Lecture 5
Arrays & Container Classes

EECS 281: Data Structures & Algorithms

Job Interview Questions

- Assume that a given array has a majority (>50%) element – find it
  in linear time using O(1) memory

  11 13 99 12 99 10 99 99 99

- Same for an array that has an element repeating at least n/3 times

  11 11 99 10 99 10 12 19 99

Arrays

"A third of the code for Projects 1 and 2 that crashed and I was asked to look at, crashed due to misuse of the C array syntax." - Fall 2012 TA

- Writing code that does not compile
- Using dynamic allocation incorrectly
  - Crashes & memory leaks
- Using dynamic allocation where it is not needed

How to write slow code and fail testcases (TLE): store too much data, spend time reading/writing it

A Contradiction in Terms

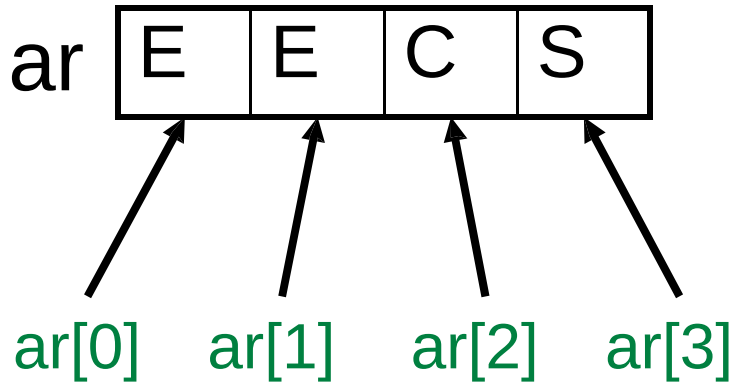
- You need to understand
  - How C arrays work, including multidimensional arrays
  - How C pointers work, including function pointers
  - How C strings work, including relevant library functions

They are great for code examples and HWs, come up at interviews & legacy code… but for projects:

- Avoid C arrays, use C++11 array<T> or vector<T>
- Avoid pointers (where possible)
  - Use STL containers, function objects, integer indices
- Use C++ string objects

Review: Arrays in C/C++

```
1   char ar[] = {'A', 'E', 'C', 'S'};
2   ar[0] = 'E';
3
4   char c = ar[2];
5   // now we have c=='C'
6
7   char *ptr = ar;
8   // now ptr points to EECS
9
10  ptr = &ar[1];
11  // now ptr points to ECS
```



same as ptr = ar + 1;
or ptr++;

- Allows random access in *O*(1) time
- Index numbering always starts at 0
- **No bounds checking**
- **Size of array must be separately stored**

2D Arrays in 1D (Arithmetic Arrays)

1D array

0	1	2	3	4	5	6	7	8

3x3 2D array

row \ column	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

column = index % num_columns
row = index / num_columns

index = row * num_columns + column

1D Index to 2D Row/Column

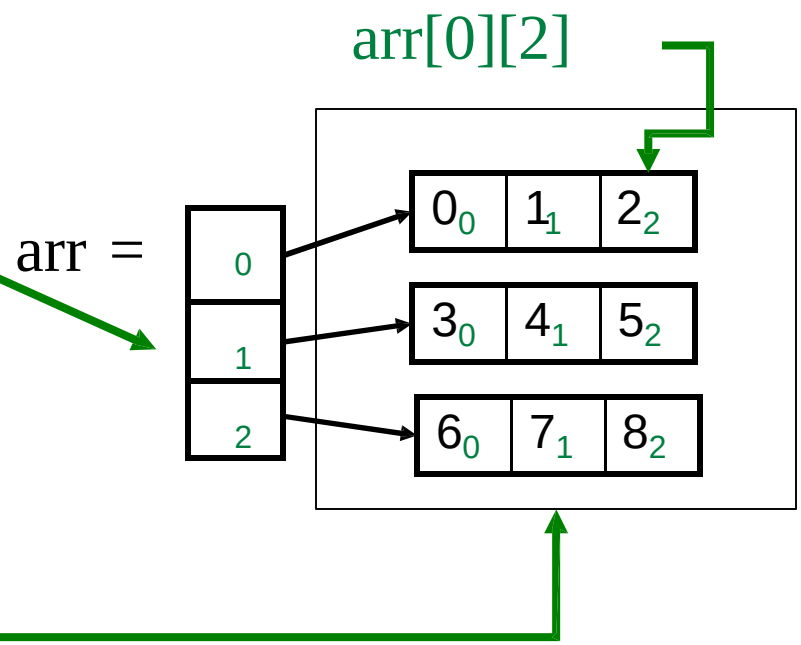
Index	Row	Column
2	2 / 3 = 0	2 % 3 = 2
3	3 / 3 = 1	3 % 3 = 0
7	7 / 3 = 2	7 % 3 = 1

2D Row/Column to 1D Index

Row	Column	Index
0	1	0 * 3 + 1 = 1
1	2	1 * 3 + 2 = 5
2	2	2 * 3 + 2 = 8

2D Arrays with Double Pointers

```
1   // Create array of rows
2   int **arr = new int * [3];
3
4   // For each row, create columns
5   for (int r = 0; r < 3; r++) {
6       arr[r] = new int[3];
7   }
8
9   int val = 0;
10  // For each row
11  for (int r = 0; r < 3; r++) {
12      // For each col
13      for (int c = 0; c < 3; c++) {
14          arr[r][c] = val++;
15      }
16  }
```

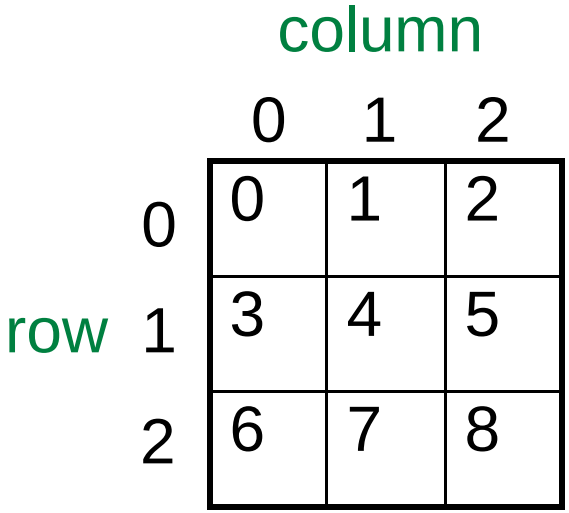



```
1  // Deleting data
2  int r;
3  for (r = 0; r < 3; r++) {
4      delete[] arr[r];
5  }
6
7  delete[] arr;
```

Built-in 2D Arrays in C/C++

```
1   int arr[3][3];
2   int val  = 0;
3
4   // For  each  row
5   for (int  r  =  0;  r  <  3;  r++)  {
6      // For  each  column
7      for (int  c  =  0;  c  <  3;  c++)  {
8         arr[r][c]  =  val++;
9      }
10  }
```

```
int a[3][3] = {{1,2,3},
               {4,5,6},
               {7,8,9}};
```



- No pointers used - safer code
- Size of 2D array set on initialization
- Uses less memory than pointer version
- g++ extension: can use variables as size declarator

Which Approach is Better?

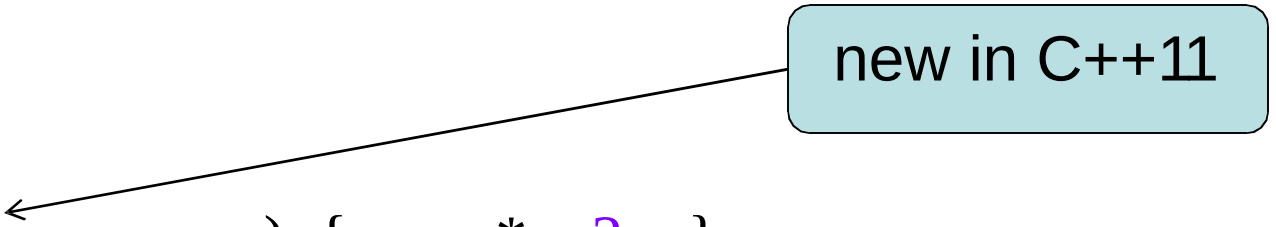
- "`int arr[N];`" **allocates memory on the program stack**, not on the heap
  - Goes out of scope automatically
  - No need to deallocate memory
  - Does not work for large *N*, may crash
- Avoid fixed-length buffers for variable-length input – this is a source of 70% of security breaches
- Dynamic memory allocation (`new`) uses the heap, but requires matching deallocation (`delete`)
  - Source of crashes, memory leaks, etc.
- C++11 STL offers cleaner solutions

Which Approach is Better?

- Built-in 2D arrays: not easy to dynamically allocate and pass as arguments to functions
- Double-pointer arrays (Iliffe vectors)
  - Support triangular arrays
  - Must be deallocated; are a little messy
  - C++11 offers a clean wrapper (vector) to simplify use
  - Allow copying, swapping rows quickly
  - `arr[i][j]` is slower than with arithmetic arrays
- Arithmetic arrays
  - Fast and not too messy
  - `arr[i][j]` uses one memory operation, not two

A Glimpse of C++11: Range-based For-loops

```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
```



- `for (int & x : my_array) { x *= 2; }`
- `for (int i = 0; i < 5; i++) { my_array[i] *= 2; }`
- `for (int i = 5; i > 0; i--) { my_array[i - 1] *= 2; }`
- `for (int i = 5; i-- > 0;) { my_array[i - 1] *= 2; }`

These are all equivalent

Strings as Arrays or Objects

C array of chars

```
1 char  s1[15] = "Hello ";
2 char  s2[] = "World";
3 strcat(s1, s2);
4 printf("%s\n", s1);
```

Use either to represent strings in C

```
1 char  *x = "Hello ";
2 char  *y = "World";
3 strcat(x, y);
4 cout  << x << endl;
5 printf("%s\n", x);
```

C++ string object

```
1 string x = "Hello ";
2 string y = "World";
3 string z = x + y;
4
5 cout  << z << endl;
6 cout << z.length() << endl;
7 printf("%s\n", z.c_str());
```

Use either to print

What would x+y do here?
Is it the same as with string x and string y?

- C++ strings are safer than C strings
- C++ strings keep track of their own length
- Extensive string libraries in stdlibc and stdlibc++ (knowing the functions is very useful)

NULL-termination Errors

```
1   char  x[10];
2   strcpy(x,  "0123456789”);
3
4   // allocate  memory
5   char  *y  =  (char*)malloc(strlen(x));
6   int  i;
7   for (i  =  1;  i <  11;  i++)  {
8      y[i]  =  x[i];
9   }
10  y[i]  =  '\0';
11  printf("%s\n",  y);
```

Copies 11 chars into space for 10, last char is a terminating null char

length too small since strlen() returns 10 instead of 11

Index should start at 0, not 1

Places null termination (’\0’) two indices too far

- Correct programs always run correctly on correct input
- **Buggy programs sometimes run correctly on correct input**
  - Sometime they crash even when input doesn’t change!

Off-by-One Errors

```
1   const int size = 5;
2   int x[size];


4   // set values to 0-4
5   for(int j = 0; j <= size; j++) {
6      x[j] = j;
7   }
8   // copy values from above
9   for (int k = 0; k <= size - 1; k++) {
10     x[k] = x[k + 1];
11  }
12  // set values to 1-5
13  for (int m = 1; m < size; m++) {
14     x[m - 1] = m;
15  }
```

Attempts to access x[5].
Should use j < size

Attempts to copy the contents of x[5] into x[4].
Should use k < (size - 1)

Does not set value of m[4].
Should use m <= size

Strings as Arrays Example

```
1   int main(int argc, const char* argv[]) {
2       char name[20];
3       strcpy(name, argv[1]);
4   }
```

What errors may occur when running the code?

How can the code be made safer?

```
1   int main(int argc, const char* argv[]) {
2       char* name = nullptr;
3       if (argc> 1) {
4           name = strdup(argv[1]);
5           if (*name == ENOMEM) // ENOMEM is from errno.h
6               cerr << ”Unable to get memory." << endl;
7       }
8       // strdupis from cstdlib, not C++
9       // use free() with it, not delete[]
10      free(name);
11  }
```

new in C++11

Container Classes

- Objects that contain multiple data items, e.g., ints, doubles or objects
- Allow for control/protection over editing of objects
- Can copy/edit/sort/order many objects at once
- Used in creating more complex data structures
  - Containers within containers
  - Useful in searching through data
  - Databases can be viewed as fancy containers
- Examples: array, list, stack, queue, map
- STL (Standard Template Library)

Most Data Structures in EECS 281 are Containers

- Ordered and sorted ranges
- Heaps, hash tables, trees & graphs,…
- Today: array-based containers as an illustration

Container Class Operations

- Constructor
- Destructor
- Add an Element
- Remove an Element
- Get an Element
- Get the Size
- Copy
- Assign an Element

What other operations may be useful?

Creating Objects & Dynamic Arrays in C++

- new calls default constructor to create an object
- new[] calls default constructor for each object in an array
  - No constructor calls when dealing with basic types (int, double)
  - No initialization either

- delete invokes destructor to dispose of the object
- delete[] invokes destructor on each object in an array
  - No destructor calls when dealing with basic types (int, double)

- Use delete on memory allocated with new
- Use delete[] on memory allocated with new[]

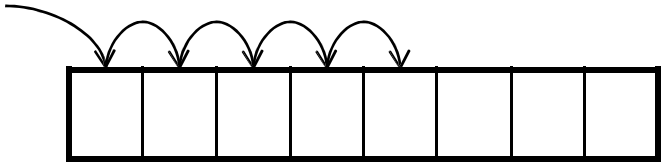
C Arrays versus C++ Arrays

- Function malloc() vs. operators new[] and new
- Function free() vs. delete[] and delete
- Not recommended to mix C calls with C++ calls
  - In C, use *malloc* and *free*
  - In C++, use new, new[], delete and delete[]
  - Standard C functions call *malloc()* and *free(): strdup()*, etc., while C++ functions call new[] and delete[]

```
1    int  *array = new int[4];
2    string  *pStr = new string("Hello");
3
4    delete[] array;
5    delete pStr;
```

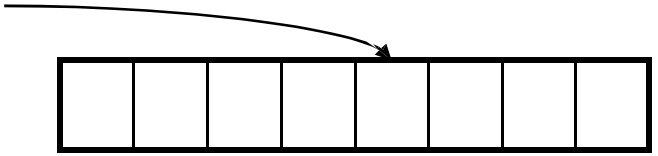
Sequential versus Random Access

Sequential



- Finds n^{th} item by starting at beginning
- Used by arrays to copy, compare two arrays, etc
- Used by disks in computers (slow)

Random



- Finds n^{th} item by going directly to n^{th} item
- Used by arrays to access data
- Used by main memory in computers (fast)

What are the advantages and disadvantages of each?

Copying with Pointers

How can we copy data from src_ar to dest_ar?

```
1  const int  size = 4;
2  double  src_ar[ ] = {3, 5, 6, 1};
3  double  dest_ar[size];
```

No Pointers

```
1  for(int i = 0; i < size; i++) {
2     dest_ar[i] = src_ar[i];
3  }
```

Pointer++

```
1  double  *sptr = src_ar;
2  double  *dptr = dest_ar;
3
4  while(sptr != src_ar + size)
5     *dptr++  = *sptr++;
```

Why would you use pointers when the code seems simpler without them?

What to Store in a Container (Data Type)

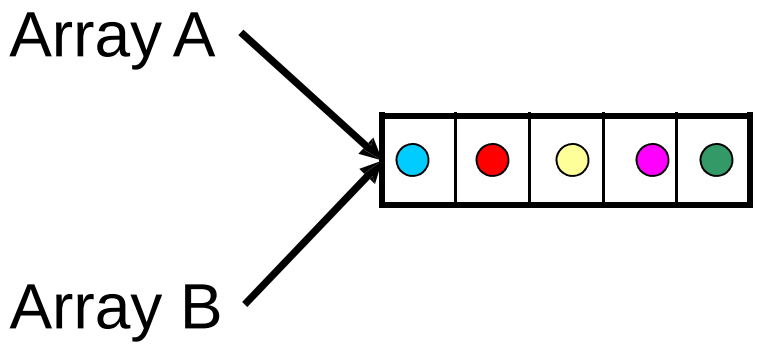
	Value	**Pointer**	**Reference**
Example	`double data;`	`double *data;`	`double &data(d);`
Data ownership	Only container edits/deletes	Container or other object	None: cannot delete by reference
Drawbacks	Large objects take time to copy	Unsafe	Must be initialized but cannot be assigned to
Usage	Most common	Used for char*, shared data	Impractical in most cases

What to Get from a Container (Return Type)

	Value	**Ptr, Const ptr**	**Reference, const ref**
Example	`int getElt(int);`	`int* getElt(int);`	`int& getElt(int);`
Notes	Costly for copying large objects	Unsafe, pointer may be invalid	Usually a good choice

Memory Ownership: Motivation

Both arrays contain pointers to objects



What happens when we delete Array A?

Memory Management

- Options for copy-construction and assignment
  - Duplicate objects are created
  - Duplicate pointers to objects are created
    - Multiple containers will point to same objects
- Default copy-constructor duplicates pointers
  - Is this desirable?

- Idea 1: Each object owned by a single container
- Idea 2: Use no ownership
  - Objects expire when no longer needed
    - Program must be watched by a “big brother”
    - Garbage collector - potential performance overhead
  - Automatic garbage collection in Java
    - Can be done in C++ with additional libraries or “smart pointers”

Memory Ownership: Pointers

- Objects could be owned by another container
  - Container may not allow access to objects (privacy,safety)
  - Trying to delete same chunk of memory twice may crash the program
- Destructor may need to delete each object
  - Inability to delete objects could cause memory leak

Safety Tip (Defensive Programming)

Use

```
delete ptr;
ptr = nullptr;
```

instead of

```
delete ptr;
```

Note that `delete nullptr;` does nothing

What's Wrong With Memory Leaks?

- When your program finishes, all memory should be deallocated
  - The remaining memory is "leaked"
  - C++ runtime may or may not complain
  - The OS will deallocate the memory
- Your code should be reusable in a larger system
  - If your code is called 100 times and leaks memory, it will exhaust all available memory and crash
  - The autograder limits program memory and is very sensitive to memory leaks
- Use: `$ valgrind –tool=memcheck ./cmd ...`

Example of a Container Class: Adding Bounds Checking to Arrays

```
1   class Array {
2       double* data;                 // Array data
3       unsigned int length;    // Array size
4   public:
5   // Why aren't data and length public?
6
7       Array(unsigned len = 0) : length(len) {
8           data = (len ? new double[len] : nullptr);
9       }
10      // methods to follow in next slides…
11  };
```

A class or a struct ?

```
1   struct Array{
2       double* data;
3       unsigned int length;
4       // insertmethods here
5   };
```

Array Class: Copy Constructor

```
1   Array(const Array &a) { // deep copy
2       length = a.getLength();
3       data = new double[length];
4       for (unsigned i = 0; i < length; i++)
5           data[i] = a[i];
6   }
```

The constructor allows the following usage:

```
1   Array a;
2   Array b(20, -1);   // Array a is of length 20
3   Array c(b);          // copy constructor
4   Array d = b;         // also copy constructor
5   a = c;             // operator=, better overloadtoo!
6   // what if we used shallow copy instead of deep copy ?
```

Array Class: Better Copying

```
1   void copyFrom(const Array &a) {   // Deep copy
2     if (length != a.length) {       // Resize array
3       delete[] data;                // deleting nullptr is OK
4       length = a.length;
5       data = new double[length];
6     }
7     // Copy array
8     for (unsigned int i = 0; i < length; i++) {
9       data[i] = a.data[i];
10    }
11  }
12
13  Array(const Array &a) : length(0), data(nullptr) {
14    copyFrom(a);
15  }
16
17  Array &operator=(const Array &a) {
18    if (this == &a) return *this;     // Idiotcheck
19    copyFrom(a);
20    return *this;
21  }
```

Array Class: Best Copying

```
1  #include <utility>  // Access  to  swap
2
3  Array(const  Array  &a)  : length(a.length),
4          data(new double[length])  {
5      // Copy  array  contents
6      for  (unsigned int  i =  0; i  <  length;  i++)  {
7          data[i]  =  a.data[i];
8  }
9
10 Array  &operator=(const  Array  &a)  {  // Copy-swap  method
11     Array  temp(a);  // Destroyed  when function  ends...
12     swap(temp);  // Swap current  object's  data  with  temp's data
13     return *this;
14 }
15
16 void  swap(Array  &other)  {
17     std::swap(data,  other.data);
18     std::swap(length,  other.length);
19 }
```

The Big 5 to Implement

- You already know that if your class contains dynamic memory as data, you should have:
  - Destructor
  - Copy Constructor
  - Overloaded operator=()
- C++ 11 provides optimizations, 2 more:
  - Copy Constructor from r-value
  - Overloaded operator=() from r-value

Array Class: Complexity of Copying

```
1 Array(const Array &a) {
2    length = a.getLength();
3    data = new double[length];
4    for (unsigned i = 0; i < length; i++) {
5       data[i] = a[i];
6    }
7 }
```

← 1 step
← 1 step
← n times
← c steps

Total: 1 + 1 + (*n* * *c*) + 1 = *O(n)*

Best Case: *O(n)*
Worst Case: *O(n)*
Average Case: *O(n)*

Why is Best == Worst == Average?

Array Class: Destructor

Assume data are pointers *owned* by the class Array:

```
1   ~Array() {
2      if (data!= nullptr) {
3         for (inti = 0; i < length; i++) {
4            delete data[i];
5         }
6         delete[] data;
7         data = nullptr;
8      }
9   }
```

← n times
← 1 step
(destructor is *O(1)* time)
← 1 step
← 1 step
Total:
(*n* * 1) + 1 + 1 = *O(n)*

What if data are not pointers?

Assume data are doubles:

```
1   ~Array() {
2      if (data) {
3         delete[] data;   // data are doubles
4         data = nullptr;
5      }
6   }
```

double has no destructor

← 1 step
← 1 step

Total: 1 + 1 = *O*(1)

Array Class: operator[]

Overloading: Defining two operators/functions of same name

```
//--- non-const version
double &operator[](int idx) {
  if (idx < length && idx >= 0)
    return data[idx];
  throw runtime_error("bad idx");
}
```

```
//--- const version
const double &operator[](int idx) const {
  if (idx < length && idx >= 0)
    return data[idx];
  throw runtime_error("bad idx");
}
```

Why do we need two versions?

Which version is used in each instance below?

```
1  Array a(3);
2  a[0] = 2.0;
3  a[1] = 3.3;
4  a[2] = a[0] + a[1];
```

Array Class: const operator[]

```
//--- const version for basic types
const double &operator[](int idx) const {
   // Return by reference
}
```

```
//--- const version for objects
const Type &operator[](int idx) const {
      // Return by const reference
}
```

- Declares read-only access (compiler enforced)
- Automatically selected by the compiler when an array being accessed is const
- Helps compiler optimize code for speed

```
1   //--- Prints array
2   ostream &operator<<(ostream &os, const Array &a) {
3      for(int i = 0; i < a.getLength(); i++) {
4         os << a[i] << "  ";
5      }
6      return os;
7   }
```

const version of operators are needed to access const data

Array Class: 2D+ Case

```
//--- const version for basic types
const double &operator()(int i,int j) const {
   // Return by const reference
}
```

```
//--- const version for objects
const Type &operator()(int i, int j) const {
   // Return by const reference
}
```

- Replace operator[] with operator()
- Everything else stays the same
- Make a non-const version also (just remove both const keywords)
- The return statements are identical in all four cases (no &, no const)

Array Class: Inserting an Element

```
1   bool  insert(int index, double val) {
2      if (index >= length ||index < 0)
3         return false;
4      for (int i = length- 1; i > index; --i)
5         data[i] = data[i - 1];
6      data[index] = val;
7      return true;
8   }
```

Why decrement *i*?
Why not increment?

---

ar

1.6	3.1	4.2	5.9		

Original array

ar.insert(1, 3.4);

Call insert

ar

1.6	3.1	3.1	4.2	5.9	

Shift data over

ar

1.6	3.4	3.1	4.2	5.9	

Insert new data

Are arrays desirable when many insertions are needed?

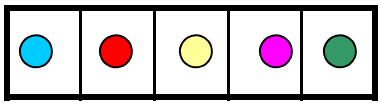
Array Class: Complexity of Insertion

```
1   bool insert(int index, double val) {
2      if (index >= size || index < 0)
3         return false;
4      for (int i = size - 1; i > index; --i)
5         data[i] = data[i - 1];
6      data[index] = val;
7      return true;
8   }
```

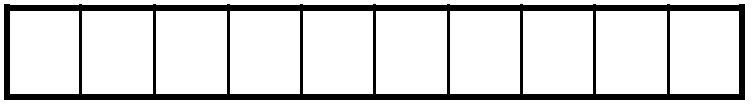
← At most *n* times

- Best Case: *O(1)*
  - Inserting after existing data
  - No data shifted
- Worst Case: *O(n)*
  - Inserting before all existing data
  - All data shifted
- Average Case: *O(n)*
  - Why is average case the same as worst case?

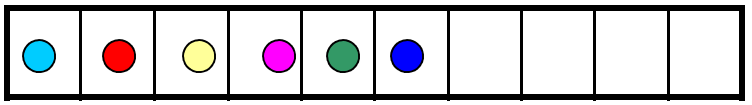
Array Class: Append Example

Original array = 

How can we append one more element?

Create a new array = 

Copy existing elements into new array and add new element

New array = 

Delete old array so that memory can be reused

Why do we have to make a new array?

Why is the new array twice as big as the old array?

Array Class: Complexity of Append

Appending *n* elements to a full array

- When array is full, resize
  - **Double** array size from *n* to 2*n* (1 step)
  - Copy *n* items from original array to new array (*n* steps)

- Appending *n* elements after array is resized
  - Place element in appropriate location (1 step * *n*)
- Total: $1 + n + n = 2n + 1$ steps
- Amortized: $(2n + 1)/n = 2 + 1/n$ steps/append = $O(1)$

10 Study Questions

1. What is memory ownership for a container?
2. What are some disadvantages of arrays?
3. Why do you need a const and a non-const version of some operators? What should a non-const op[ ] return?
4. How many destructor calls (min, max) can be invoked by:
   `operator delete` and `operator delete[]`
5. Why would you use a pointer-based copying algorithm ?
6. Are C++ strings null-terminated?
7. Give two examples of off-by-one bugs.
8. How do I set up a two-dim array class?
9. Perform an amortized complexity analysis of an automatically-resizable container with doubling policy.
10. Discuss pros and cons of pointers and references when implementing container classes.