C++ standard

Initializing Arrays

- Arrays can be initialized
  - One element at a time
  - By using a for loop
  - Or by assigning the array values on the same line as the declaration
    - `int fib[] = { 0,1,1,2,3,5,8,13 };`
  - Note that the size does not have to be specified since it can be derived

Array Assignments

- A new array *cannot* be assigned to an existing array

```
int arr1[4];
int arr2[4];
int n = 4;
…
arr1 = arr2; //can't do this!
arr1 = {1,3,5,7}; //… or this …
```

- Array *elements* can be assigned values

```
for(int i=0; i < n; i++) {
  arr1[i] = arr2[i];
}
```

Array Parameters and Arguments

- Array parameters looks like array variables
  - Except that the size is not specified
- C++ arrays do not have a size member
  - Or any members, since they are not classes
  - Functions with array parameters often need a parameter for the size of the array

```
int sum(int arr[], int n) { //… };
```

- Array variables are passed to functions by name
  - Do not include []s

```
int grades[200];
// …
sum(grades, 200);
```

What's in an Array Variable

- An array variable records the address of the first element of the array
  - This address cannot be changed after the array has been declared
  - It is therefore a *constant pointer*
- This is why existing array variables cannot be assigned new arrays
- And why arrays passed to functions may be modified by those functions

Memory in C++

- C++ gives programmers a lot of control over where variables are located in memory
- There are three classes of main memory
  - Static
  - Automatic
  - Dynamic
- Automatic memory is generally used to allocate space for variables declared inside functions
  - Unless those variables are specifically assigned to another class of storage

Arrays and Memory in C++

- Arrays are allocated space in automatic storage
  - At least as they have been discussed so far, and
  - Assuming that they were declared in a function
- Variables allocated space on the call stack are not permitted to change size
  - As stack memory is allocated in sequence and this could result other variables being over-written

Dynamic Memory

- What happens if we want to determine how much memory to allocate at *run time*?
  - Stack memory size is determined at compile time so it would need to be allocated somewhere else
  - Let's call *somewhere else* the *heap* or the *free store*
- We still need automatic variables that refer or point to the dynamically allocated memory
  - In C++ such variables are *pointers*

Variables in Dynamic Memory

- Create a variable to store an address
  - A pointer to the type of data to be stored
  - Addresses have a fixed size
  - If there is initially no address, it should be assigned a special value (*NULL* or *nullptr*)

```
int* arr = nullptr;
```

- Create new data in dynamic memory
  - When needed (i.e. at run time)

```
arr = new int[n];
```

- Assign the address of the data to the pointer
- This involves more a more complex management system than using automatic memory

Indexing Arrays in Dynamic Memory

- Arrays created in dynamic memory are indexed as normal

```
int* arr = new int[100];
for (int i=0; i < 100; ++i){
        arr[i] = i+1;
}
```

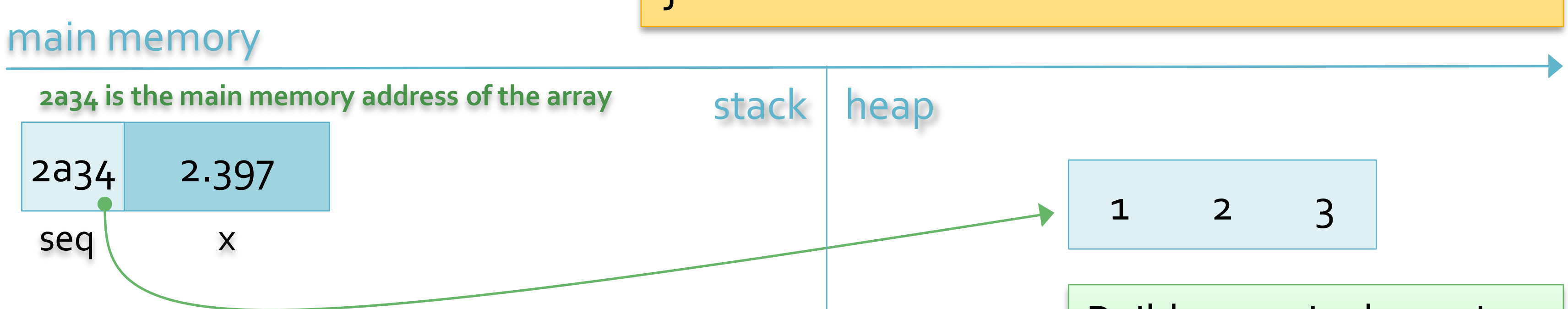
- Pointers to existing arrays in dynamic memory can be assigned new arrays

```
delete[] arr; //release memory
arr = new int[1000000];
```

A Dynamic Array

```
int* seq = NULL;
double x = 2.397;
seq = sequence(1, 3);
```

```
// Returns  pointer to array:
// {start, start + 1, … start + n-1}
int* sequence(int start, int n){
        int* result =  new int[n];
        for(int i=0; i < n; i++) {
                result[i] = start + i;
        }
        return result;
}
```



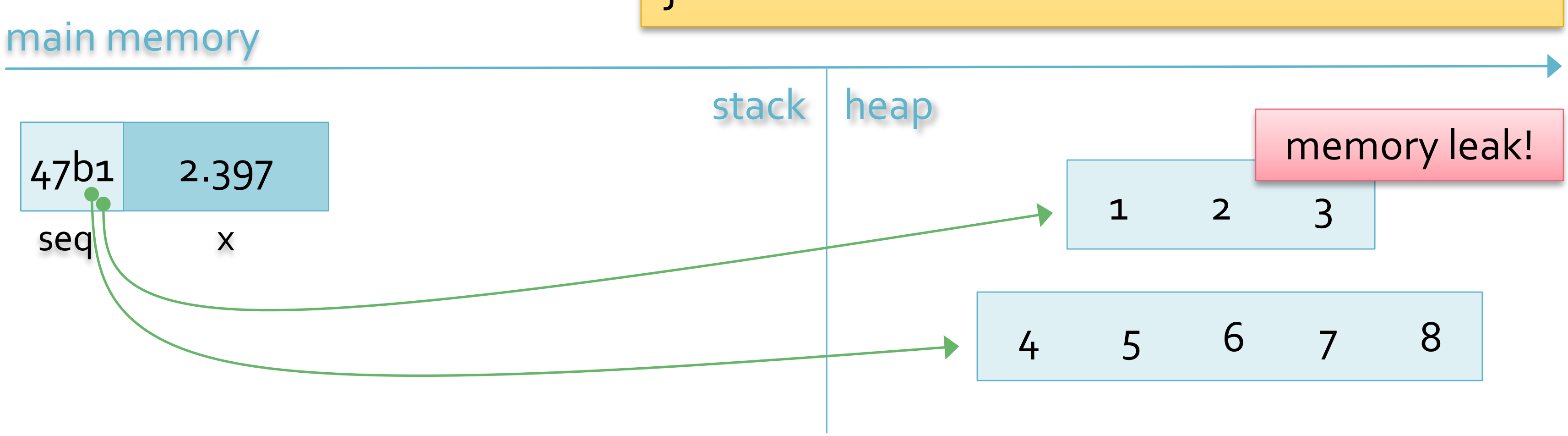
Builds array in dynamic storage (heap, free store)

A Dynamic Array

```
int* seq = NULL;
double x = 2.397;
seq = sequence(1, 3);
seq = sequence(4, 5);
```

no call to delete

```
// Returns  pointer to array:
// {start, start + 1, … start + n-1}
int* sequence(int start, int n){
       int* result =  new int[n];
       for(int i=0; i < n; i++) {
              result[i] = start + i;
       }
       return result;
}
```



Releasing Dynamic Memory

- When a function call is complete its stack memory is released and can be re-used
- Dynamic memory should also be released
  - Failing to do so results in a *memory leak*
- It is sometimes not easy to determine when dynamic memory should be released
  - Data might be referred to by more than one pointer
    - Memory should only be released when it is no longer referenced by *any* pointer

Dynamic vs Static

- When should a data object be created in dynamic memory?
  - When the object is required to change size, or
  - If it is not known if the object will be required
- Languages have different approaches to using static and dynamic memory
  - In C++ the programmer can choose whether to assign data to static or dynamic memory

Pointers and Arrays

- Elements of arrays can be accessed via their addresses, as well as their indexes
  - `int arr[] = { 10,20,30,40 };`
  - `cout << *(arr+2) << endl;` Prints 30
- Pointer arithmetic overloads the **+** operator
  - If *arr* is a pointer *arr* + 2 does **not** add 2 to the address stored in *arr*
  - It adds 2 * the **size** of the type that *arr* points to
- This technique can be useful for passing part of an array to a function Particularly recursive functions