COP3503

Data Structures and Storage

Every Program Needs Storage

- In the real world, we have lots of storage options.
  - Boxes, bins, drawers, cabinets, shelves, etc.
  - Depending on the item, we may store it differently.
  - Storing large pieces of lumber? Large, open supports to hold a lot of weight is necessary
  - Storing large quantities of small items (nails, screws, LEGO bricks, etc.)? Lots of small drawers might be better

- In our programs, we use **data structures** to store information.
  - Like physical containers, they have ideal and not-so-ideal situations.
  - They can all have different functionality.
  - Some are more memory efficient; some are faster.

Arrays

- The simplest “data structure”, the first you learn in most languages
- Arrays are meant to hold an exact number of something. No more, no less.

```
// Making a game with exactly 4 players
Player players[4];
```

```
const int PLAYER_COUNT = 4;
Player players[PLAYER_COUNT];
```

- A simple box that is built with exactly the size that you specified. No bells, no whistles, no special features.
- In some languages, arrays are classes (which **do** have some bells and whistles).

Vectors

- More sophisticated, “dynamic” arrays. Vectors are good all-around storage.
- Need to hold exactly some number of something? Basic stuff, done!
- Need to expand that to hold 6 later on? They can resize!
- Need to shrink the container to only hold 3? They can resize!
- Need to know how big the container currently? They store count info!
- Have the Big Three implemented for easy copying and cleanup
- Easier to pass and return, good all-around replacements for arrays

If Vectors Exist, Why Would You Ever Use Arrays?

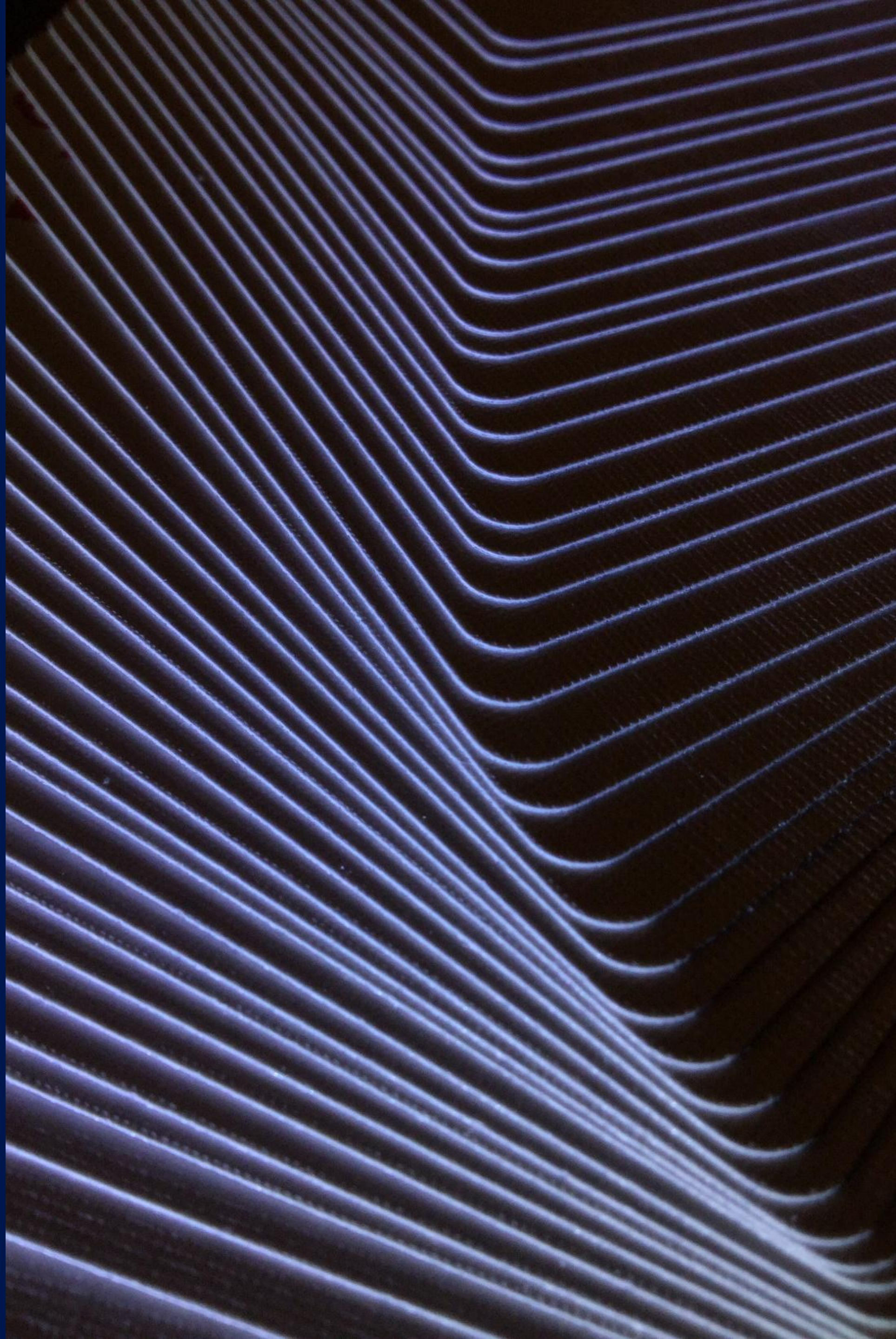
- If you know exactly the number you need, and that number won't change, use an array
- Ask yourself this question: Do you **need** any vector functionality?
  - Going to expand or remove elements?
  - Need some kind of complex searching/sorting functionality?
  - No? Then an array works just fine, though there's little "downside" to using a vector in simple ways

```
int someArray[10];
vector<int> someVector(10);

for (int i = 0; i < 10; i++)
{
    someArray[i] = i;
    someVector[i] = i;
}
```

Which of these is better or worse?
Neither: Just two ways to hold some stuff.

Stacks

- Last-In, First-Out (LIFO) storage, adding and retrieving follows a specific pattern.
- Drawing from the top of a deck of cards?
  A stack is a good fit.
- A common (and critical!) feature of many programs: **undo** and **redo**.
- You may have a “stack” of actions a program has taken to change something about the program.
- To undo you can “**pop**” the top action off a stack and reverse the changes it made.
- Want to redo that action? When you pop something off the undo stack, “**push**” it onto a redo stack.



Undo Stack Example: Typing in a Text Editor

Example: Typing the Word “Stack”

Editor Window

S t a c k

Each time we want to undo, we “pop” from the undo stack…

Undo stack
k
c
a
t
S

Redo stack
S
t
a
c
k

…and we “push” onto the redo stack.

There would be a lot more to making the rest of this program work!

The main thing to remember here is the ordering of a stack.

For many data structures, the order of data is exactly why you use it.

Queue

- First-In, First-Out (FIFO) storage
- The first item added is the first to be removed.
- Queues are good for simulating lines, orders, customers, anything “first come, first served”.
- Behind the scenes, very similar to a stack
- May be “just” a dynamic array, with some logic to determine order of access.

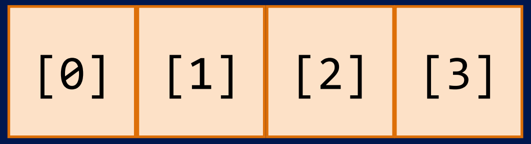
Linked Lists

- **Non-contiguous** data structure
- Uses **nodes** to store information, generic “units” of a data structure
  - Nodes are not unique to linked lists.
  - Nodes are typically simple classes or structures that may not “do” much.
  - Primarily for holding data, may contain little functionality.
- Nodes store data, but also connections (pointers) to other nodes.
  - Some may have a single connection, some kind of “next” node.
  - Others may have multiple connections (sometimes without limit!).
- Node-based data structures use more memory, but are often faster when adding and removing elements

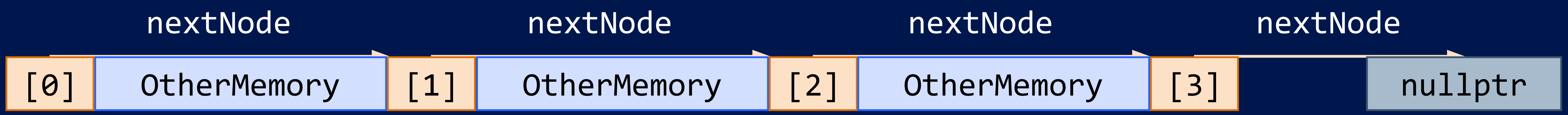
Contiguous Data vs. Non-Contiguous Data

- **Contiguous** data is adjacent in memory.
- Arrays are contiguous—they are a single “block” of memory that is connected, unbroken.

```
int someArray[4];
```



- **Non-contiguous** memory is not adjacent.
- Node-based data structures are non-contiguous (basically any data structure that isn’t array-based)
- Pointers are used to store the location of other elements.



Data Structure Operations

Not all data structures will support all of these operations!

Every data structure has common operations you might use:

Insertion / Addition
Adding some elements to a container

Deletion / Removal
Removing some elements from a container

Searching / Retrieval
Finding something from the container

Traversal
Accessing each element in the container for processing

"Go through the container and **do something** with each bit of data"

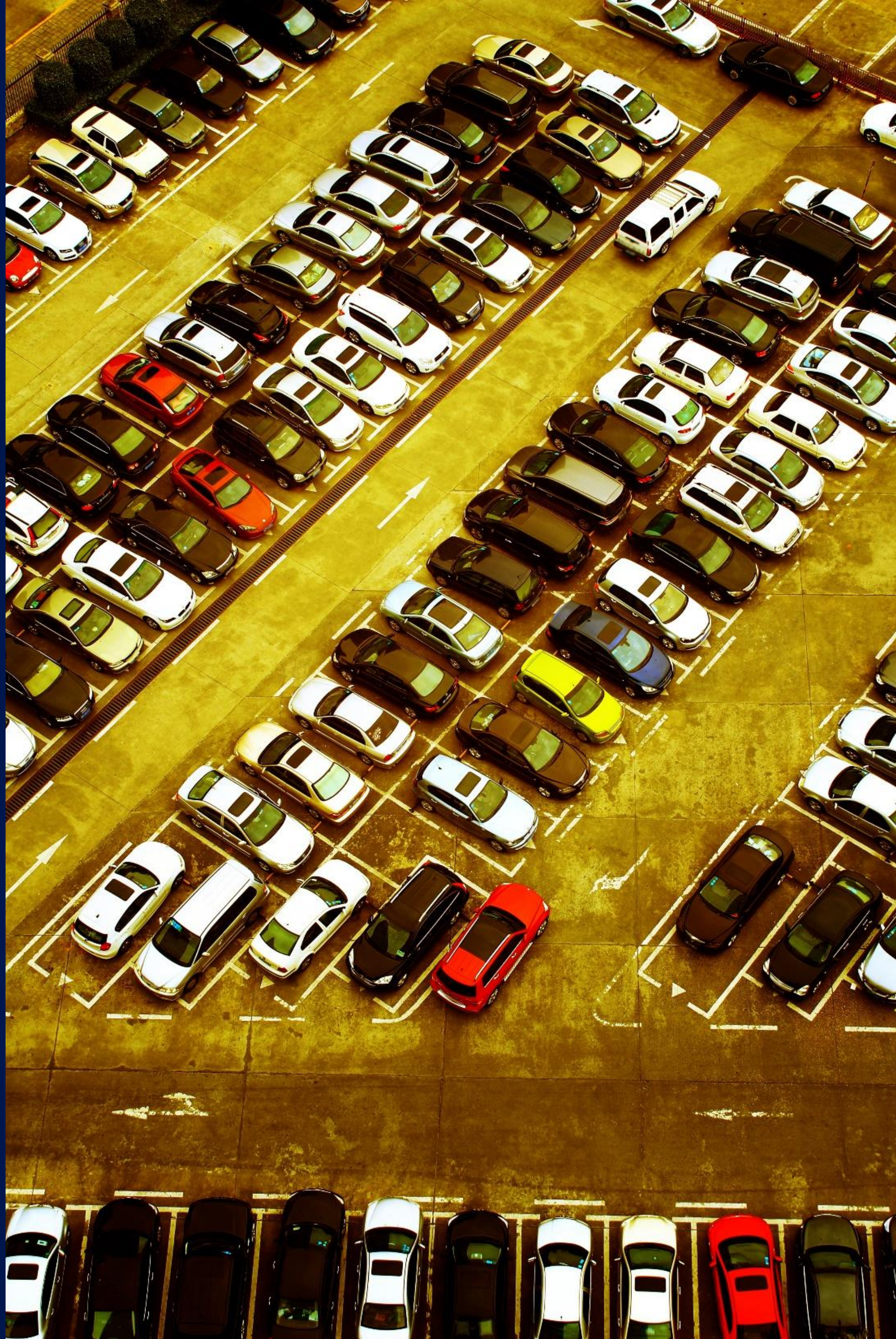
Sorting
Reorder elements in ascending or descending order, for later retrieval

Not all data structures are meant to be sorted.

Merging
Combining multiple containers' worth of data into one

Data Structure Operations

- These operations may be implemented quite differently from one data structure to the next.
- Learning how just a few data structures work gives you insight to how most will work.
- The concepts behind each of them will likely be similar.
  - There are only so many ways to allocate memory, redirect pointers, assign values, etc.
- For example, all cars “drive” or “turn on” but the engineering for any one car could be very different.



Operation Terminology

- Operations aren't always named the same thing.
- Computers don't care about names—these are for our benefit!

	std::vector	List (C# vector)	Stack	Queue	Linked List
Adding	push_back()	Add()	push()	enqueue()	addNode() addHead() addTail()
Removing	erase()	Remove() RemoveAt()	pop()	dequeue()	removeNode() removeHead() removeTail()
Searching	at() operator[]	Find()	pop() getTop() peek()	dequeue() getFirst()/getFront() peek()	getNode() Some sort of traversal function

Using Data Structures to Implement Other Data Structures

- An array is an array—we start here.
- A dynamic array is a class we build to be an inflexible array into something flexible—the details are hidden behind an interface.
- A **stack** could be a dynamic array we customize to work in a specific way (Last-in, first-out).
- A **queue** could be a dynamic array we customize to work in a specific way (First-in, first-out).
- We could implement stacks and queues as linked lists as well.
- Anyone using the class doesn't have to know about the internal details—but the person programming it (i.e., you!) certainly does!

“Standard” Containers

- Most data structures have a “standard” way of working (or close to it).
- Queue supports enqueue/dequeue as a standard.
  - Enter at the back of the line
  - Leave from the front of the line
  - That’s how lines work!
- Stacks support push and pop as a standard.
  - Put a card on the top of a deck (push).
  - Draw a card from the top of the deck (pop).
- You can add anything you need beyond these common “staples”.

Adding Custom Functionality

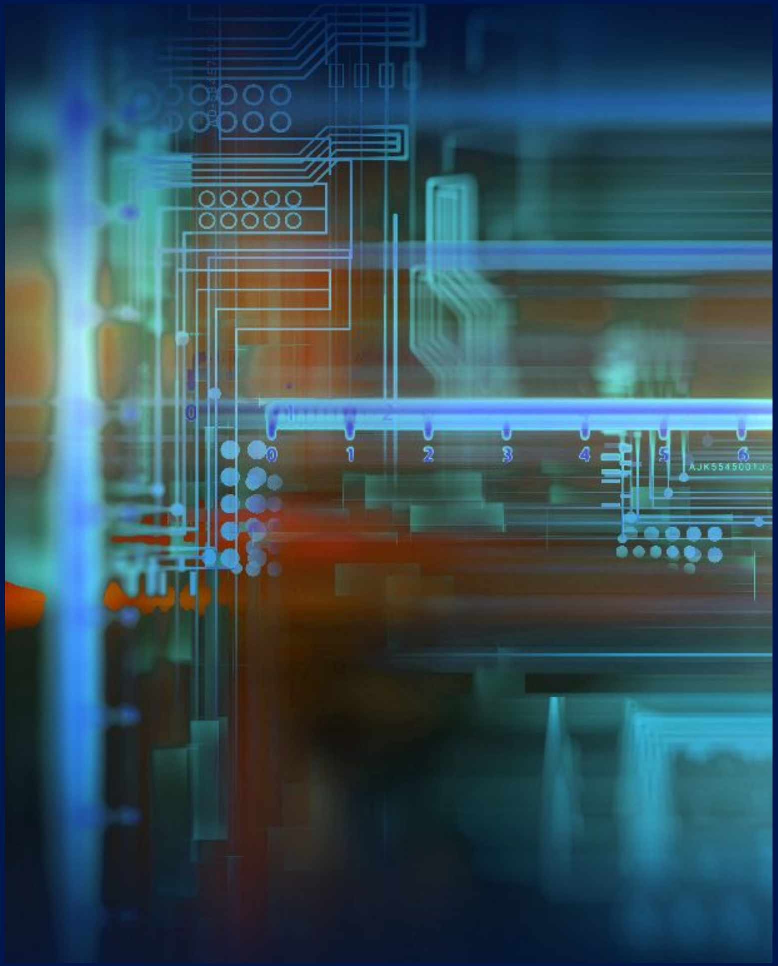
```
class Queue
{
    // Dynamic array for storage
    Person* _allPersons;
    int _count;

public:
    void Enqueue(Person newPerson);
    Person Dequeue();

    // Non-standard functions you might write
    void Enqueue_InMiddle(Person p, int index); // Friend saved your place?
    void Enqueue_AtFrontOfLine(Person p);       // Cut to the front?
    Person Dequeue_FromMiddle(int index);       // Someone leaves the line
};
```

If the basics aren't enough, customizing data structures can be very helpful!

Recap

- **Data structures** are classes that allow us to store data in a variety of ways.
- All data structures have **common operations** to add, remove or retrieve data.
- Each type of data structure has “standard” or typical functionality.
- Each type **can be customized** to fit the specific needs of your program.
- Like other programming features, **data structures are just another tool**—it’s always important to choose the right tool for the job
- There are a lot more data structures than we’ve looked at here.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.