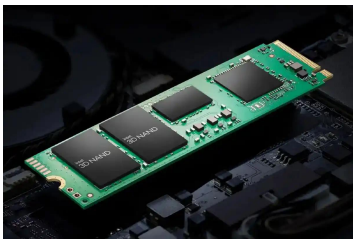
Arrays & Container Classes



Data Structures & Algorithms

Understanding and Using

- 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++ 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

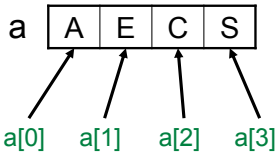
Know your Arrays!

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

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


Review: Arrays in C/C++

```
1   char a[] {'A', 'E', 'C', 'S'};
2   a[0] = 'E';
3
4   char c = a[2];
5   // now c contains 'C'
6
7   char *p = a;
8   // now p points to 1st E in a
9
10  p = &a[1];
11  // now p points to 2nd E in a
```



same as p = a + 1;
or p++;

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

Fixed Size Arrays: 1D and 2D

1D array
int a1D[9];

index

0	1	2	3	4	5	6	7	8
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];

row \ column	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++

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

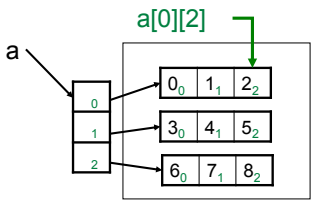
```
11  // static initialization
12  int a[3][3] { {0,1,2},
13                {3,4,5},
14                {6,7,8} };
```

row \ column	0	1	2
0	0	1	2
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

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




```
16  // Deleting data
17  for (r = 0; r < rows; ++r)
18    delete[] a[r];
19
20  delete[] a;
```


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
  - a[i][j] is slower than with built-in arrays
- C++ STL offers cleaner solutions such as vector<>

Off-by-One Errors

```
1  const size_t SIZE = 5;
2  int x[SIZE];
3  size_t i;
4
5  // set values to 0-4
6  for (i = 0; i <= SIZE; ++i)
7    x[i] = i;
8
9  // copy values from above
10 for (i = 0; i <= SIZE - 1; ++i)
11   x[i] = x[i + 1];
12
13 // set values to 1-5
14 for (i = 1; i < SIZE; ++i)
15   x[i - 1] = i;
```

Attempts to access x[5]. Should use i < SIZE

Copies x[5] into x[4]. Should use i < (SIZE - 1)

Does not set value of x[4]. Should use i <= SIZE

Range-based for-loops

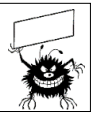
(C++11+)

```
1  // two ways to double the value of each element in my_array:
2  int my_array[5] {1, 2, 3, 4, 5};
3
4  // Classic for-loop
5  for (size_t i = 0; i < 5; ++i)
6    my_array[i] *= 2;
7
8  // Range-based for-loop C++11+
9  for (int &item : my_array)
10   item *= 2;
```

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

Strings as Arrays Example

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



What errors may occur when running the code?
How can the code be made safer?

```
5  int main(int argc, char *argv[]) {
6    string name;
7
8    if (argc > 1)
9      // string has a convert-assignment from char *
10     name = argv[1];
11
12   // When main() ends, string destructor runs automatically
13   return 0;
14 } // main()
```

Job Interview Question

- Assume that a given array has a majority (>50%) element – find it with constraints:
  - **Use *O*(*n*) time and *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
  - **Use average *O*(*n*) time and *O*(k) memory**

  11 11 99 10 99 10 12 19 99

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: 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

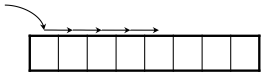
Container Class Operations

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

What other operations may be useful?

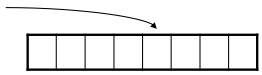
Accessing Container Items

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 an Array

```
1  const size_t SIZE = 4;
2  double src_ar[] {3, 5, 6, 1};
3  double dest_ar[SIZE];
4
5  for (size_t i = 0; i < SIZE; ++i)
6    dest_ar[i] = src_ar[i];
7
8  double *sptr = src_ar;
9  double *dptr = dest_ar;
10
11 while (sptr != src_ar + SIZE)
12   *dptr++ = *sptr++;
```

How can we copy data from src_ar to dest_ar?

No Pointers

Pointer++

Why use pointers when the code looks simpler without them?

What to Store in a Container (Data Type)

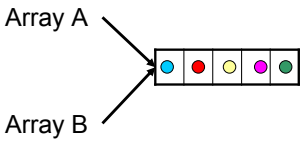
	Value	Pointer	Reference
Example	`char data;`	`char *data;`	`char &data(c);`
Data ownership	Only container edits/deletes	Container or other object	None: cannot delete by reference
Drawbacks	Large objects take time to copy	Unsafe	Must be initialized but cannot be assigned to
Usage	Most common	Used for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

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[]`

Example of a Container Class: Adding Bounds Checking to Arrays

```
1  class Array {
2    size_t length = 0;       // Array size
3    double *data = nullptr;  // Array data
4
5  public:
6    Array(size_t len) : length{len},
7                        data{new double[length]} {}
8    ~Array() {
9      delete[] data;
10     data = nullptr;
11   } // ~Array()
12
13   size_t size() const {
14     return length;
15   } // size()
16 }; // Other methods to follow...
```

What to use, `class` or `struct`?

Array Class: Copy Constructor

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

The class allows the following usage:

```
7  Array a(10);      // Array a is of length 10
8  Array b(20);      // Array b is of length 20
9  Array c(b);       // copy constructor
10 Array d = b;      // also copy constructor
11 a = c;            // needs operator=, shallow copy only!
                     // how do we do a deep copy?
```

Array Class: Complexity of Copying

```
1  // deep copy
2  Array(const Array &a) : length{a.length},          ← 1 step
3                          data{new double[length]} { ← 1 step
4    for (size_t i = 0; i < length; ++i)              ← n times
5      data[i] = a.data[i];                           ← c steps
6  } // Array()
```

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

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

Array Class: OK Copying

```
1  void copyFrom(const Array &other) {  // deep copy
2    delete[] data;  // safe to delete even if nullptr
3    length = other.length;
4    data = new double[length];
5
6    // Copy array
7    for (size_t i = 0; i < length; ++i)
8      data[i] = other.data[i];
9  } // copyFrom()
10
11 Array(const Array &other) {
12   copyFrom(other);
13 } // Array()
14
15 Array &operator=(const Array &other) {
16   if (this != &other) // idiot check
17     copyFrom(other);
18   return *this;
19 } // operator=()
```


Array Class: Best Copying

```
1  #include <utility>  // Access to std::swap
2
3  Array(const Array &other) : length{other.length},
4                              data{new double[length]} {
5    // copy array contents
6    for (size_t i = 0; i < length; ++i)
7      data[i] = other.data[i];
8  } // Array()
9
10 Array &operator=(const Array &other) {  // Copy-swap method
11   Array temp(other);  // use copy constructor to create object
12
13   // "assign" to this object by swapping with temp
14   std::swap(length, temp.length);
15   std::swap(data, temp.data);
16   return *this;  // delete original, return copied object
17 } // 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

Why Know About *r*-values?

- If you have the function shown below, and you're compiling with C++11 or later, the `vector` won't be copied on return:

```
1  vector<string> readData() {
2    vector<string> data;
3    string temp;
4    while (cin >> temp)
5      data.push_back(temp);
6    return temp;
7  } // readData()
```


Array Class: operator[]

Overloading: Defining two operators/functions of same name

```
// non-const version
double &operator[](size_t i) {
  if (i < length)
    return data[i];
  throw runtime_error("bad i");
} // operator[]()
```

```
// const version
const double &operator[](size_t i) const {
  if (i < length)
    return data[i];
  throw runtime_error("bad index");
} // operator[]()
```

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[]

- 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

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

const operators are needed to access const data

Array Class: Inserting an Element

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

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

At most *n - 1* 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?

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.