Lecture 5

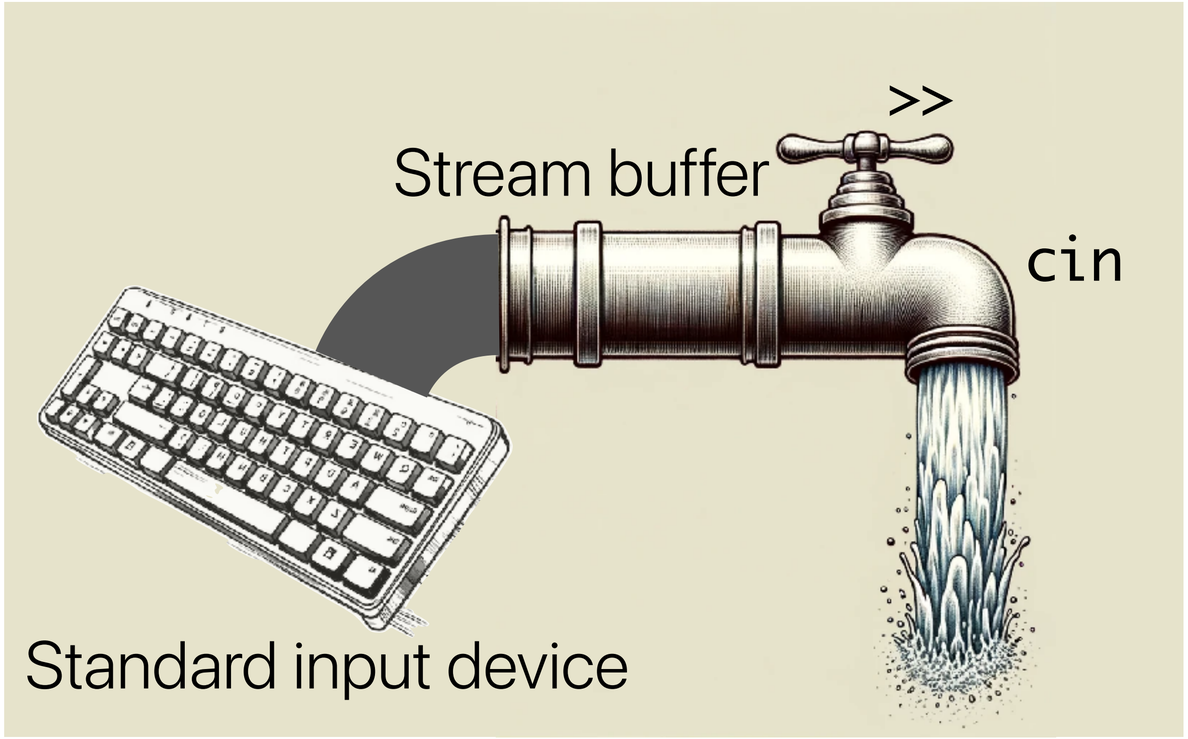
# C++ Standard Library (2)

Yohan Jo

# Previous Lecture

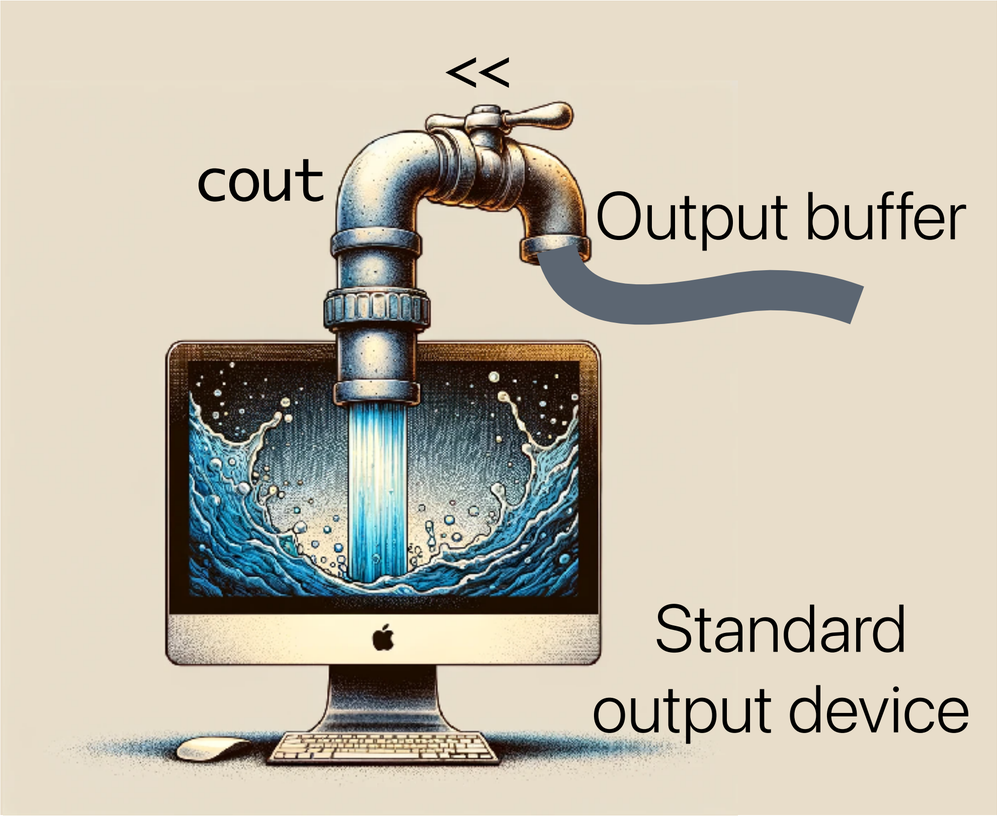
- C++ Standard Library
  - **Standard I/O streams**
  - File I/O streams
  - Strings
  - Containers

# Previous Lecture




- `cin` is an instance of the `istream` class to read input from the standard input device (keyboard)
- `cin` can be likened to a faucet with a water pipe and the >> operator to the action of turning it on
- The >> operator extracts as many characters as expected by the data type of the variable on the right or until whitespace appears, and assigns them to the variable

# cout




- `cout` is an instance of the `ostream` class to write output to the standard output device (console)
- `cout` can be likened to a faucet with a water pipe, and the << operator to the action of turning it on
- The `endl` manipulator inserts `'\n'` and flushes the output buffer
- If not manually flushed, the buffer is flushed under certain conditions

# Overview

- C++ Standard Library
  - Standard I/O streams
  - **File I/O streams**
  - **Strings**
  - **Containers**
- Quiz 2 (next lecture)

# File I/O Streams

# File I/O Streams

- File input and output (I/O) is essential for data processing
- File I/O operations in C++ are handled using the `ifstream` (input file stream) and `ofstream` (output file stream) classes
- These classes are part of the C++ Standard Library's `<fstream>` header
- File I/O streams can be used in a very similar way to standard I/O streams

# File I/O Streams – Reading

- Open a file for reading
  - `ifstream file("PATH");`
  - `file.open("PATH");`
- Check if the file is successfully open
  - `file.is_open()`
- Read each line
  - `getline(ifstream, string)`
  - Returns the file stream
- Close the file using `file.close()`

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream file("data.txt");
    if (!file.is_open()) {
        std::cerr << "Failed to open file" <<
std::endl;
        return 1;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << std::endl;
    }

    file.close();
    return 0;
}
```

# File I/O Streams – Reading

- Without the `getline` function, `ifstream` extracts characters from the input buffer until whitespace appears or as expected by the data type of the variable on the right, as in `cin`
- Indeed, `ifstream` inherits from the `istream` class (the class of `cin`), exhibiting similar behavior for the extraction operator >>

```
data.txt

1 2
3
```

```
std::ifstream file("data.txt");
int number;
while (file >> number) {
    std::cout << number << std::endl;
}
file.close();

/*
Output:
1
2
3
*/
```

# File I/O Streams – Writing

- Open a file for writing
  - `ofstream file(<PATH>);`
  - `file.open(<PATH>);`
- Check if the file is successfully open using `file.is_open()`
- Write data to the file as you would do with the standard output stream `cout`
- Close the file using `file.close()`

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("output.txt");
    if (!file.is_open()) {
        std::cerr << "Failed to open file"
<< std::endl;
        return 1;
    }

    file << "Hello, Data Science!" <<
std::endl;

    file.close();
    return 0;
}
```

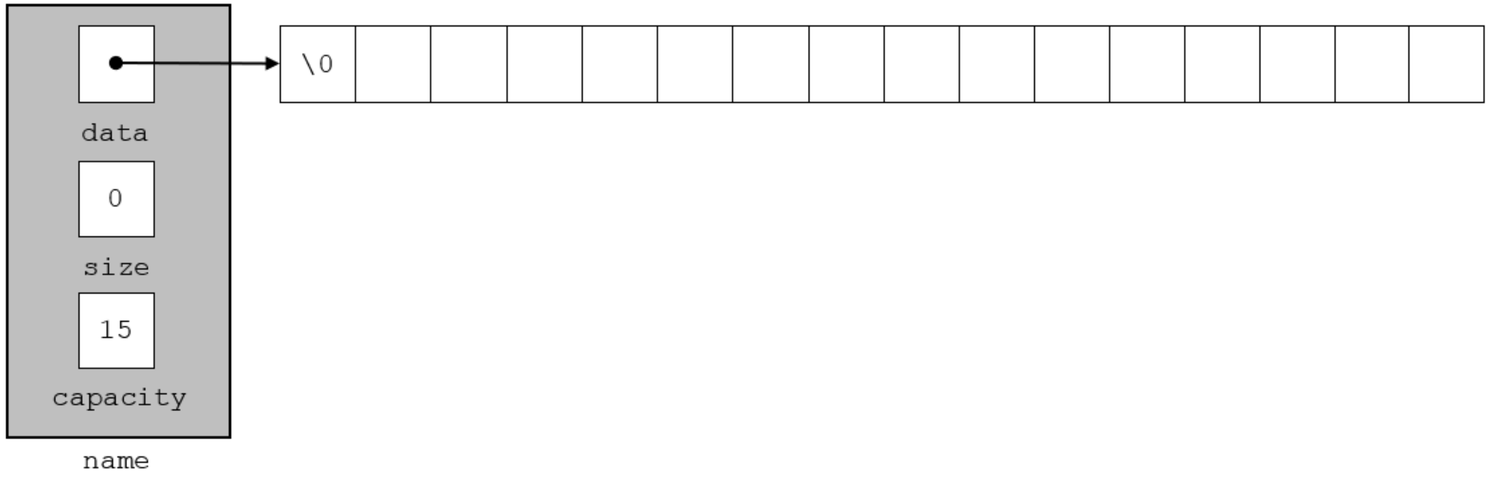
# Strings

# Strings

- In C, a string is typically represented as a character array
- The C++ Standard Library provides the `string` class, which internally uses a dynamic array for handling a sequence of characters

# Strings

- `string` maintains several information internally, including:
  - **Data:** A pointer to a char array (i.e., a contiguous memory block) that contains the text data
  - **Size:** The length of the current text data
  - **Capacity:** The number of characters that the char array can hold



# Strings – Initialization

- `std::string` is included by adding "`#include <string>`"
- Ways to initialize a `string`
  - Copy initialization: The RHS is copied into the newly created object `str2`
  - Direct initialization: Directly calls a constructor of the string class

```cpp
#include <string>

int main() {
    std::string str1; // Empty
    std::string str2 = "String2";
    std::string str3("String3");

    return 0;
}
```

# Strings – Concatenation

- Strings can be concatenated using the + operator

```
string firstName = "Data";
string lastName = "Scientist";
string fullName = firstName + " " + lastName;
cout << fullName << endl; // Output: Data Scientist
```

- A string can be appended to another using the += operator or the *append* method

```
string name = "Data";
name += " Scientist";
cout << name << endl;
// Output: Data Scientist
```

```
string name = "Data";
name.append(" Scientist");
cout << name << endl;
// Output: Data Scientist
```

# Strings – Comparison

- Strings can be compared using the ==, !=, <, > operators or the `compare` method
- Comparison is lexicographical

```
string str1 = "Apple";
string str2 = "Banana";

if (str1 == str2) { ... }
else if (str1 > str2) { ... }
else if (str1 < str2) { ... }
else { ... }
```

```
string str1 = "Apple";
string str2 = "Banana";

cout << str1.compare(str2); // -1
cout << str2.compare(str1); // 1
cout << str1.compare(str1); // 0
```

# Strings – Finding Substrings

- Use the `find` method to locate a substring within a string

```
string fullName = "Data Science";
size_t pos = fullName.find("Science");
if (pos != string::npos) {
    cout << "Found 'Science' at position: " << pos << endl;
}
// Output: Found 'Science' at position: 5
```

  - `size_t` is an unsigned integer type that is used to represent sizes and counts
  - `string::npos` is a constant static member of the `string` class representing the largest possible value for `size_t`
    - `find()` returns `string::npos` if it fails to find a substring

# Strings – Finding Substrings

- `substr(position, length)` returns a substring of the string, starting from the `position` and having the `length`

```
string str = "Data Science";
cout << str.substr(5, 3);
```

```
// Output: Sci
```

# Strings – Replacing Substrings

- `replace(start, length, replacement)` replaces the substring that begins at position `start` of length `length` with the substring `replacement`

```
string str = "Data Science";
string replacedString = str.replace(0, 4, "Bio");
cout << replacedString << endl;
```

```
// Output: Bio Science
```

# Strings – Conversion

- `std::stoi` and `std::stod` convert a string to an integer and a double, respectively

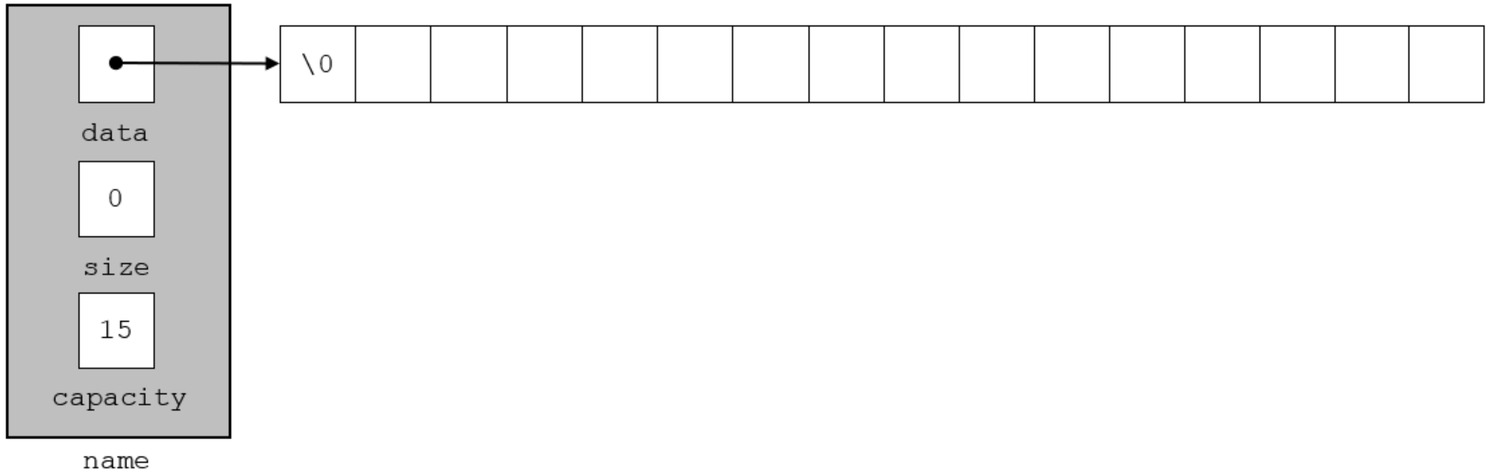
```
std::string number = "42";
int intNum = std::stoi(number);
double doubleNum = std::stod(number);
```

- `std::to_string` converts a number to a string

```
double doubleValue = 123.456;
std::string doubleStr = std::to_string(doubleValue);
```

# Strings – Memory

- `string` maintains several information internally, including:
  - **Data:** A pointer to a char array (i.e., a contiguous memory block) that contains the text data – the array is reallocated when the size exceeds the capacity
  - **Size:** The length of the current text data
  - **Capacity:** The number of characters that the char array can hold



# Strings – Memory

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string myString = "Hello, World!";
    cout << "Initial Size: " <<
myString.size() << endl; // 13
    cout << "Initial Capacity: " <<
myString.capacity() << endl; // 22
    cout << "Initial Memory Address: " <<
(void*)myString.c_str() << endl; //
0x7ff7b33eb241

    // Append a small string that doesn't
exceed the current capacity
    myString += "!";
    cout << "\nAfter small append:" << endl;
    cout << "Size: " << myString.size() <<
endl; // 14
    cout << "Capacity: " <<
myString.capacity() << endl; // 22
    cout << "Memory Address: " << (void*)
myString.c_str() << endl; // 0x7ff7b33eb241

    // Append a large string to force
reallocation
    myString += " A large string";
    cout << "\nAfter large append:" << endl;
    cout << "Size: " << myString.size() <<
endl; // 29
    cout << "Capacity: " <<
myString.capacity() << endl; // 47
    cout << "Memory Address: " << (void*)
myString.c_str() << endl; // 0x7ff2b6f05e30

    return 0;
}
```

# Strings – Memory

- If appending characters to the string requires more space than the current capacity of the char array, the array might be reallocated to a new memory location with a larger size to accommodate the additional characters

# Stringstreams

- A `stringstream` object treats strings as streams, allowing for both reading from and writing to strings as if they were files or the console
- It's included in the `<sstream>` header

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main() {
    // Let's parse a text by commas
    std::stringstream parser("42,3.14,Hello World");
    int intValue;
    double doubleValue;
    std::string strValue;
    char ignoreChar; // Used to ignore the commas

    parser >> intValue >> ignoreChar >> doubleValue >>
ignoreChar;
    std::getline(parser, strValue); // Read the remainder

    std::cout << "Integer: " << intValue << ", Double: "
<< doubleValue << ", String: " << strValue << std::endl;
    // Integer: 42, Double: 3.14, String: Hello World

    return 0;
}
```

# Stringstreams

- Avoid mixing reading and writing to the same `stringstream`
- This mixing may lead to an unexpected result unless the `stringstream`'s state is properly adjusted in-between

```cpp
#include <iostream>
#include <sstream>

int main() {
    std::stringstream ss;

    ss << 100;
    ss << 3.14;
    ss << "Hello";
    std::cout << ss.str() << std::endl;
    // Output: 1003.14Hello

    return 0;
}
```

# Containers

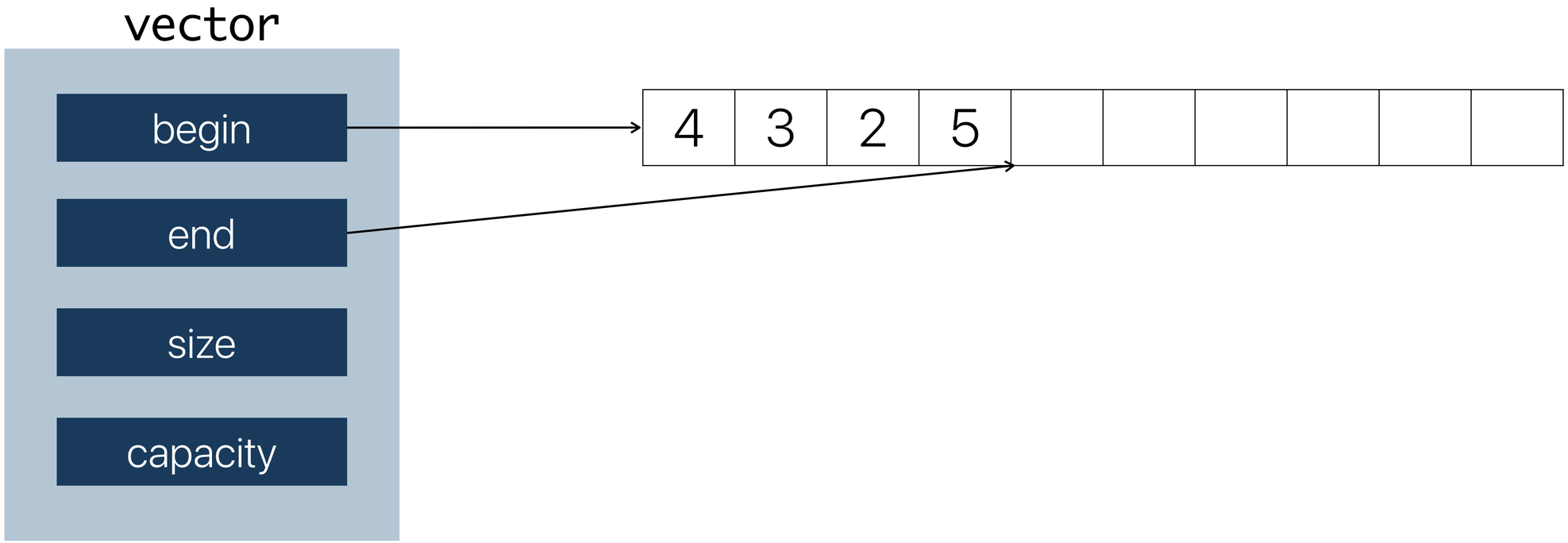
# Containers

- C++ Standard Library provides useful data structures for containers
  - `vector`
  - `list`
  - `map`
  - `set`
  - `deque`
  - `unordered_map`
  - `unordered_set`

# `vector`

- A vector is a sequence container that encapsulates dynamic-size arrays, similar to `list` in Python
- Elements are stored in contiguous storage
- Supports random access to elements, i.e., direct access by their position index
- Offers efficient insertion and deletion of elements at the end
- Resizes automatically when elements are added beyond their capacity or removed

# vector




This figure is conceptual, and details may be different from actual implementations

# vector – Initialization

- Include the <vector> header
- When declaring a vector, the data type of its elements should be specified
  - E.g., std::vector<int>, std::vector<std::string>
  - Unlike Python's list, vector can contain only one type of data

```cpp
#include <iostream>
#include <vector>

int main() {
    // Direct list initialization
    std::vector<int> vec1 = {1, 2, 3, 4, 5};

    // From an array
    int arr[] = {6, 7, 8, 9, 10};
    std::vector<int> vec2(
        std::begin(arr), std::end(arr));

    // With a specific size and value
    std::vector<int> vec3(5, 100);

    return 0;
}
```

# vector – Insertion

- Insert an element using the `push_back` or `insert` methods

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

int main() {
    vector<int> vec;

    // Adding elements using push_back
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);

    // Printing the second element
    cout << "Second element: " << vec.at(1)
<< endl;

    return 0;
}
```

# vector – Deletion

- Delete elements using the `pop_back` and `erase` methods

```
vector<int> vec = {10, 20, 30, 40, 50};

// Delete the last element
vec.pop_back();

// Delete the element at the third position
vec.erase(vec.begin() + 2);
```

  - `vec.begin()` returns an iterator that points to the first element

# vector

- `push_back(value)`: Adds an element to the end
- `pop_back()`: Removes the last element
- `at(index)`: Returns a reference to the element at a specified position, with bounds checking
- `size()`: Returns the number of elements
- `capacity()`: Returns the size of the storage space currently allocated to the vector
- `resize(n)`: Resizes the container so that it contains n elements
- `empty()`: Checks if the container has no elements

# Iteration – Range-Based For Loops

- C++11 supports range-based for loops
  - `for (declaration : range) statement;`
  - Similar to "`for ... in ...`" in Python
- Range-based loops are often used with the `auto` keyword to automatically deduce the data type of the elements
  - `auto` indeed can be used in various declarations when data types can be inferred

```cpp
int main() {
    vector<int> vec = {1, 2, 3};

    for(int val : vec) {
        cout << val << endl;
    }

    for(auto val : vec) {
        cout << val << endl;
    }

    return 0;
}
```

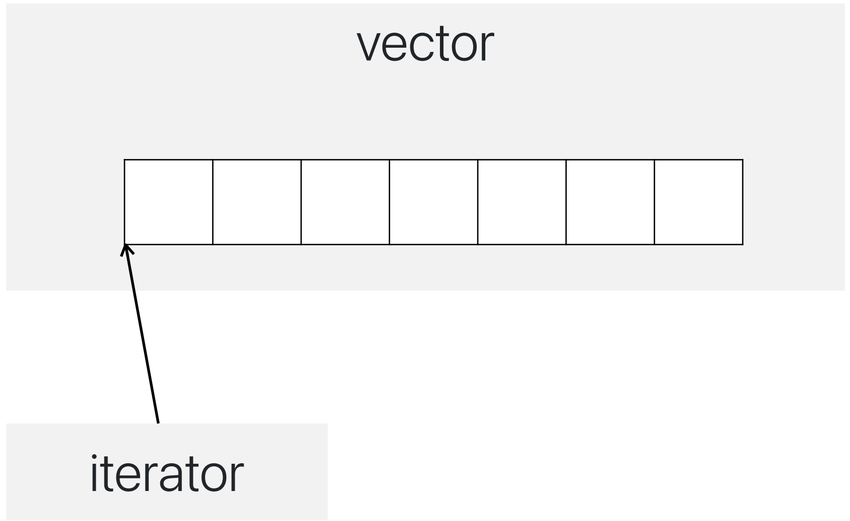
# Iteration – Iterators

- Using an iterator to iterate over a vector

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

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Iterate using an iterator
    for(vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << endl;
    }

    return 0;
}
```

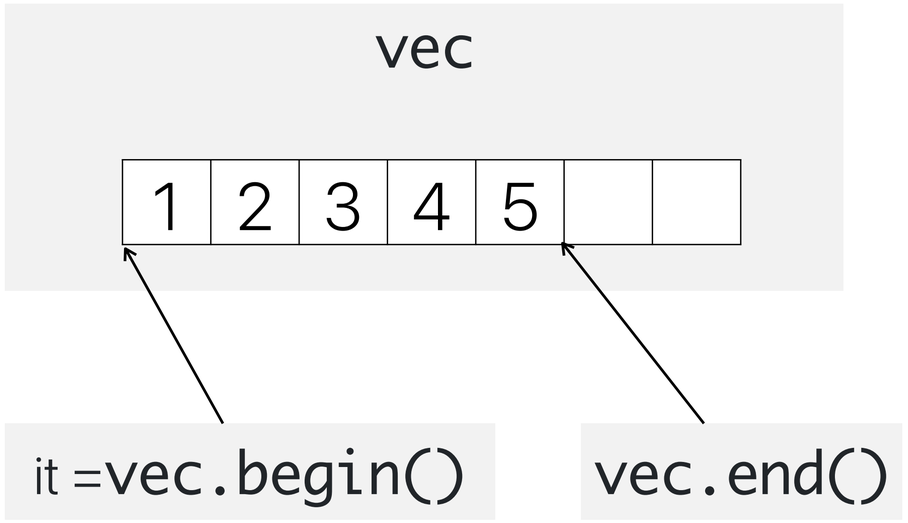
# Iteration – Iterators

- An iterator is an object that internally points to an element within a container, providing a way to access and navigate container elements
- Many container classes (e.g., vector, list, map, set) define their own iterator class within (e.g., `set<int>::iterator`)
- But iterators from different container classes are designed to serve similar functionalities
- While iterators are not pointers per se, they support similar syntax to pointers (e.g., `*it`)



# Iteration – Iterators

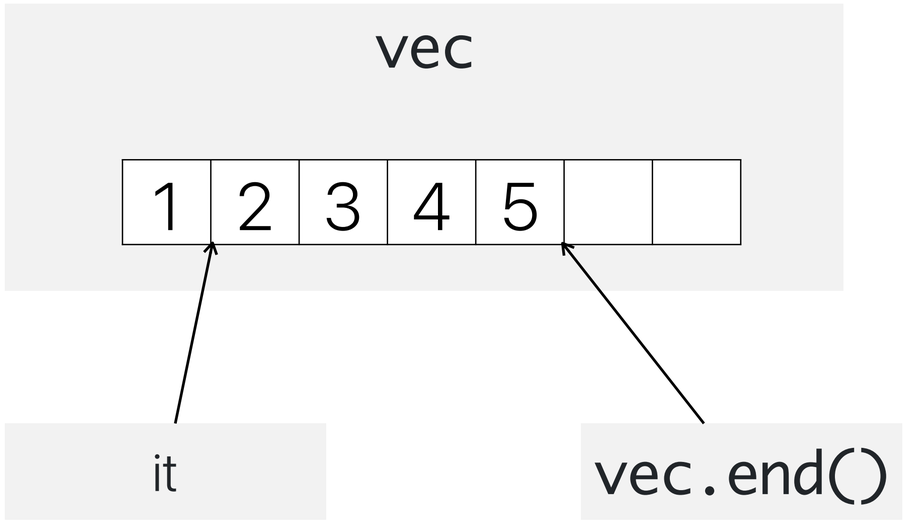
```
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```




- `vector`'s `begin` and `end` methods return iterators that point to the first element and one past the last element

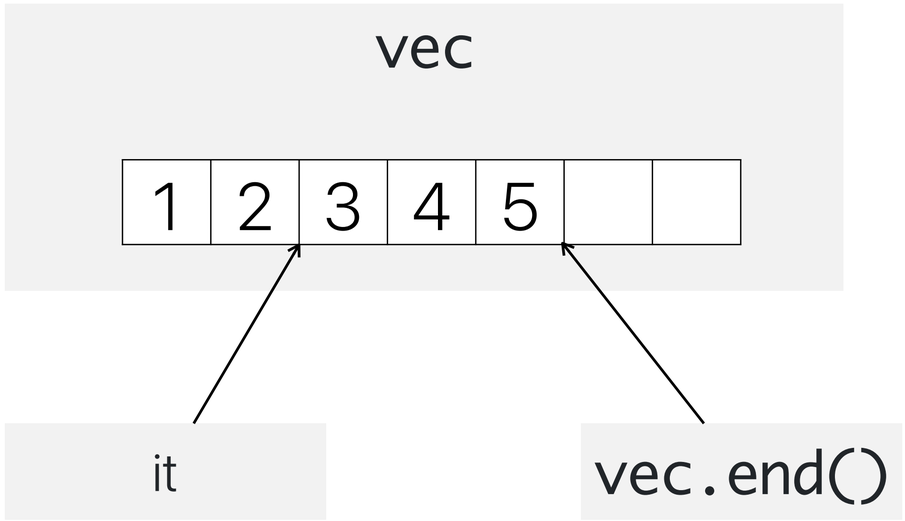
# Iteration – Iterators

```
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```



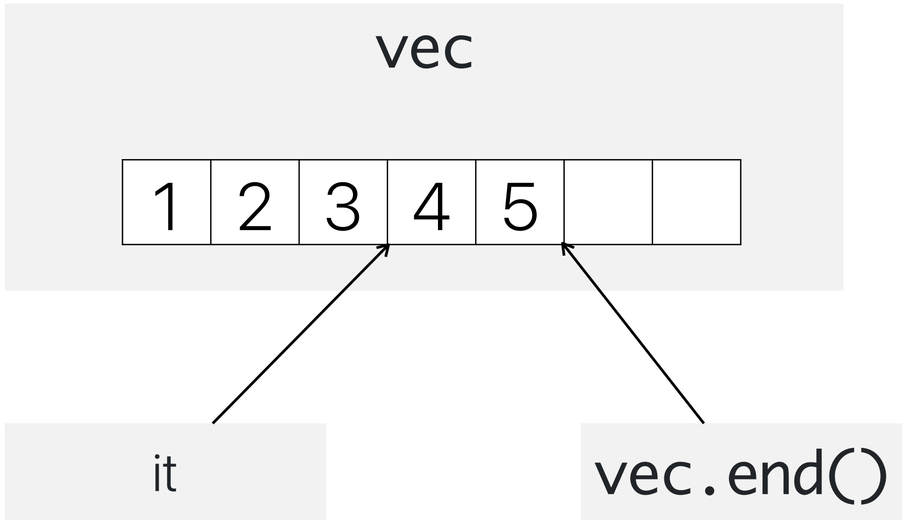
# Iteration – Iterators

```
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```



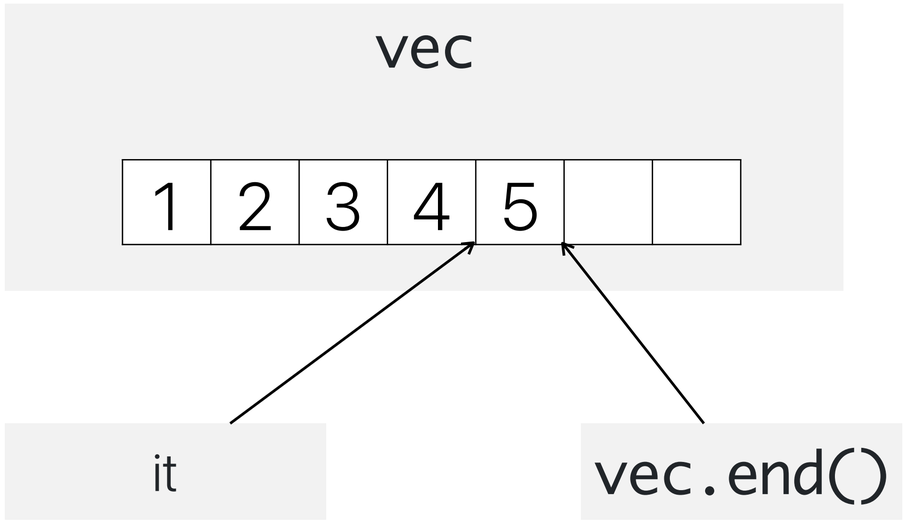
# Iteration – Iterators

```
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```



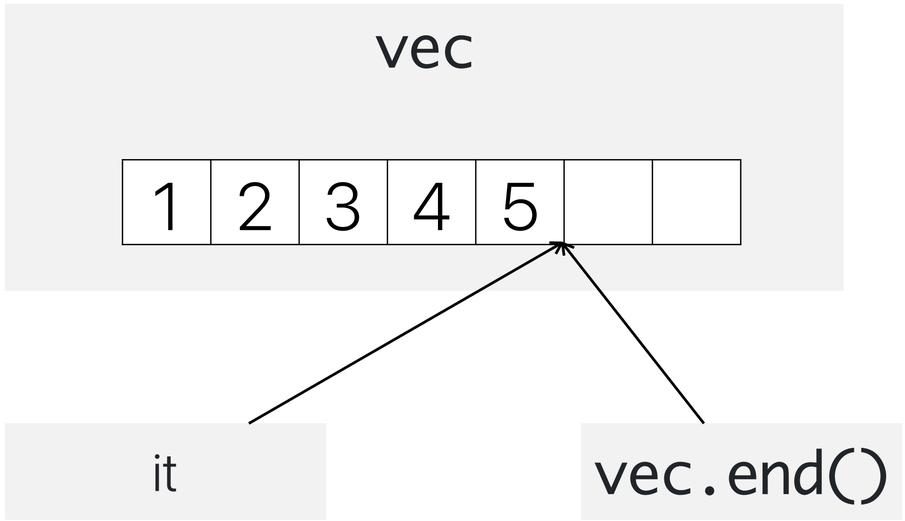
# Iteration – Iterators

```
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```



# Iteration – Iterators

```cpp
vector<int> vec = {1, 2, 3, 4, 5};

// Iterate using an iterator
for(auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << endl;
}
```



# Iteration – Iterators

- The `erase` method takes as an argument an iterator that points to the element to erase

```
vector<int> vec = {10, 20, 30, 40, 50};

// Delete the element at the third position
vec.erase(vec.begin() + 2);
```

  - `vec.begin()` returns an iterator that points to the first element
  - Not all containers support the + operator for their iterators

# Finding Elements

- Use the `std::find` function to search for an element in a container
- `find` returns an iterator that points to the found element if it exists, and one past the last element otherwise
- `std::distance` can be used to calculate the offset of the found element

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::find

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Find the first occurrence of 3 in the vector
    auto it = std::find(vec.begin(), vec.end(), 3);

    if (it != vec.end()) {
        std::cout << "Found 3 at index: " <<
std::distance(vec.begin(), it) << std::endl;
    } else {
        std::cout << "Element 3 not found." << std::endl;
    }

    return 0;
}
```

# Accumulating Elements

- Use the `std::accumulate` function to compute the sum of the elements of a container
- `accumulate` returns the final accumulated result
- For other types of accumulation (e.g., product, sum of squares, etc.), you can pass a custom operation function as an additional argument

```cpp
#include <iostream>
#include <vector>
#include <numeric> // for std::accumulate

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Sum all elements in the vector
    int sum = std::accumulate(vec.begin(), vec.end(), 0);

    std::cout << "Sum: " << sum << std::endl;
    // Output: Sum: 15

    return 0;
}
```

# Lecture Summary

- Output formatting
- File I/O streams
- Containers
  - `vector`
- Iterations
  - Range-based for loops
  - Iterators
- Useful algorithm functions for containers