Introduction to containers and arrays

Introduction to containers and arrays

Similarly, containers typically implement a significant subset of the following operations:

- Create a container (e.g. empty, with storage for some initial number of elements, from a list of values).
- Access to elements (e.g. get first element, get last element, get any element).
- Insert and remove elements.
- Get the number of elements in the container.

std::vector and the unsigned length and subscript problem

prefer std::size() for getting size of vector and arrays

Accessing array elements using the at() member function does runtime bounds checking

operator [] does no bound checking

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

int main()
{
    std::vector prime{ 2, 3, 5, 7, 11 };

    std::cout << prime.at(3); // print the value of element with index 3
    std::cout << prime.at(9); // invalid index (throws exception)

    return 0;
}
```

Passing a std::vector using a generic template or abbreviated function template

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

template <typename T>
void passByRef(const T& arr) // will accept any type of object that has an overloaded operator[]
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const std::vector<double>&)

    return 0;
}
```

or use auto C++ 20

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

void passByRef(const auto& arr) // abbreviated function template
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const std::vector<double>&)
```

```
    return 0;
}
```

Returning std::vector, and an introduction to move semantics

How move semantics is invoked

Normally, when an object is being initialized with or assigned an object of the same type, copy semantics will be used (assuming the copy isn't elided).

However, when all of the following are true, move semantics will be invoked instead:

- The type of the object supports move semantics.
- The initializer or object being assigned from is an rvalue (temporary) object.
- The move isn't elided.
  Here's the sad news: not that many types support move semantics. However, std::vector and std::string both do!

Range-based for loops and type deduction using the auto keyword

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    for (auto num : fibonacci) // compiler will deduce type of num to be `int`
        std::cout << num << ' ';

    std::cout << '\n';

    return 0;
}
```

Best practise: Use type deduction (auto) with range-based for loops to have the compiler deduce the type of the array element.

Avoid element copies using references

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

int main()
{
    using namespace std::literals; // for s suffix for std::string literals
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s }; //
std::vector<std::string>

    for (const auto& word : words) // word is now a const reference
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

Best practice: In range-based for loops, the element declaration should use a (const) reference whenever you would normally pass that element type by (const) reference.

Consider always using const auto& when you don’t want to work with copies of elements

---

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

int main()
{
    using namespace std::literals;
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s }; //
obvious we should update this

    for (auto word : words) // Probably not obvious we should update this
too
        std::cout << word << ' ';

    std::cout << '\n';
```

```
    return 0;
}
```

If using type deduction in a range-based for loop, consider always using const auto& unless you need to work with copies. This will ensure copies aren't made even if the element type is later changed.

count enumerator

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

namespace Students
{
    enum Names
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        // add future enumerators here
        max_students // 5
    };
}

int main()
{
    std::vector<int> testScores(Students::max_students); // Create a vector with 5 elements

    testScores[Students::stan] = 76; // we are now updating the test score belonging to stan

    std::cout << "The class has " << Students::max_students << " students\n";

    return 0;
}
```

std::vector resizing and capacity

A std::vector can be resized after instantiation by calling the resize() member function with the new desired length:

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

int main()
{
    std::vector v{ 0, 1, 2 }; // create vector with 3 elements
    std::cout << "The length is: " << v.size() << '\n';

    v.resize(5);              // resize to 5 elements
    std::cout << "The length is: " << v.size() << '\n';

    for (auto i : v)
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

- The length of a vector is how many elements are “in use”.
- The capacity of a vector is how many elements have been allocated in memory.