ENGR 101 – Chapter 16

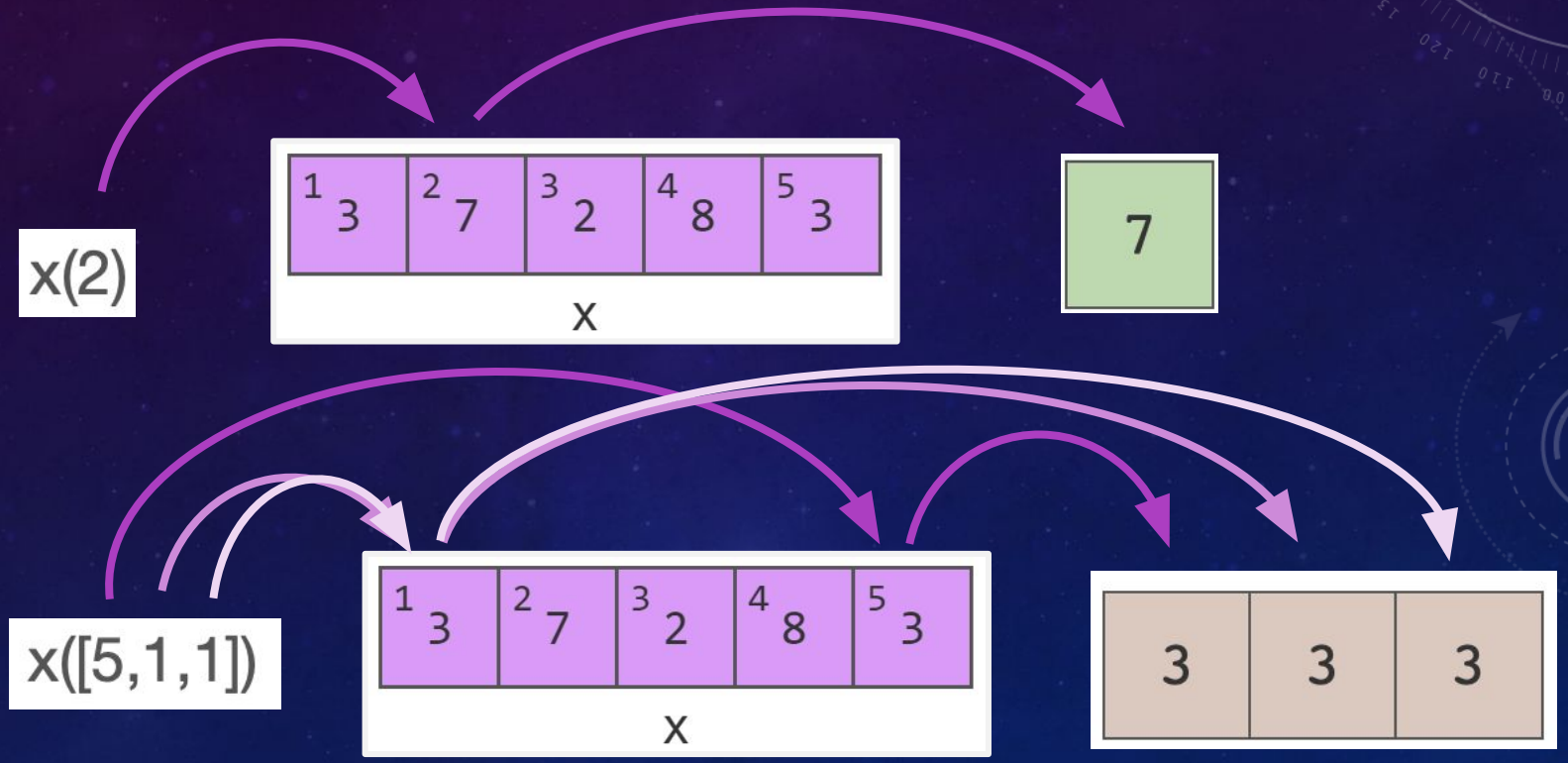
Vectors

3/13/21

INTRO

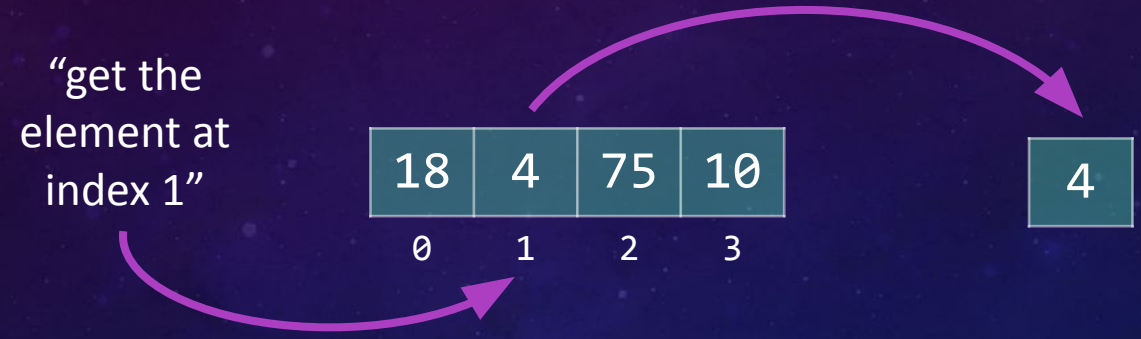
Recall: Using Indexing to Access Data in a Vector

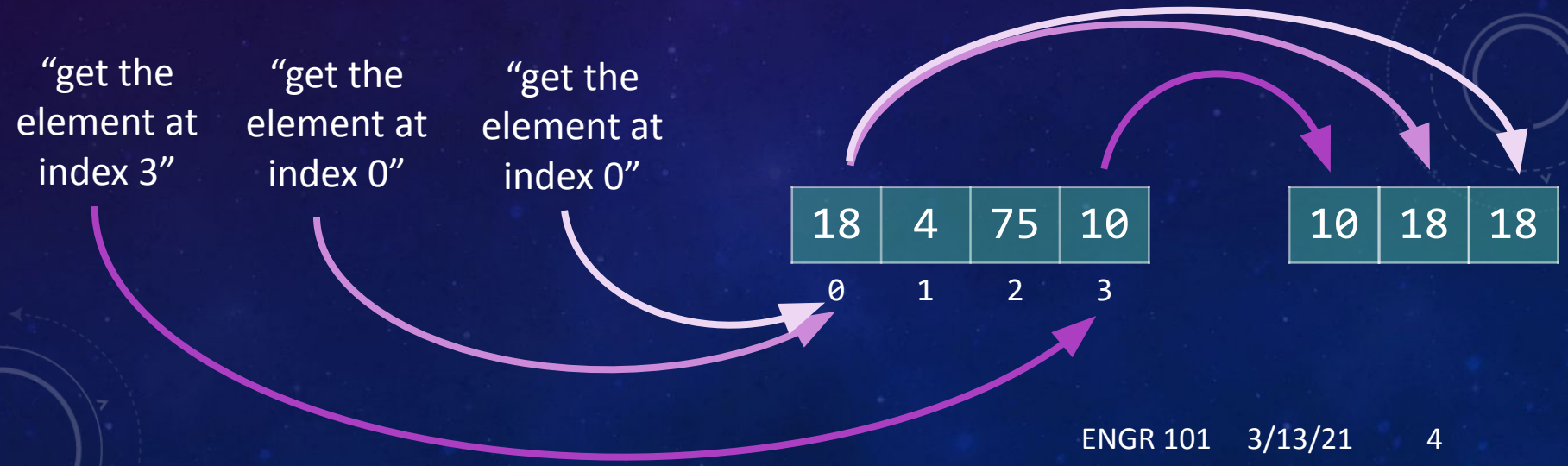
- Indexing in MATLAB uses **random access** to read and write data. The indices start at **1**.



Indexing in C++

- Indexing in C++ also uses **random access** to read and write data, but the indices start at **0** (just like with strings in C++).





DECLARING AND INITIALIZING VECTORS

vectors in C++

- In C++, a `vector` is used to store a sequence of elements.
  - The elements must be **homogenous** – i.e. all of the same type.

- This is conceptually similar to a vector in MATLAB, but the details are different.

- To use vectors, first include the vector library:

```
#include <vector>
using namespace std;
```

vectors in C++

- Declare a vector like this:

```
vector<int> someInts;
```

The name `someInts` will refer to the `vector` as a whole

In addition to the base type of `vector`, provide the type of elements it will hold.

- A vector can store elements of any type, as long as they match the type with which it is declared.

```
vector<double> someDoubles;
vector<bool> someBools;
```

Initializing a vector

- By default, a vector is empty.

```
vector<int> someInts;
```

- Specify one number to allocate an initial number of elements. Their values aren't initialized, though.

```
vector<int> someInts(5);
```

- You can provide a default initial value for all elements.

```
vector<int> someInts(4,42);
```

- You can provide different initial values for the elements.

```
vector<int> someInts{1,5,7};
```

This style needs C++11 to work. You may need to compile using `g++ -std=c++11`

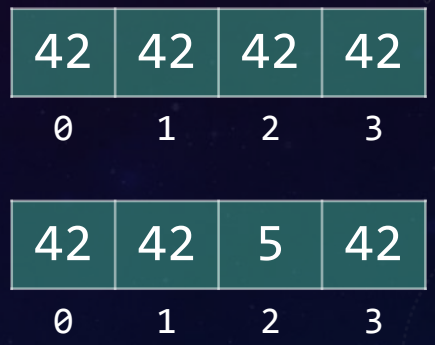
Internally, these lines are using different **constructors** for `vector`.

INDEXING INTO VECTORS IN C++

vector indexing

- Can use **indexing** with [ ] to work with individual elements.
  - As with `strings`, indices start with `0` and go up to `length - 1`.

```
vector<int> someInts(4,42);

someInts[2] = 5;

cout << "index 2: " << someInts[2] << endl;
```

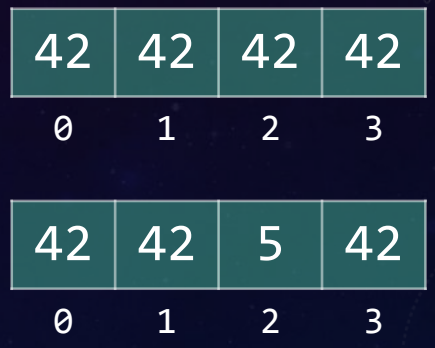



```
index 2: 5
```

vector indexing

- Can use **indexing** with [ ] to work with individual elements.
  - As with strings, indices start with 0 and go up to length – 1.

```
vector<int> someInts(4,42);

someInts[2] = 5;

cout << "index 3: " << someInts[3] << endl;
```




```
index 3: 42
```

Indexing Out Of Bounds

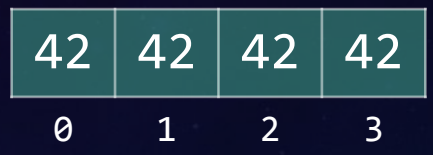
- As with `strings`, it's possible to index off the end of a `vector`, which results in undefined behavior at runtime.
- Basically, this goes to whatever memory happens to be next to the `vector`.
  - Maybe you get "lucky" and this memory wasn't important.
  - Maybe you mess up another variable that happens to be there.
  - Maybe your program isn't allowed to use that chunk of memory!
    - This causes a crash called a **segmentation fault** *(aka seg fault)*.
  - Maybe it catches on fire[1]. Who knows!

1 We're just kidding. This won't actually happen.

The at Function

- Again, you have the option to use the `.at` function rather than indexing with the square brackets.
  - This contains an implicit check to make sure the index is valid.
- The tradeoff is that `at` is slightly slower than `[]`.

```
vector<int> someInts(4,42);
```




```
someInts[4] = 43;
```

Causes undefined behavior. (Harder to debug.)

```
someInts.at(4) = 43;
```

Causes a runtime error with a nice message. (Easier to debug.)

VECTOR FUNCTIONS

Vector functions

- Many different functions can be called on a vector, using the dot notation. Here are a few common ones:

`size`	Returns the number of elements.
`front`	Returns a reference to the first element.
`back`	Returns a reference to the last element.
`at`	Works like indexing, but does bounds checking.
`empty`	Returns whether the `vector` is empty (as a `bool`).
`clear`	Removes all elements from the `vector`.
`push_back`	Adds a new element to the back of the `vector`.
`pop_back`	Removes the last element in the `vector`.

A Quick Technicality…

- The `vector size()` function returns an "unsigned" `int`.
  - Some compilers will give warnings about comparing this to a regular `int`.
- You can fix this by adding an explicit conversion to an `int`.

```
// Returns the value of the maximum element in the vector
int max_element(const vector<int> &vec) {

  int max_so_far = vec[0]; // assume first is largest

  // iterate through vector, looking for any larger
  for (int i = 0; i < int(vec.size()); ++i) {
    if (vec[i] > max_so_far) {
      max_so_far = vec[i];
    }
  }
  return max_so_far;
}
```

Tells the compiler to convert the result of `vec.size()` to a regular `int`.

TRAVERSING A VECTOR

Traversing a vector

- Use a loop to **traverse** through each element in a `vector`:

```
vector<int> vec(4,42);
vec[2] = 5;
```

42	42	5	42
0	1	2	3

`i` is called the **index variable**.

Keep going as long as `i` is within bounds.

Increment to next index on each iteration.

```
for (int i = 0; i < vec.size(); ++i) {
  cout << vec.at(i) << endl;
}
```

On each iteration, access the element at index `i`.

`vectors` have a `.size()` function.

3 min

Exercise: print

printVectorOfInts.cpp

- Write a function called `print`, that will print out a `vector<int>` in the format `{ _ _ _ _ }`, where `_` represents an element.

```
// Write the print function here




int main() {
  vector<int> someInts(4,42);
  someInts[2] = 5;
  print(someInts); // prints { 42 42 5 42 }
}
```

42	42	5	42
0	1	2	3

If you want to use C++11, you may need to compile thus:

```
g++ -std=c++11 printVectorOfInts.cpp -o printInts
```

Solution: print

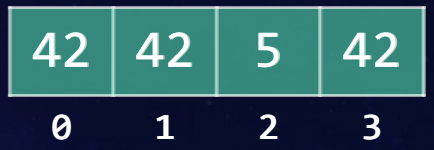
printVectorOfInts.cpp

- Write a function called `print`, that will print out a `vector<int>` in the format { _ _ _ _ }, where _ represents an element.

```
void print(vector<int> vec) {
  cout << "{ ";
  for (int i = 0; i < vec.size(); ++i) {
    cout << vec[i] << " ";
  }
  cout << "}" << endl;
}

int main() {
  vector<int> someInts(4,42);
  someInts[2] = 5;
  print(someInts); // prints { 42 42 5 42 }
}
```

This function is inefficient…why?

The pass by value copy is expensive!

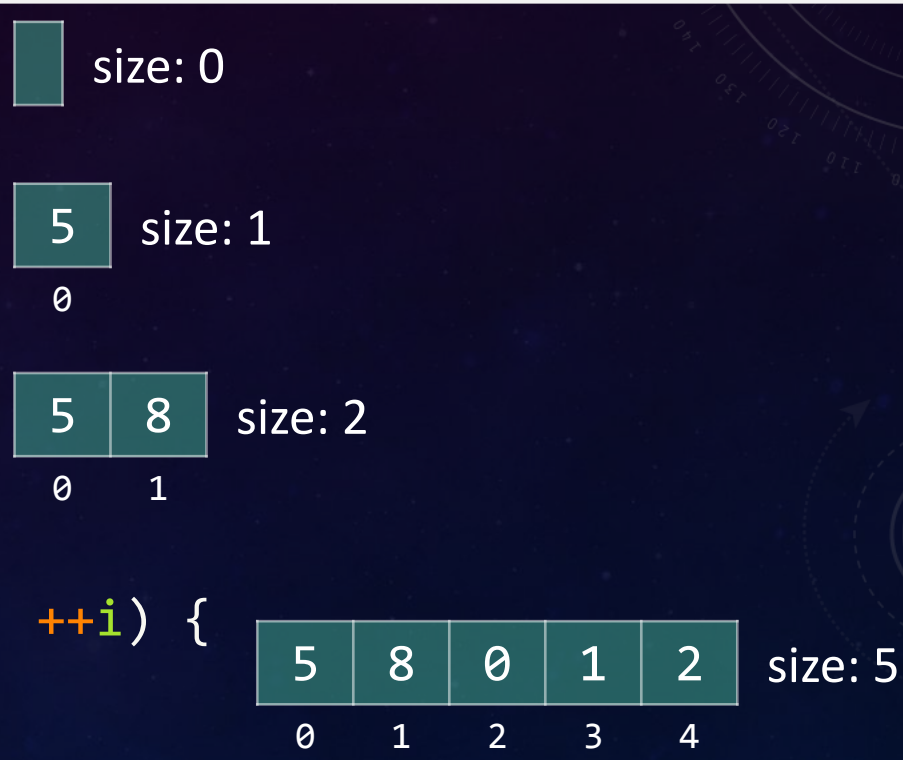


ADDING/REMOVING ELEMENTS FROM A VECTOR

Adding/Removing from a vector

- To add an element to the back, use `push_back`:

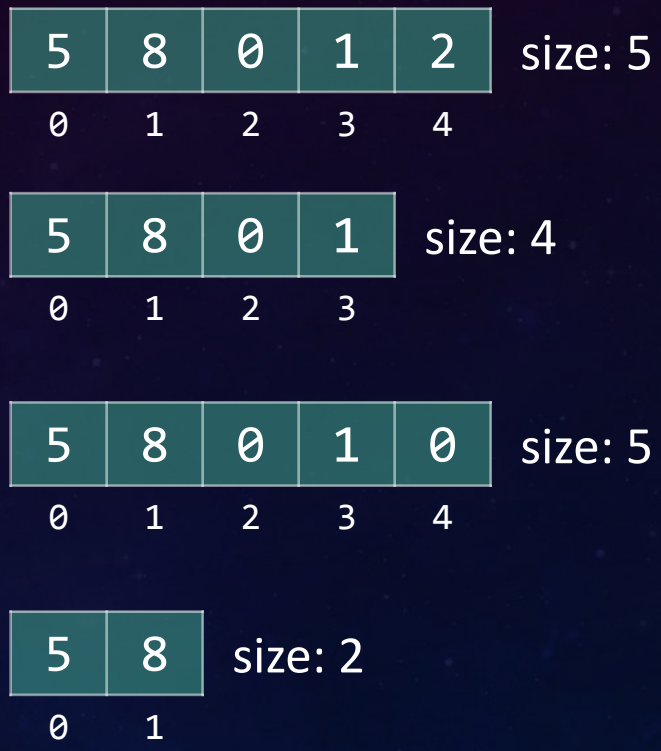
```
vector<int> vec;

vec.push_back(5);

vec.push_back(8);

for (int i = 0; i < 3; ++i) {
  vec.push_back(i);
}
```



Adding/Removing from a vector

- To remove an element from the back, use pop_back:

```
...
...
...

vec.pop_back();


vec.push_back(0);

vec.pop_back();
vec.pop_back();
vec.pop_back();
```



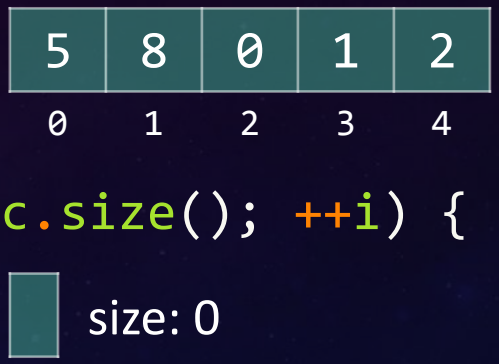
2 min

Exercise: Emptying a vector

- Both of these look good, but only one works. Which one?

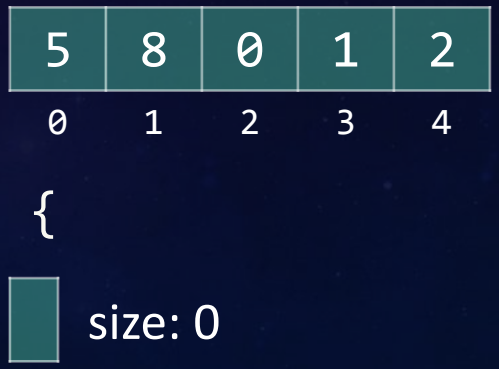
1

```
vector<int> vec;
...

for (int i = 0; i < vec.size(); ++i) {
  vec.pop_back();
}
```



2

```
vector<int> vec;
...

while (vec.size() > 0) {
  vec.pop_back();
}
```



Solution: Emptying a vector

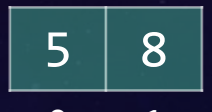
- Both of these look good, but only one works. Which one?

1

```
vector<int> vec;
...

for (int i = 0; i < vec.size(); ++i) {
  vec.pop_back();
}
```

The size changes!

size: 5



size: 2

The loop stops when `i` is 3 and size is 2.

2

```
vector<int> vec;
...

while (vec.size() > 0) {
  vec.pop_back();
}
```

size: 5

size: 0

THE .erase FUNCTION

vector erase

- To remove elements from a `vector`, use the `erase` function:

```
int main() {
  vector<int> vec;
  // Put 8, 6, 7, 5, 3, 0, 9 into the vector

  // this would remove the first element (e.g. the 8)
     vec.erase(vec.begin());




}
```

The `.begin()` function refers to the first element in the vector. When using the `erase` function, you have to specify indices relative to `.begin()` – it's just how the `erase()` function works.

vector erase

- To remove elements from a `vector`, use the `erase` function:

```
int main() {
  vector<int> vec;
  // Put 8, 6, 7, 5, 3, 0, 9 into the vector

  // this would remove the first element (e.g. the 8)
  // vec.erase(vec.begin());

  // this would remove the element at index 2 (e.g. the 7)
     vec.erase(vec.begin() + 2);



}
```

specify indices relative to `.begin()`

vector erase

- To remove elements from a `vector`, use the `erase` function:

```
int main() {
  vector<int> vec;
  // Put 8, 6, 7, 5, 3, 0, 9 into the vector

  // this would remove the first element (e.g. the 8)
  // vec.erase(vec.begin());

  // this would remove the element at index 2 (e.g. the 7)
  // vec.erase(vec.begin() + 2);

  // this version would remove a range of indices (e.g. 7, 5, 3)
     vec.erase(vec.begin() + 2, vec.begin() + 5);



}
```

Note that the upper bound is "exclusive", so the 5th element here wouldn't be removed.

vector erase

- To remove elements from a vector, use the erase function:

```
int main() {
  vector<int> vec;
  // Put 8, 6, 7, 5, 3, 0, 9 into the vector

  // this would remove the first element (e.g. the 8)
  // vec.erase(vec.begin());

  // this would remove the element at index 2 (e.g. the 7)
  // vec.erase(vec.begin() + 2);

  // this version would remove a range of indices (e.g. 7, 5, 3)
  // vec.erase(vec.begin() + 2, vec.begin() + 5);

  // this version erases all the way to the end
     vec.erase(vec.begin() + 2, vec.end());

}
```

PASSING VECTORS TO FUNCTIONS

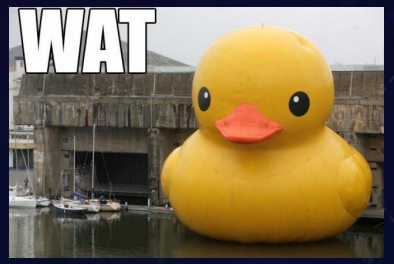
Pass by Reference

- Pass by reference gets rid of the expensive copy…
  - …but it's also risky, because it allows modification of the parameter.
- Consider this code:

Pass by reference: work with the original, not a copy.

```
bool anyZeros(vector <int> &vec) {
  for (int i = 0; i < vec.size(); ++i) {
    if ( vec[i] = 0 ) { return true; }
  }
  return false;
}

int main() {
  vector<int> someInts(4,42); // { 42 42 42 42 }
  cout << anyZeros(someInts) << endl; // prints false
  print(someInts); // prints { 0 0 0 0 }
}
```

Oops. We meant for this to be ==.



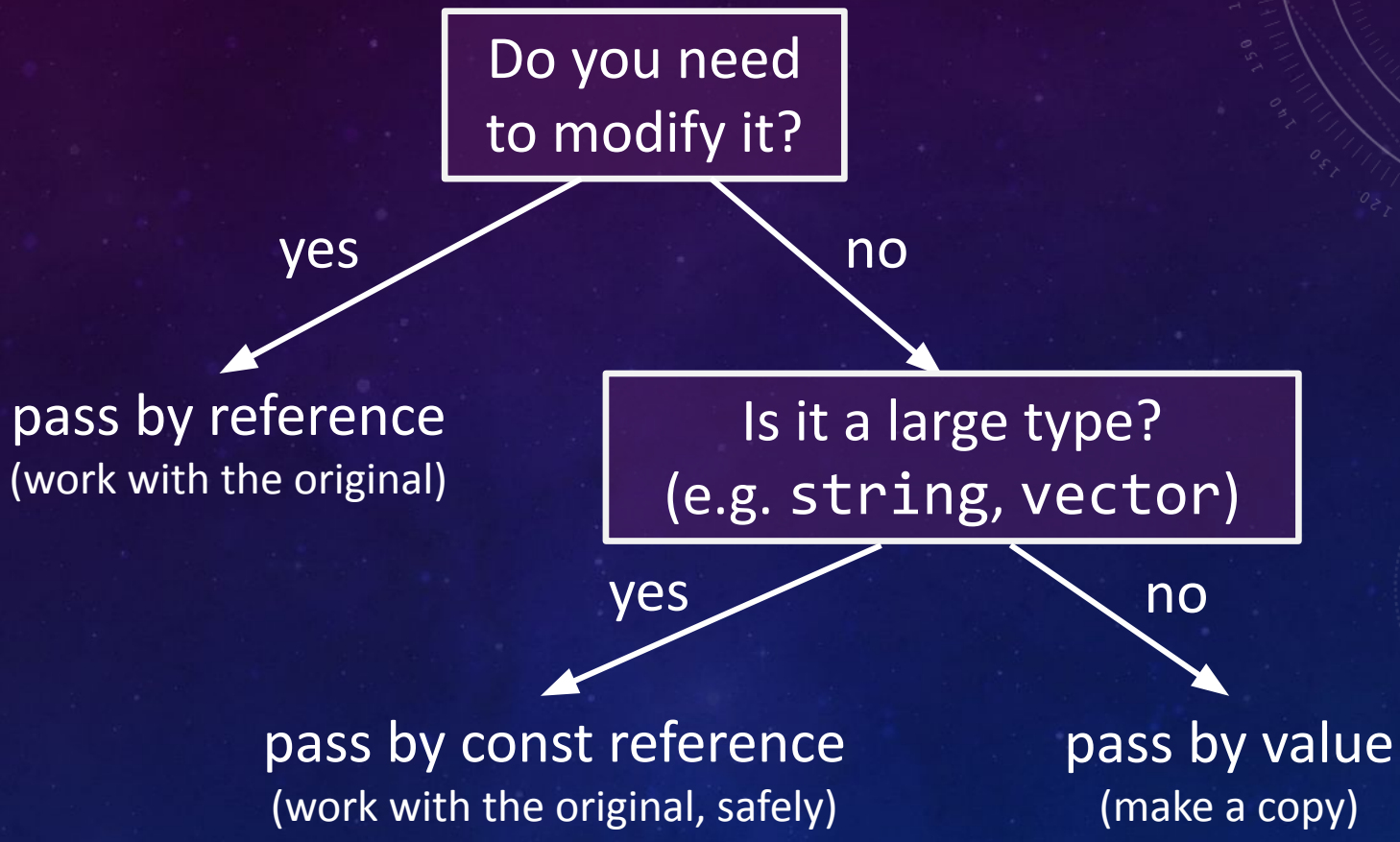
Pass by const Reference

- To prevent accidental modification, add the `const` keyword.
  - `const` is short for "constant".
  - You'll get a compiler error to warn you if you accidentally change it.

```
bool anyZeros(const vector &vec) {
  for (int i = 0; i < vec.size(); ++i) {
    if ( vec[i] = 0 ) { return true; }
  }
  return false;
}

int main() {
  vector<int> someInts(4,42);
  cout << anyZeros(someInts) << endl; // prints false
  print(someInts); // prints { 0 0 0 0 }
}
```

Error! vec is const.

Parameter Passing

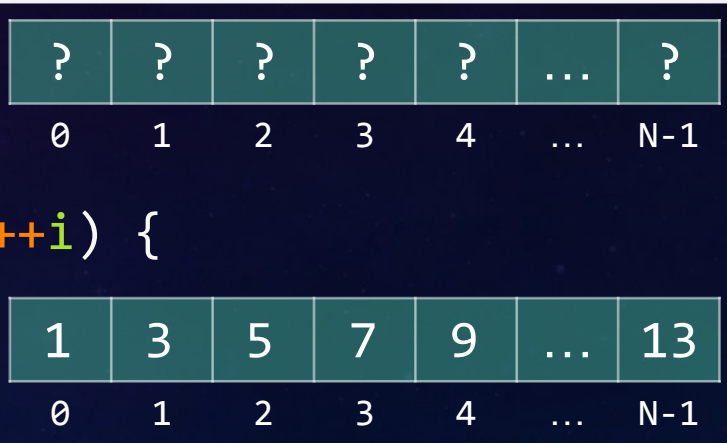


COMMON PATTERNS

Common Pattern: "Make space, then fill"

- If you know ahead of time how many elements you need:
  - First allocate enough elements.
  - Then fill in values.
- Example: Put the first N odd numbers in a vector

```
int N = 7;
vector<int> vec(N);

for (int i = 0; i < N; ++i) {
  vec[i] = 2 * i + 1;
}
```

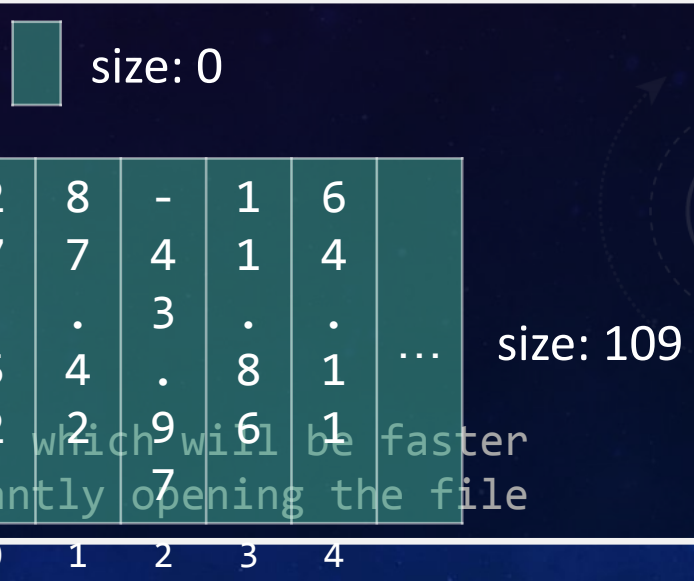


Common Pattern: "Fill as you go"

- If you don't know ahead of time how many elements you need:
  - Just add them as you go by using `push_back`.
  - The vector will grow as needed to accommodate everything.
- Example: Read values into a `vector` from a data file.

Now `double` is our element type.

```
vector<double> data;
ifstream fileIn("sensor.dat");
double value;
while (fileIn >> value) {
  data.push_back(value);
}
fileIn.close();
// Now we can use the data vector, which will be faster
// and more convenient than constantly opening the file
```



Common Pattern: Using an "Accumulator"

- Task: Compute the result of combining a set of elements
  - Example: Find the sum or product of elements in a vector
- Strategy: *Start a "running total" with the identity for the operation you're using. Then add elements one at a time.*

```
// Returns the sum of elements in a vector
int sum(const vector<int> &vec) {

  int sum = 0; // start at 0 because it's the additive identity

  // iterate through vector, adding each one to running total
  for (int i = 0; i < vec.size(); ++i) {
    sum += vec[i];
  }
  return sum;
}
```

Common Pattern: Finding the "Best" Element

- Task: Find the "best" element according to some criteria.
  - Example: Find the maximum or minimum
- Strategy: *Keep track of the best so far, compare it to each element, and replace if you find a better one.*

```
// Returns the value of the maximum element in the vector
int max_element(const vector<int> &vec) {

  int max_so_far = vec[0]; // assume first is largest

  // iterate through vector, looking for any larger
  for (int i = 0; i < vec.size(); ++i) {
    if (vec[i] > max_so_far) {
      max_so_far = vec[i];
    }
  }
  return max_so_far;
}
```

Note: This function can't be used on an empty `vector`.

3 min

analyze.cpp

Exercise: Find the Minimum

- Write a function called `minVal`, that will return the minimum value of a vector of `doubles`

```
      minVal(                ) { // complete this line

// Write the minVal function here
}

int main() {
// code provided to read data from sensor.dat
cout << "The lowest sensor reading was: "
cout << minVal(data) << endl; // prints -98.44

}
```

If you want to use C++11, you may need to compile thus:

```
g++ -std=c++11 analyze.cpp –o analyze
```

Solution: Find the Minimum

analyze.cpp

```
double minVal(const vector<double> &vec) {

  // assume first is smallest
  double min_so_far = vec.at(0);

  // iterate through vector, looking for any smaller
  for (int i = 0; i < vec.size(); ++i) {
    if (vec.at(i) < min_so_far) {
      min_so_far = vec.at(i);
    }
  }

  return min_so_far;
}
```

Question

We can't use this approach to find the smallest element and change it. Why not?

If you want to use C++11, you may need to compile thus:

```
g++ -std=c++11 analyze.cpp -o analyze
```

Common Pattern: Finding the Index of the "Best" Element

- Find the index of the minimum element rather than it's value.
  - This is useful for looking the element up later to change it.

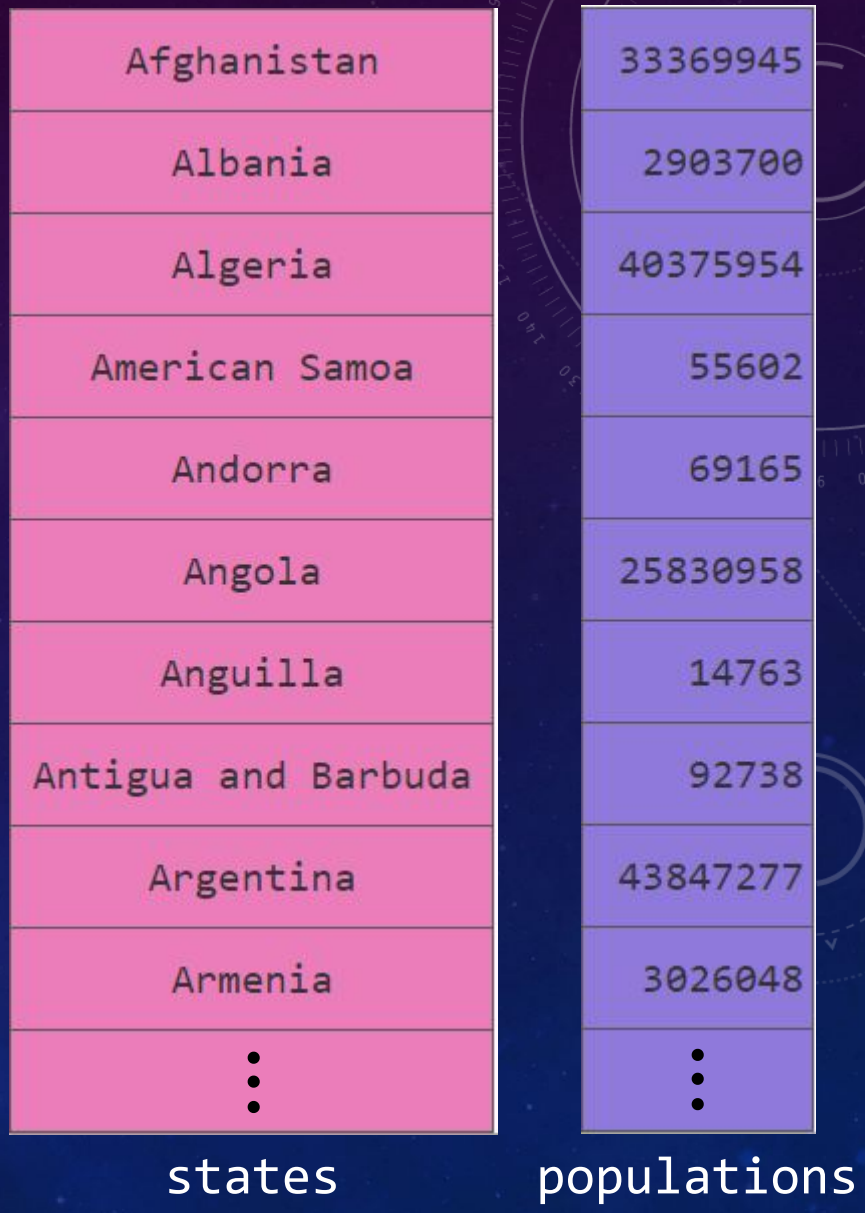
```
// Returns the value of the minimum element in the vector
int index_of_min_element(const vector<int> &vec) {

  int index_of_min = 0; // start at first index

  for (int i = 0; i < vec.size(); ++i) {
    if (vec[i] < vec[index_of_min]) {
      index_of_min = i;
    }
  }

  return index_of_min;
}
```

We keep track of the index of the best element instead of the value.

Recall: Parallel Vectors

- `states`
  This is a column vector containing a list of the states of Earth in ascending alphabetic order. This variable actually contains a different "type" of data called a `string` that is used to represent words.
- `populations`
  A parallel column vector to states that stores the population of the corresponding state.
- The index number links the two vectors together

`states.at(3)` `populations.at(3)`



Source: https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)

Common Pattern: Accessing Parallel Vectors

- To get data that is “parallel”, access each vector using the same index number
- Example: Displaying state names and populations.

```
vector<string> states;
vector<string> populations;
...
// Display first state
cout << "The first state is: " << states.at(0);
cout << " -- population " << populations.at(0) << endl;

// Display 10th state
cout << "The first state is: " << states.at(9);
cout << " -- population " << populations.at(9) << endl;

// Display last state
cout << "The first state is: " << states.at(states.size() - 1);
cout << " -- population " << populations.at(states.size() - 1) << endl;
```

*As we will soon see, an alternative often used in C++ is to create a custom data type that encapsulates both a state's name and population.

Common Pattern: Any/All

- Task: Check if any (or all) element(s) match some criteria
  - Example: Are there any zeros? Are all elements positive?
- Strategy: *Always frame it as an "any" question. Then use a loop with **early termination** to check for any such element.*

```
// Returns whether there are any zeros in the vector
bool any_zeros(const vector<int> &vec) {

  // iterate and check for any zeros
  for(int i = 0; i < vec.size(); ++i) {
    if ( vec[i] == 0 ) {
      return true;
    }
  }
  return false;
}
```

Checking for any element that matches. If one is found, return immediately! (No need to check the rest.)

If we make it to here, there weren't any.

Common Pattern: Any/All

- Task: Check if any (or all) element(s) match some criteria
  - Example: Are there any zeros? Are all elements positive?
- Strategy: *Always frame it as an "any" question. Then use a loop with **early termination** to check for any such element.*
  - *Use negation to turn an "all" question into an "any" question.*

```
// Returns whether all the elements in the vector are positive
bool all_positive(const vector<int> &vec) {

  // iterate and check for any non-positives
  for(int i = 0; i < vec.size(); ++i) {
    if ( !(vec[i] > 0) ) {
      return false;
    }
  }
  return true;
}
```

Checking for counterexamples. If one is found, return immediately!

If we make it to here, they must have all been ok.

Common Pattern: Searching for a Value

- Task: Find the location of a particular element.
  - Example: At what index does the value 0 first occur?
- Strategy: *This is similar to an "any" question, but we return the current index instead of* `true`.

```
// Returns the index at which the given value first occurs in
// the vector. If it is not present, returns -1
int find(const vector<int> &vec, int value) {

  // iterate and check for the value
  for(int i = 0; i < vec.size(); ++i) {
    if ( vec[i] == value ) {
      return i;
    }
  }
  return -1;
}
```

Checking for any element that matches. If one is found, return the index!

If we make it to here, there weren't any. Use -1 to represent "no elements found".