Lecture 4 Sequence Containers

#CS106L

- **Finishing Up C++ Types**
- **Survey Results!**
- **Overview of STL**
- **Sequence Containers**
- **Container Adaptors**

---

uniform initialization:generic method(look over the slides for more details)

A `deque` (pronounced “deck”) is a double ended queue,it can do everything a vector can do. Unlike a vector, it is possible (and fast) to `push_front` and `pop_front`.

`vector` is the type of sequence that should be used by default... `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence. -- C++ ISO Standard (section 23.1.1.2)

Q1:When should I use a stringstream in c++?please give me some specific examples.

A stringstream in C++ is used to easily manipulate strings as if they were input/output streams.

Here are some specific examples of when you might use a stringstream:

1. Parsing a CSV (Comma Separated Values) file: You can use a stringstream to extract the values between commas and store them in variables for further processing.
2. Converting between numerical data types: You can use a stringstream to convert a string representation of a number (e.g "123") to an int or a

double.

3. Tokenizing a string: You can use a stringstream to extract individual words or tokens from a sentence or paragraph.
4. Extracting multiple fields from a string: You can use a stringstream to extract multiple fields of data from a string that is in a specific format, such as extracting the day, month, and year from a date string.
5. Extracting decimal values from a string: you can use a stringstream to extract decimal values from a string.

Q2:I don't quite understand this line of code.

```
        std::copy(vec.begin(), vec.end(), std::ostream_iterator<int>(cout, "\n"));
```

The line of code you provided is using the `std::copy` algorithm from the C++ Standard Template Library (STL) to copy elements from a vector of integers (`vec`) to an output stream (`cout`).

Here's a breakdown of the line of code:

- `vec.begin()` and `vec.end()` specify the range of elements in the vector that should be copied. `vec.begin()` refers to the first element in the vector and `vec.end()` refers to the element one past the last element in the vector.
- `std::ostream_iterator<int>(cout, "\n")` is an instance of the `std::ostream_iterator` template class, which is an iterator that writes to an output stream. In this case, the output stream is `cout`, and the second argument, "\n", specifies a newline character to be inserted after each element is written to the stream.

So the `std::copy` algorithm will copy all elements from the vector vec, and write them to the output stream `cout` with a newline character after each element.

Example:

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

        std::copy(vec.begin(), vec.end(), std::ostream_iterator<int>(cout, "\n"));
```

This will output: 1 2 3 4 5

It's useful when you need to output a large number of elements and you don't want to use a loop to do it.

Q3:Why doesn’t std::vector bounds check by default?

Remember the philosophy of C++.If you write your program correctly, bounds checking will just slow your code down.It's a trade-off between safety and performance.

`std::vector` is a container class in the C++ Standard Template Library (STL) that is designed to be efficient in terms of both time and space. One way it achieves this efficiency is by not performing bounds checking by default.

Performing bounds checking on every access to a vector element would require additional computational time, and it would also increase the memory footprint of the vector. This extra overhead would be unnecessary in many situations, especially when the programmer is certain that the accesses to the vector are always within its bounds.

Additionally, bounds checking can also make the code less readable because it would make the code longer and more complex.

However, C++11 introduced a new feature called "bounds checking iterators" that can be used to check the access to the elements of the vector.

It's important to note that not performing bounds checking by default doesn't mean that vectors are not safe to use, it just means that it's the responsibility of the programmer to ensure that accesses to the vector are within its bounds.

Q4:How to compare the time difference between vector and deque?I mean their "Timing Version".

To compare the time difference between `std::vector` and `std::deque` in C++, you can create a simple benchmark program that performs a specific operation (e.g. insertion, deletion, random access, etc.) on both containers and measures the time it takes to complete the operation. Here is an example of how you could compare the time it takes for insertion at the end of a container:

```
#include <vector>

#include <deque>
```

```
#include <chrono>



int main() {

    std::vector<int> vec;

    std::deque<int> deq;

    int n = 1000000;

    auto start = std::chrono::high_resolution_clock::now();



    // Insertion at the end of the vector

    for (int i = 0; i < n; i++) {

        vec.push_back(i);

    }



    auto end = std::chrono::high_resolution_clock::now();

    auto vec_time = std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count();



    start = std::chrono::high_resolution_clock::now();



    // Insertion at the end of the deque

    for (int i = 0; i < n; i++) {

        deq.push_back(i);

    }
```

```
    end = std::chrono::high_resolution_clock::now();

    auto deq_time = std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count();



    std::cout << "Vector insertion time: " << vec_time << " microseconds" <<
std::endl;

    std::cout << "Deque insertion time: " << deq_time << " microseconds" <<
std::endl;



    return 0;

}
```

This code will measure the time it takes to insert `n` elements at the end of a vector and a deque, and it will output the time in microseconds. You can adjust the value of `n` to a value that is appropriate for your use case and also you can repeat the test multiple times and take the average of all the time to get more accurate results.

It's important to note that the performance of containers depends on the specific operations you are performing, as well as the size of the container and the specific implementation of the STL on your platform. So it's always a good idea to test with different inputs and different operations and compare the results.

There are several tools that can be used to visualize the performance of different algorithms or data structures, such as `std::vector` and `std::deque`. One popular tool is the Google Performance Tools library, which includes a profiler called `pprof` that can be used to generate performance profiles of C++ programs. It can generate a variety of output formats, including text, HTML, and graphical representations of the performance data.

Another tool that can be used is the `gprof` tool, it is a standard Unix profiler. It can generate a flat profile of the program, which shows the total amount of time spent in each function, and a call graph, which shows the function call relationships between different functions.

You can also use other performance measurement tools like `valgrind` which can also be used to generate performance information for C++ programs, including memory usage and cache misses.

It's worth noting that these tools can be quite complex to set up and use, and may require some knowledge of performance analysis and computer architecture. And also the results obtained from these tools are not that intuitive as compared to images, But they are more accurate and precise.