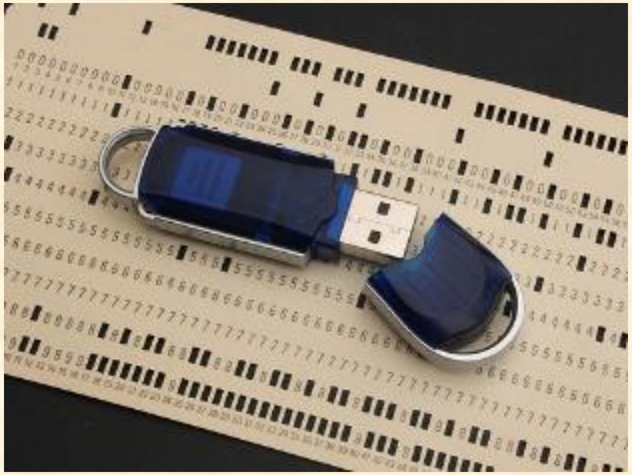
Lecture 5
Arrays & Container Classes



EECS 281: Data Structures & Algorithms

Job Interview Questions

- Assume that a given array has a majority (>50%) element – find it with constraints:

  Linear time using O(1) memory

  11 13 99 12 99 10 99 99 99

- Same for an array that has an element repeating more than n/3 times

  11 11 99 10 99 10 12 19 99

Know your Arrays!

- What does this code do? Is line 5 a compiler error, runtime error, or works?

```
1   double a[] = {1.1, 2.2, 3.3};
2   int i = 1;
3
4   cout << a[i] << endl;
5   cout << i[a] << endl;
```

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++1z `vector<T>`
  - Or `array<T, SIZE>`, but it's not as useful (cannot grow)
- Avoid pointers (where possible)
  - Use STL containers, function objects, integer indices, iterators
- Use C++ `string` objects

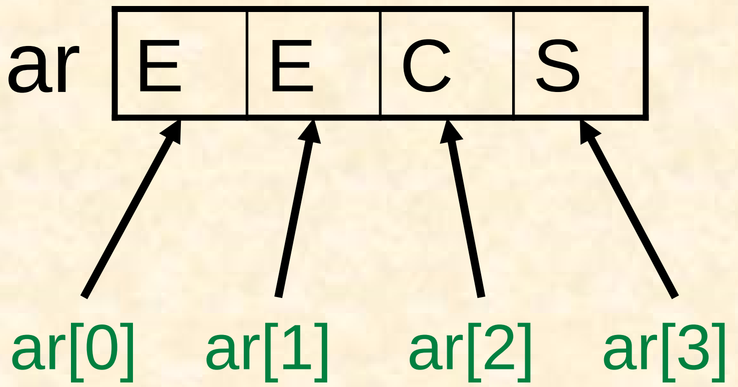
Review: Arrays in C/C++

```c
char ar[] = {'A', 'E', 'C', 'S'};
ar[0] = 'E';

char c = ar[2];
// now we have c=='C'

char *ptr = ar;
// now ptr points to EECS

ptr = &ar[1];
// 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**

Fixed Size Arrays: 1D and 2D

1D array

```
int a1D[9];
```

0	1	2	3	4	5	6	7	8

column = index % num_columns
row = index / num_columns

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

3x3 2D array

```
int a2D[3][3];
```

column

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

index = row * num_columns + column

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

Fixed size 2D Arrays in C/C++

```
const int ROWS = 3, COLS = 3;
int arr[ROWS][COLS];
int val = 0;

// For each row
for (int r = 0; r < ROWS; ++r)
  // For each column
  for (int c = 0; c < COLS; ++c)
    arr[r][c] = val++;
```

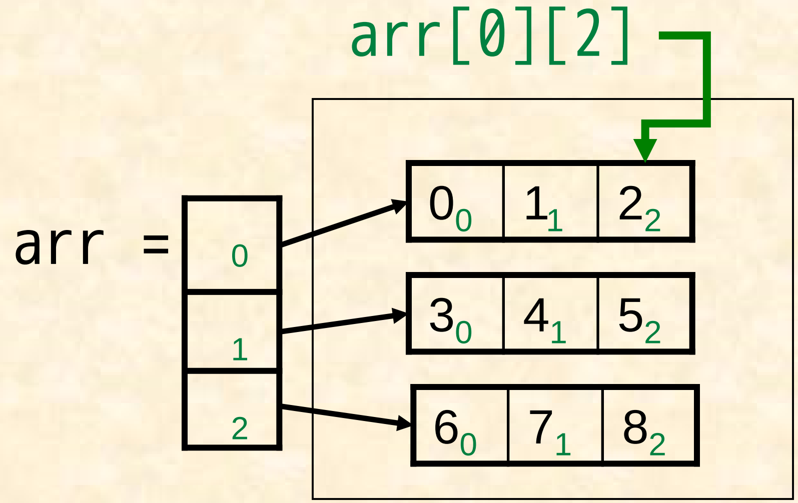
```
int a[3][3] = {{1,2,3},
               {4,5,6},
               {7,8,9}};
```

column

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

- 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

2D Arrays with Double Pointers




```
// Create array of rows
int rows, cols;
cin >> rows >> cols;
int **arr = new int * [rows];

// For each row, create columns
for (int r = 0; r < rows; ++r)
  arr[r] = new int[cols];

int val = 0;
// For each row
for (int r = 0; r < rows; ++r)
  // For each col
  for (int c = 0; c < cols; ++c)
    arr[r][c] = val++;
```

```

// Deleting data
int r;
for (r = 0; r < rows; ++r)
  delete[] arr[r];

delete[] arr;
```

Pros and Cons: Fixed

Allocated on the stack (not the heap)

- Pros:
  - Deallocated when it goes out of scope (no **`delete`**)
  - `arr[i][j]` uses one memory operation, not two
- Cons:
  - Does not work for large *N*, may crash
  - Size fixed at compile time
  - Difficult to pass as arguments to functions
- Avoid fixed-length buffers for variable-length input – This is a source of 70% of security breaches

Pros and Cons: Dynamic

Double-pointer arrays are allocated on the heap

- Pros:
  - Support triangular arrays
  - Allow copying, swapping rows quickly
  - Size can be changed at runtime
- Cons:
  - Requires matching `delete`; can crash, leak memory
  - `arr[i][j]` is slower than with built-in arrays
- C++ STL offers cleaner solutions such as **`vector`**

Off-by-One Errors

```
const int SIZE = 5;
int x[SIZE];

// set values to 0-4
for (int j = 0; j <= SIZE; ++j) {
  x[j] = j;
} // for
// copy values from above
for (int k = 0; k <= SIZE - 1; ++k) {
  x[k] = x[k + 1];
} // for
// set values to 1-5
for (int m = 1; m < SIZE; ++m) {
  x[m - 1] = m;
} // for
```

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`

Range-based for-loops

(since C++11)

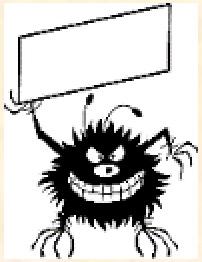
```
int my_array[5] = {1, 2, 3, 4, 5};
// two ways to double the value of each element in my_array:
// Classic for-loop
for (int i = 0; i < 5; ++i)
  my_array[i] *= 2;

// Range-based for-loop >= C++11
for (int &x : my_array)
  x *= 2;
```

- Notice the reference parameter, *x*
- Range-based loops either by value or reference

Strings as Arrays Example

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



What errors may occur when running the code?

How can the code be made safer?

```
int main(int argc, const char *argv[]) {
  string name;

  if (argc > 1)
    // string has a convert-assignment from char *
    name = argv[1];

  // When main() ends, string destructor runs automatically
  return 0;
} // main()
```

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[]`

Container Classes

- Objects that contain multiple data items, e.g., `int`s, `double`s 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: vector, list, stack, queue, deque, 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

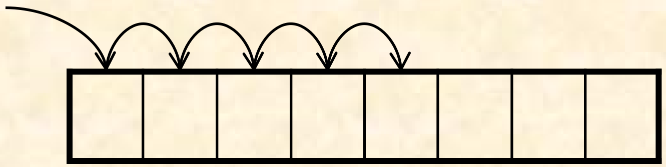
Container Class Operations

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

What other operations may be useful?

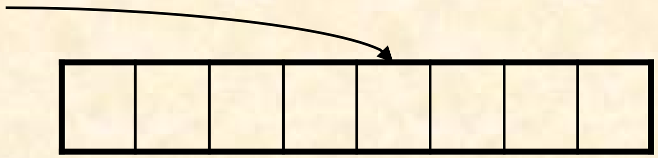
Accessing Container Items

Sequential



- Finds n^{th} item by starting at beginning
  - Example: linked list
- Used by disks in computers (slow)

Random Access



- Finds n^{th} item by going directly to n^{th} item
- Used by arrays to access data
- Used by main memory in computers (fast)
- Arrays can still proceed sequentially to copy, compare contents, etc.

What are the advantages and disadvantages of each?

Copying with Pointers

How can we copy data from src_ar to dest_ar?

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

No Pointers

```
for (int i = 0; i < SIZE; ++i) {
  dest_ar[i] = src_ar[i];
} // for
```

Pointer++

```
double *sptr = src_ar;
double *dptr = dest_ar;

while(sptr != src_ar + SIZE)
  *dptr++ = *sptr++;
```

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

What to Store in a Container (Data Type)

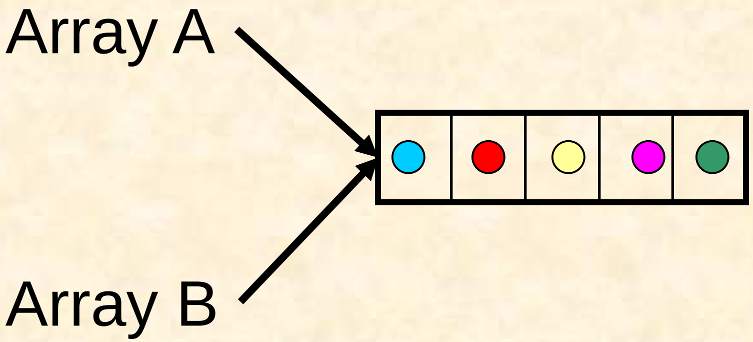
	Value	**Pointer**	**Reference**
Example	`char data;`	`char *data;`	`char &data(c);`
Data ownership	Only container edits/deletes	Container or other object	None: cannot delete by reference
Drawbacks	Large objects take time to copy	Unsafe	Must be initialized but cannot be assigned to
Usage	Most common	Used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

What to Get from a Container (Return Type)

	Value	**Ptr, Const ptr**	**Reference, const ref**
Ex.	`char getElt(int);`	`char *getElt(int);`	`char &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 to A when we modify B?
What happens when we delete Array A?
What happens when we later delete Array B?

Memory Ownership: Issues

- 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 ./program ...`

Example of a Container Class: Adding Bounds Checking to Arrays

```
class Array {
  double *data;           // Array data
  unsigned int length;  // Array size

public:
  Array(unsigned len = 0) : length{len} {
    data = (len ? new double[len] : nullptr);
  } // Array()
  unsigned int size() const { return length; }
  // other methods to follow in next slides...
}; // Array
```

A class or a struct ?

```
struct Array {
  double *data;
  unsigned int length;
  // insert methods here
}; // Array
```

Array Class: Destructor

Assume data are doubles:

```
~Array() {
  delete[] data;  // data are doubles
  data = nullptr;
} // ~Array()
```

double has no destructor

1 step

1 step

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

What if we have a 2D dynamic memory array?

Assume data are pointers owned by the class Array:

```
 ~Array2D() {
   if (data != nullptr) {
     for (unsigned i = 0; i < length; ++i)
       delete[] data[i];

     delete[] data;
     data = nullptr;
   } // if
 } // ~Array2D()
```

n times

1 step

(destructor is *O(1)* time)

1 step

1 step

Total: $n + 1 + 1 = O(n)$

Array Class: Copy Constructor

```
// deep copy
Array(const Array &a) : data{new double[length]},
                        length{a.length} {
  for (unsigned i = 0; i < length; ++i)
    data[i] = a[i];
} // Array()
```

The class allows the following usage:

```
Array a;           // Array a is of length 0
Array b(20);       // Array b is of length 20
Array c(b);        // copy constructor
Array d = b;       // also copy constructor
a = c;             // needs operator=, shallow copy only!
                   // how do we do a deep copy?
```

Array Class: Better Copying

```
void copyFrom(const Array &a) {  // deep copy
  delete[] data;                 // deleting nullptr is OK
  length = a.length;
  data = new double[length];

  // Copy array
  for (unsigned int i = 0; i < length; ++i)
    data[i] = a.data[i];
} // copyFrom()

Array(const Array &a) : data{nullptr}, length{0} {
  copyFrom(a);
} // Array()

Array &operator=(const Array &a) {
  if (this == &a)        // idiot check
    return *this;        // idiot check
  copyFrom(a);
  return *this;
} // operator=()
```

Array Class: Best Copying

```
#include <utility> // Access to swap

Array(const Array &a) : data{new double[a.length]},
                        length{a.length} {
  // copy array contents
  for (unsigned int i = 0; i < length; ++i)
    data[i] = a.data[i];
} // Array()

void swap(Array &other) {
  std::swap(data, other.data);
  std::swap(length, other.length);
} // swap()

Array &operator=(const Array &a) { // Copy-swap method
  Array temp(a); // destroyed when function ends...
  swap(temp);    // swap this object's data with temp's data
  return *this;
} // operator=()
```

The Big ~~3~~ 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

```
Array(const Array &a) : data{new double[length]},
                        length{a.length} {
  for (unsigned i = 0; i < length; ++i)
    data[i] = a[i];
} // Array()
```

1 step
1 step
n times
c steps

Total: $1 + 1 + 1 + (n * (2 + c)) + 1 = O(n)$

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

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");
} // operator[]()
```

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

Why do we need two versions?

Which version is used in each instance below?

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

Array Class: const operator[]

- Declares read-only access
  - Compiler enforced
  - Returned references don't allow modification
- Automatically selected by the compiler when an array being accessed is const
- Helps compiler optimize code for speed

```
//--- Prints array
ostream &operator<<(ostream &os, const Array &a) {
  for (unsigned int i = 0; i < a.size(); ++i)
    os << a[i] << " ";

  return os;
} // operator<<()
```

const version of operators are needed to access const data

Array Class: 2D+ Case

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

- Replace `operator[]` with `operator()`
  - Because `operator[]` can only have 1 parameter
- Everything else stays the same
- Make a non-const version also (just remove both `const` keywords)
- The return statements are identical in every version (no `&`, no `const`)

Array Class: Inserting an Element

```
bool insert(int index, double val) {
  if (index >= length || index < 0)
    return false;
  for (unsigned i = length - 1; i > index; --i)
    data[i] = data[i - 1];
  data[index] = val;
  return true;
} // insert()
```

Why decrement *i*?

Why not increment?

---

ar	1.6	3.1	4.2	5.9	7.3	8.4	Original array

`ar.insert(2, 3.4);` Call insert

ar	1.6	3.1	4.2	4.2	5.9	7.3	Copy data (losing 8.4)
ar	1.6	3.1	3.4	4.2	5.9	7.3	Overwrite old with new

Are arrays desirable when many insertions are needed?

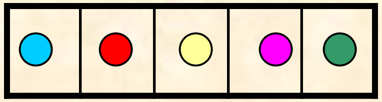
Array Class: Complexity of Insertion

```
bool insert(int index, double val) {
  if (index >= length || index < 0)
    return false;
  for (unsigned i = length - 1; i > index; --i)
    data[i] = data[i - 1];
  data[index] = val;
  return true;
} // insert()
```

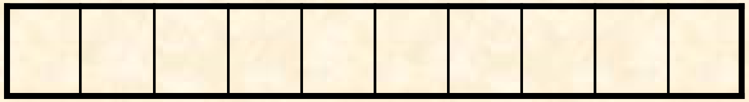
At most *n* times

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

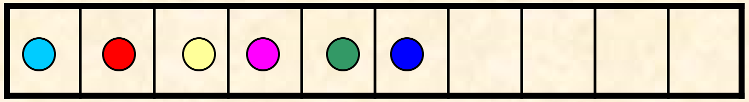
Array Class: Append Example

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
  - **<u>Double</u>** array size from n to $2n$ (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.