Class 05

More on STL

fmt

Outline

- More on std::vector
- std::array
- STL algorithms
- fmt

std::vector

- The vector container is a STL container that provides the following:
  - Every element is adjacent in memory
  - It is resizable
  - The amortized complexity to insert something into the vector is O(1)
    - This is not complex at all!
  - It can manage only a single type at a time.
    - e.g. a vector cannot contain doubles, chars, and floats simultaneously
  - Size and capacity are differentiated:
    - size – how many elements are in the vector currently
    - capacity – how many elements can fit in the vector before it is full

std::vector

- There are multiple ways to create vector:

```
auto data1 = std::vector<T>{};           // empty
auto data2 = std::vector<T>{ ... };      // prepopulated
auto data3 = data1;                      // copy a vector
auto data4 = std::vector<T> data4(N);    // creates N default elements
auto data5 = std::vector<T> data5(N, V); // creates N elements equal to V
```

- Here T is a ***template***. It is a placeholder for the type that the vector manages. N is an integer representing size, and V is some value of type T.

- We can create vectors that are empty, pre-filled with data, copied from another vector, set to a specific *size*, or set to a specific size with all elements equal to a specific value.

std::vector::push_back

- When adding a new element to a vector that is already filled to its capacity (e.g. using push_back), the vector's content must be moved in memory to a location large enough to contain the old content along with the new content. This is called ***reallocation***.

- C++ will find a block of memory *twice as large, not just 1 larger,* to guarantee fewer reallocations over time. This is *always* done when a vector is automatically resized.

std::vector::reserve

- We can also tell C++ to reserve memory for a vector given some number of elements.
  - This is useful when you know how many elements you will need, but do not know what those elements are yet.
  - We use the *reserve* method to request enough memory for the specified number of elements. ***This updates the vector's capacity!***

```
auto data = std::vector<double>{};
data.reserve(100);
```

  - The reserve method will find a block of memory large enough for the specified number of elements, moving any elements currently in the vector.
    - If the number of elements specified is less than the capacity of the vector, nothing happens.
    - You are still limited by your machine's hardware!

std::vector::reserve

- We can check the capacity of a vector using the *capacity* method:

```
auto data = std::vector<double>{};
data.reserve(100);
std::cout << data.capacity() << std::endl;  // prints 100
```

- We can even *shrink* the vector's capacity to match the size:

```
auto data = std::vector<double>{1, 2, 3};   // capacity = 3, size = 3
data.reserve(100);                          // capacity = 100, size = 3
data.shrink_to_fit();                       // capacity = 3, size = 3
std::cout << data.capacity() << std::endl;  // prints 3
```

std::vector

- What is the difference between these two vectors?

```
auto data_1 = std::vector<double>{};
data.reserve(100);


auto data_2 = std::vector<double>(100);
```

std::vector

- What is the difference between these two vectors?

```
auto data_1 = std::vector<double>{};
data.reserve(100);


auto data_2 = std::vector<double>(100);
```

- While both vectors have a capacity of 100, the first has a size of 0 and the second has a size of 100!

- The second method here creates a vector with default elements! *Note the use of parentheses, and not curly-braces*

std::vector

- When should we use one method over another?
  - Your go-to should be an empty vector, using reserve to preallocate memory.
  - If you plan on updating data from a baseline, then construct with default values.
- If we are going to *generate* data, then it is likely more appropriate to just create an empty vector and preallocate with reserve.
  - e.g. we want to generate a ballistic trajectory; we know *approximately* how many points we will need but do not know what that data will ultimately be.
- If we are going to create a set of data that we will transform and update, then it is likely more appropriate to create the vector with default values.
  - e.g. we want to implement Conway's Game of Life; we know how many points we will want *and* defaulting all cells to 0 is relevant.

Range-Based Loop

- Whenever we want to loop over some container (like a vector), we typically would iterate across the *positions* of the elements, and then get the element from its position (e.g. something like *data[i]*, where *i* is the position).

- Range-Based Loops provide a cleaner way of iterating over containers where instead of getting the positions and getting the elements from those positions, we get the element directly. They have a new syntax that we can use:

```
for (auto element_name: my_collection)
{
    // do stuff with element_name, instead of my_collection[i]
}
```

Range-Based Loop

- Example using traditional loop

```
auto data = std::vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
for (auto i = size_t{0}; i < data.size(); ++i)
{
    data[i] = data[i] * 2;
}
```

Range-Based Loop

- Example using a range-based for-loop

```
auto data = std::vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
for (auto &element : data)
{
    element *= 2;
}
```

Range-Based Loop

- Example using a range-based for-loop

```
auto data = std::vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
for (auto &element : data)
{
    element *= 2;
}
```

- Note the ampersand (&) in the code snippet above! This is called a *reference* is necessary if you want to mutate the elements in the vector using a range-based loop.

- More on this in a later class!

How Will We Use Vectors?

- Much of what we will do in this course is generate sequences of data, usually time series data.
- Data will be buffered using vectors; this data can then be sent to other components as one large piece of data. Can be sent to:
  - algorithms (generate statistics, sorting, slicing, etc.)
  - visualizers
- Many tools deal with multidimensional data by representing the data as a series of x values, series of y values, series of z values, etc. rather than a series of points.
  - e.g. the points (1, 10), (2, 20), (3, 30) become the series (1, 2, 3) and (10, 20, 30)

STL Containers other than std::vector

- There are other containers in C++ that we need to mention, though we will use these a little more sparingly for various reasons.
  - std::array
  - std::list
  - std::deque
  - std::forward_list

std::array

- This is really the only other container we will care about.
- The primary differences between it and a vector are:
  - arrays are **not resizable**
  - arrays **size must be known at compile time**
- These two requirements allow arrays to outperform vector is every common use case.
- Vectors being resizable and dynamic are huge benefits, but in the scenarios where those features are not required, std::array is much better.

std::array

- To create an array, we first need to include its header:

```
#include <array>
```

- Then we can create one is a very similar way as a vector

```
auto data1 = std::array<int, 3>{};            // create array of 3 integer, all 0
auto data2 = std::array<int, 3>{10, 20, 30};  // create array of 3 integers, 10, 20, and 30
auto data3 = std::array{10, 20, 30};          // deduce type and size, not recommended
```

STL Algorithms

- The C++ STL as mentioned is packed with many algorithms implemented for you.
  - An algorithm is a set of instructions to carry out a task, whereas a function *implements* the instructions of the algorithm.
  - Many of these algorithms work with STL containers!
- Algorithms are generally found in one of the following headers in C++:
  - #include <algorithm>
  - #include <cmath>
  - #include <functional>
  - #include <numeric>
- For now, we are going to focus on algorithms meant for STL containers.

STL Algorithms

- The STL algorithm architecture for containers is unintuitive. Coders new to them expect to be able to do things like:

```
auto data = std::vector<int>{5, 7, 2, 9, 1};
std::sort(data);   // this does not work =[
```

- Unfortunately, they do not work like this. We must use special functions to retrieve the *beginning* and *end* of the container.

```
auto data = std::vector<int>{5, 7, 2, 9, 1};
std::sort(data.begin(), data.end());   // this works!
```

- The algorithms are designed to allow them to be applied to slices of containers, rather than the entire container all the time.

STL Algorithms - Ranges

- About half of the algorithms in C++ (soon to be more!) have *ranges* variants that are much more intuitive to use. For any algorithm found in the *algorithm* header we can use a better version of the algorithm!

```
auto data = std::vector<int>{5, 7, 2, 9, 1};
std::ranges::sort(data);
```

- Sadly in C++20 ranges only applies to the algorithm header. STL algorithms, namely found in the numeric header are missing this feature because the C++ standards committee is horribly slow.

- When possible we will use ranges over the old-style algorithms.

STL Algorithms

- STL algorithms always work in-place
- This means that when you apply an algorithm to some container, *a new container is never created*.
- **For example, the following is invalid code:**

```
auto data = std::vector<int>{5, 7, 2, 9, 1};
auto data2 = std::ranges::sort(data);  // Compiler error!
```

STL Algorithms Lists

- https://en.cppreference.com/w/cpp/algorithm
- https://en.cppreference.com/w/cpp/numeric

sort

- `#include <algorithm>`
- Sorts a container by placing the elements in *ascending order*.

```
auto data1 = std::vector<int>{5, 67, 2, 3, 7};
std::ranges::sort(data1);  // 2, 3, 5, 7, 67
```

- We can also sort using a predicate:

```
std::ranges::sort(data1, std::greater{});  // 67, 7, 5, 3, 2
```

- Default sorting uses the ***less-than*** operator to achieve *ascending order*. By overriding it to use the ***greater-than*** operator we can reverse sort.

min_element/max_element

- `#include <algorithm>`
- Respectively gives of the maximum and minimum elements of the container.
- Returns an *iterator* that we need to *dereference with an asterisk*.

```
auto data1 = std::vector<int>{5, 67, 2, 3, 7};
auto min = *std::ranges::min_element(data1); // 2
auto max = *std::ranges::max_element(data1); // 67
```

accumulate

- `#include <numeric>`
- Computes the sum of the elements in the container.
- We need to give it a starting number (usually 0!), and we need to ensure that the starting number is the correct type!

```
auto data1 = std::vector<int>{5, 67, 2, 3, 7};
auto sum = std::accumulate(data1.begin(), data1.end(), 0);  // 84
```

- Note that we are not using ranges here. Algorithms from numeric use this legacy form where we need to use begin and end positions of the container.

transform

- `#include <algorithm>`
- Applies a function (transformation) to every element in the container.
- Need to tell the transform function where to place the transformed elements.

```
auto foo(int x) -> int {
    return x + 100;
}

auto data1 = std::vector<int>{5, 67, 2, 3, 7};
std::ranges::transform(data1, data1.begin(), foo); // 105, 167, 102, 103, 107
```

transform

- We can use transform's output parameter to tell C++ *where* to insert the transformed elements.

- This requires one of two things to be done:
  - Initialize the destination to the appropriate size (e.g. create a vector with default elements; reserve is not enough!!)
  - Use a *back inserter*. This is a little wacky to look at, but is preferable in many cases!

```
auto data1 = std::vector<int>{5, 67, 2, 3, 7};
auto data2 = std::vector<int>(data1.size());
auto data3 = std::vector<int>{};

std::ranges::transform(data1, data2.begin(), foo);
std::ranges::transform(data1, std::back_inserter(data3), foo);
```

fmtlib

- *fmtlib* is an open source project that brings Pythons string-formatting to C++. We will use this over cout for the rest of the semester.

- We can access fmt by including the following:

```
#include <fmt/format.h>
```

- It is much simpler and familiar to use, and we can even use it to print containers like vectors!

```
fmt::print("hello world!"); // prints "hello world"
fmt::print("hello world!\n"); // prints "hello world" followed by a newline
fmt::println("hello world!"); // prints "hello world" followed by a newline
```

fmtlib - parameters

- We can format strings with placeholders, and let C++ inject variables/expressions into those placeholders.

- Placeholders are specified as empty {} within the strings being formed.

```
fmt::println("{} + {} = {}", 1, 2, 1+2);

auto name = std::string{"Goku"};
fmt::println("Hello {}, how are you?", name);

auto data = std::vector<int>{1, 2, 3};
fmt::println("my vector: {}", fmt::join(data, ","));
```

fmtlib - formatting

- We can also provide special formatting for placeholders, indicating how to format the data being printed.

- Commonly we will use this to adjust the number of decimals printed with floats.

```
fmt::println("{:.4f}", std::numbers::pi);  // print pi with 4 decimals
fmt::println("{:0>8d}", 9);  // print 9 with 7 leading 0s
fmt::println("{:0>8d}", 9);  // print 9 with 7 trailing #s
```

- The full syntax specification can be found here:
  - https://fmt.dev/latest/syntax.html