Vectors and Their Functions

Thomas Merkh
tmerkh55@gmail.com

Section 2, Week 8

May 19, 2020
1. Vectors

2. Vector Methods

3. Next Time
Last Time

Last time we introduced operator overloading and copy constructors. As a refresher:

- Operator overloading is a way of giving additional meanings to the common operators, i.e. +, -, <<, += etc., usually so that they can operate on user-defined classes.

- Copy constructors are constructors which allow us to create a new class instance as a direct copy of an already existing instance.

- It is normal to have several additional constructors defined beyond the default and copy constructors, depending on the functionality of your class.
Last time we introduced operator overloading and copy constructors. As a refresher:

- Operator overloading is a way of giving additional meanings to the common operators, i.e. +, -, <<, += etc., usually so that they can operate on user-defined classes.
- Copy constructors are constructors which allow us to create a new class instance as a direct copy of an already existing instance.
- It is normal to have several additional constructors defined beyond the default and copy constructors, depending on the functionality of your class.

Today, we will cover our new favorite class, vectors. Vectors are a part of the c++ standard library and can be included in your program using the #include <vector> statement.
1 Vectors

2 Vector Methods

3 Next Time
What Are Vectors?

A vector is an array-like container that can dynamically change in size. Vectors are not restricted to just holding numbers; there can be vectors of strings, vectors of bools, and even vectors of class objects.

- Vectors are similar to those in which you learned in mathematics, but are not exactly the same! They do not have the same functionality, such as taking dot products.
- Vectors store their data values using a contiguous locations in computer memory, so we can iteratively move in memory location to retrieve the data values.
- Vectors and arrays are similar. Arrays can not change size, vectors can change in size.
What Are Vectors?

Since vectors can change size and are contiguous in memory, occasionally adding an element to the vector can result in the entire array of data values moving in memory.

- This is expensive, so typically vectors allocate extra storage to accommodate for adding elements.
- This is always done automatically so no need to worry, but reallocations tend to happen in logarithmically growing intervals to optimize efficiency.
- This means that vectors actually tend to use far more memory than arrays, with the (worthy) trade-off of being able to change in size.

Last, vectors are really class templates, meaning that when instantiating a vector, the type of data values being held must be specified.
Consider the following code:

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

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    return 0;
}
```

What do you think is going on here?
Consider the following code:

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

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    return 0;
}
```

What do you think is going on here? Answer: We are initializing the vector of integers `g1`, and then adding the numbers 1, 2, 3, 4, 5 to `g1`. 
Seems simple enough, so then what do we do the rest of the time?

Discuss methods that are defined for vectors, such as

- Iterating over a vector
- Determining the capacity and size of a vector
- Modifying the elements of the vector

Additionally, there are at least 5 ways to initialize a vector itself, so we will list some of those.
1. Vectors

2. Vector Methods

3. Next Time
There are many different methods related to capacity, I have listed many of them:

- `size()` - Returns the number of elements in the vector
- `max_size()` - Returns the maximum number of elements that the vector can hold
- `capacity()` - Returns the size of the storage space currently allocated to the vector expressed as number of elements
- `resize(n)` - Resizes the container so that it contains ’n’ elements
- `empty()` - Returns whether the container is empty
Vector Capacity Example

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

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size();
    cout << " Capacity: " << g1.capacity();
    cout << " Max Size: " << g1.max_size();

    g1.resize(4);
    cout << " Size: " << g1.size();

    if (g1.empty() == false) {
        cout << " Vector is not empty";
    }
}
```
There are many different methods related to vector modification:

- `push_back()` - appends a new element to the end of a vector
- `pop_back()` - removes the last item from a vector
- `insert()` - inserts a new element before the element at the specified position
- `erase()` - removes elements of a vector from the specified position or range
- `clear()` - removes all the elements of the vector container
- `assign()` - assigns a new value to the vector element at the index specified
```
#include <vector>
using namespace std;

int main()
{
    vector<int> v;
    v.assign(5, 10); // {10,10,10,10,10}
    v.push_back(15); // {10,10,10,10,10,15}
    int n = v.size();
    cout << '\nThe last element is: ' << v[n - 1];

    v.pop_back();
    v.insert(v.begin(), 5); // inserts 5 at the beginning
    v.erase(v.begin()); // removes the first element
    v.clear();
    cout << '\nVector size after erase(): ' << v.size();
}
```
There are many different techniques for iterating over a vector:

- `begin()` - Returns an iterator pointing to the first element in the vector.
- `end()` - Returns an iterator pointing to the theoretical element that follows the last element in the vector.
- `rbegin()` - Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element.
- `cbegin()` - Returns a constant iterator pointing to the first element in the vector.

The next slide shows how these can be used.
```cpp
int main()
{
    vector<int> g1 = {1, 2, 3, 4, 5};

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i){
        cout << *i << " ";
    }
    // 1 2 3 4 5
    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i) {
        cout << *i << " ";
    }
    // 1 2 3 4 5
    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir) {
        cout << *ir << " ";
    }
    // 5 4 3 2 1
}
```
A common question is: What is the difference between cbegin and begin, or using a constant iterator and ordinary iterator?
A common question is: What is the difference between cbegin and begin, or using a constant iterator and ordinary iterator?

A constant iterator is one that can only be used to access elements of the vector, and cannot be used for modification. A non-const iterator can be used to modify the container.

If there is any concept to bring away from this slide, its that const types are commonly used for access, and non-const types are commonly used for modification.
In this class, we have not formally learned about arrays. Arrays are constant sized containers, and “the container” used in C programming. Most people in this class probably haven’t had to deal with these yet, but trust me, they’re not as user-friendly as vectors. Additionally to really use arrays, we would need to know a fair deal about pointers (which we haven’t formally discussed yet).
I figured I’d point out one error that commonly arises when initializing vectors. Consider:

```cpp
vector<int> a(10);
vector<int> b(0);
vector<int> c;
vector<int> d();
```

What is the error?
A Common Error

I figured I’d point out one error that commonly arises when initializing vectors. Consider:

```cpp
vector<int> a(10);
vector<int> b(0);
vector<int> c;
vector<int> d();
```

What is the error? The error is in the last statement, where `d` is being *called* as if its a function. This is not how to declare a vector.
Methods To Initialize Vectors

Although I have not strictly included sample code here, I can post it, there are several valid ways of initializing vectors. These are:

- Creating an empty vector and pushing back values one by one
- Copying another vector which is already in existence
- Initializing a vector from an array
- Initializing a vector from a bracketed list
- The list truly does go on...
1 Vectors

2 Vector Methods

3 Next Time
Next time we will be learning about arrays, which are the underlying structure used in vectors.