Ch11-Vectors

August 16, 2020

1 Vectors

1.1 Topics

- what is and why vectors
- how to use vectors
- various operations and methods on vectors
- applications and example codes using vectors
- sorting vectors

1.2 Vectors

- vector is a collection of values where each value is identified by a number (index)
- anything that can be done by C-array (Array chapter) can be done using vectors
    - unlike C-array, vector is an advanced type like C++ string
- vector is defined in the C++ Standard Template Library (STL)
    - vector is one of my containers library - https://en.cppreference.com/w/cpp/container
    - array, set, map, queue, stack, priority_queue are some other containers
- learning vector is similar to learning C++ string container
    - main difference is vector can store any type of element
    - learn all the operations provided by vector
        * what they are; what they do; how to use them...
    - apply it to solve problems
- vector and other containers provided in STL are templated, hence STL
    - the actual type that you're storing in those containers need to be specified
    - very similar to template struct types covered in **Structures** chapter
    - if you're interested to learn about all these STL containers and more, there are Jupyter Notebooks you can use: https://github.com/rambasnet/STL-Notebooks
- must include <vector> header to use vector type

1.3 Vector objects

- C++ vector is a type defned in **<vector>** header
- objects must be instantiated or declared to allocate memory before we can store data into them
- since vector uses template type, you must provide the actual type of the data
- syntax:

```
#include <vector>
```

```
vector<type> objectName;
```

```
[1]:  #include <iostream>
      #include <vector>
      #include <string>
      #include <sstream>
      #include <cassert>

      using namespace std;
```

```
[2]:  // declare empty vectors
      vector<string> names;
      vector<float> tests;
      vector<int> numbers;
```

```
[3]:  // let's see the contents
      names
```

```
[3]:  {}
```

```
[4]:  // declare and initialize vectors
      vector<string> words = {"i", "love", "c++", "vectors"};
      vector<float> prices = {1.99, 199, 2.99, 200.85, 45.71};
```

```
[5]:  // let's see the contents
      words
```

```
[5]:  { "i", "love", "c++", "vectors" }
```

```
[6]:  prices
```

```
[6]:  { 1.99000f, 199.000f, 2.99000f, 200.850f, 45.7100f }
```

1.4 Accessing elements

- mostly using index just like in C-array or string
- index starts from 0 and goes to 1 less than the vector size or length
- **at(index)** : access specified element with bounds checking
- **operator[index]** : access specified element by index
- **front( )** : access the first element
- **back( )** : access the last element
- *sounds familiar? same way as accessing characters in string objects*

[13]:
```
// access elements
// change i to I in words
words[0] = "I";
cout << words[1] << endl; // print 2nd word
cout << prices.at(3) << endl;
cout << prices.front() << endl;
cout << prices.back() << endl;
```

```
love
200.85
1.99
45.71
```

1.5 Capacity

- unlike C-array, vector provides member functions to work with the capacity of the vector objects
- the following are the commonly used methods
- **empty( )** : checks whether the container is empty; returns true if empty; false otherwise
- **size( )** : returns the number of elements or length of the vector
- **max_size( )** : returns the maximum possible number of elements that can be stored

[15]:
```
cout << boolalpha; // convert boolean to text true/false
cout << "is prices vector empty? " << prices.empty() << endl;
cout << "size of words: " << prices.size() << endl;
cout << "size of prices: " << prices.size() << endl;
cout << "max size of words: " << words.max_size() << endl;
cout << "max capacity of rectangles: " << rectangles.max_size() << endl;
```

```
is prices vector empty? false
size of words: 5
size of prices: 5
max size of words: 768614336404564650
max capacity of rectangles: 2305843009213693951
```

1.6 Modifiying vectors

- vectors once created can be modified using various member functions or methods

- some commonly used methods are:

- **clear( )** : clears the contents

- **push_back(element)** : adds an element to the end

- **pop_back( )** : removes the last element

- *Note: if C-array was used, programmers would be have to implement these functions*

[16]:
```
vector<int> age = {21, 34, 46, 48, 46};
```

```
[17]: // see the initial contents
      age
```

```
[17]: { 21, 34, 46, 48, 46 }
```

```
[18]: // let's clear age vector
      age.clear();
```

```
[19]: // is age cleared?
      age
```

```
[19]: {}
```

```
[20]: // double check!
      age.empty()
```

```
[20]: true
```

```
[21]: // let's add element into the empty age vector
      age.push_back(25);
```

```
[22]: age.push_back(39);
```

```
[23]: age.push_back(45.5); // can't correctly add double to int vector
```

```
input_line_38:2:16: warning: implicit conversion from
'double' to 'std::__1::vector<int, std::__1::allocator<int> >::value_type' (aka
'int') changes
      value from 45.5 to 45 [-Wliteral-conversion]
 age.push_back(45.5); // can't correctly add double to int vector
          ~~~~~~~~~ ^~~~
```

```
[24]: age
```

```
[24]: { 25, 39, 45 }
```

```
[25]: // let's see the last element
      age.back()
```

```
[25]: 45
```

```
[26]: // let's remove the last element
      age.pop_back();
```

```
[27]: // check if last element is gone
      age
```

```
[27]: { 25, 39 }
```

1.7 Traversing vectors

- similar to string and C-array, vectors can be accessed from the first to last element
- use loop and index or iterators

```
[32]: for(auto val: words)
          cout << val << "; ";
```

```
I; love; c++; vectors;
```

```
[33]: for(int i=0; i<words.size(); i++) {
          cout << words[i] << " is " << words[i].length() << " characters long." <<␣
      ↪endl;
      }
```

```
I is 1 characters long.
love is 4 characters long.
c++ is 3 characters long.
vectors is 7 characters long.
```

1.8 Iterators

- similar to string iterators, vector provides various iterators
- iterators are special pointers that let you manipulate vector
- several member function of vector uses iterator to do its operation
- let's revist the iterators we went over in string chapter
- **begin( )** - returns iterator to the first element
- **end( )** - returns iterator to the end (past the last element)
- **rbegin( )** - returns reverse iterator to the last element
- **rend( )** - returns a reverse iterator to the beginning (prior to the first element)

```
[37]: // let's use iterator to traverse vectors
      // very similar to using for loop with index
      for(auto iter = words.begin(); iter != words.end(); iter++)
          cout << *iter << "; "; // iter is a pointer; so must derefernce to access␣
      ↪value pointed to by iter
```

```
I; love; c++; vectors;
```

```
[38]: // let's reverse traverse
      for(auto iter = words.rbegin(); iter != words.rend(); iter++)
          cout << *iter << "; "; // iter is a pointer; so must derefernce to access␣
      ↪value pointed to by iter
```

```
vectors; c++; love; I;
```

1.9 Aggregagte operations

- some aggegate operators such as assignment (=) and comparison operators (>, ==, etc.) are overloaded and work out of the box on vector objects as a whole
- sorta! - depends on what type of vector it is and is there predfined ordering of values in that type!
- input, output (<<, >>) operators do not work on vector objects as a whole

[39]:
```cpp
// create words_copy vector with copy assignment
vector<string> words_copy = words; // deep copies words into words_copy
```

[42]:
```cpp
// string can be compared out of the box
if (words == words_copy)
    cout << "two vectors are equal!";
else
    cout << "two vectors are not equal!";
```

```
two vectors are equal!
```

1.10 Passing vectors to functions

- vector can be passed to functions by value or by reference
- unless required, it's always efficient to pass containers type such as vectors to function by reference
  - copying data can be costly (take a long time) depending on the amount of data vector has

[44]:
```cpp
// given a vector of values find and return average
float findAverage(const vector<int> & v) {
    float sum = 0;
    for (auto val: v)
        sum += val;
    return sum/v.size();
}
```

[45]:
```cpp
// let's see the values of age vector
age
```

[45]: { 25, 39 }

[46]:
```cpp
cout << "average age = " << findAverage(age);
```

```
average age = 32
```

[47]:
```cpp
// printVector function
template<class T>
void printVector(const vector<T>& v) {
```

6

```
        char comma[3] = {'\0', ' ', '\0'};
        cout << '[';
        for (const auto& e: v) {
            cout << comma << e;
            comma[0] = ',';
        }
        cout << "]\n";
    }
```

[48]: ```
printVector(words);
```

```
[I, love, c++, vectors]
```

[49]: ```
printVector(age)
```

```
[25, 39]
```

1.11 Returning vector from functions

- since vector supports (=) copy operator, returning vector from functions is possible
- since returned vector needs to be copied (which can be costly),
  - it's best practice to use pass-by reference to get the data/results out of functions

[14]: ```
// function that gets vector of integers
void getNumbers(vector<int> & numbers) {
    cout << "Enter as many whole numbers as you wish.\nEnter 'done' when done:
    ↪\n";
    int num;
    while(cin >> num) // cin will return false when it fails
        numbers.push_back(num);
}
```

[15]: ```
// create an empty vector
vector<int> my_numbers;
```

[16]: ```
getNumbers(my_numbers);
```

```
Enter as many whole numbers as you wish.
Enter 'done' when done:
1000
898
10
345
4232
end
```

[53]: ```
my_numbers
```

[53]: { 10, 99, 100, 345 }

1.12   Two-dimensional vector

- if we insert vectors as an element to a vector, we essentially get a 2-d vector
  - similar to 2-d array

```
[10]: // let's declare a 2-d vector of integers
      vector< vector<int> > matrix;
```

```
[11]: // add the first vector - first row
      matrix.push_back({1, 2, 3, 4});
```

```
[12]: matrix[0]
```

[12]: { 1, 2, 3, 4 }

```
[13]: // let's add an empty vector as the second element or 2nd row
      matrix.push_back(vector<int>());
```

```
[14]: // let's add elements to the 2nd vector or 2nd row
      matrix[1].push_back(5);
      matrix[1].push_back(6);
      matrix[1].push_back(7);
      matrix[1].push_back(8);
```

```
[15]: matrix[1]
```

[15]: { 5, 6, 7, 8 }

```
[16]: // access element of vector elements
      // first row, first column
      matrix[0][0]
```

[16]: 1

```
[17]: // 2nd row, fourth column
      matrix[1][3]
```

[17]: 8

1.13   Sort vector

- vector like array needs to sorted often to solve many problems
- let's use built-in sort function in algorithm library
  - way better than using bubble sort, or implementing anorther faster alogrithm such as quick sort

```
[2]:  #include <vector>
      #include <algorithm> // sort()
      #include <iterator> // begin() and end()
      #include <functional> // greater<>();
      #include <iostream>

      using namespace std;
```

```
[3]:  vector<int> some_values = { 100, 99, 85, 40, 1233, 1};
```

```
[7]:  // let's sort some_values
      sort(begin(some_values), end(some_values));
```

```
[8]:  some_values
```

```
[8]:  { 1, 40, 85, 99, 100, 1233 }
```

```
[18]:  // let's sort 1st row of matrix in reverse order
       matrix[0]
```

```
[18]:  { 1, 2, 3, 4 }
```

```
[19]:  // sort in increasing order
       sort(matrix[0].begin(), matrix[0].end());
```

```
[20]:  matrix[0]
```

```
[20]:  { 1, 2, 3, 4 }
```

```
[21]:  // sort in on-increasing order
       sort(matrix[0].begin(), matrix[0].end(), greater<int>());
```

```
[22]:  matrix[0]
```

```
[22]:  { 4, 3, 2, 1 }
```

1.14   Exercises

1.14.1   Sove all exercises listed in Array chapter using vector instead.

1. Write a function that splits a given text/string into a vector of individual words
   - each word is sequence of characters separated by a whitespace
   - write 3 test cases

```
[23]:  // Solution to Exercise 1
       void splitString(vector<string> &words, string text) {
           string word;
           stringstream ss(text);
```

```
        while (ss >> word) {
            words.push_back(word);
        }
    }
```

[24]:
```
void test_splitString() {
    vector<string> answer;
    splitString(answer, "word");
    vector<string> actual = {"word"};
    assert(answer == actual);
    answer.clear();
    splitString(answer, "two word");
    vector<string> actual1 = {"two, word"};
    assert(answer == actual1);
    answer.clear();
    splitString(answer, "A sentence with multiple words!");
    vector<string> actual2 = {"A", "sentence", "with", "multiple", "words!"};
    assert(answer == actual2);
    answer.clear();
    cerr << "all test cases is passed for splitString()\n";
}
```

[25]:
```
test_splitString();
```

```
all test cases is passed for splitString()
```

[26]:
```
vector<string> tokens;
```

[27]:
```
// not needed but just in case!
tokens.clear();
```

[28]:
```
splitString(tokens, "This is a long sentence so long that it's hard to␣
 ↪comprehend!");
```

[29]:
```
tokens
```

[29]:
```
{ "This", "is", "a", "long", "sentence", "so", "long", "that", "it's", "hard",
"to", "comprehend!" }
```

1.14.2 complete program can be found in demo_programs/Ch11/splitString.cpp

2. Airline Reservation System:
   - Write a C++ menu-driven CLI-based program that let's an airline company manage airline reservation on a single aircraft they own with the following requirements:
   - aircraft has 10 rows with 2 seat on each row
   - program proviedns menu option to display all the avaialable seats
   - program provides menu option to let user pick any available seat

- program provides menu option to creates total sales report
- program provides menu option to update price of any seat

1.15   Kattis problems

- problems that require to store large amount of data in sequential order in memory can use vector very effectively
- design your solutions in a way that it can be tested writing automated test cases

1. Dice Game - https://open.kattis.com/problems/dicegame
2. Falling Apart - https://open.kattis.com/problems/fallingapart
3. Height Ordering - https://open.kattis.com/problems/height
4. What does the fox say? - https://open.kattis.com/problems/whatdoesthefoxsay
5. Army Strength (Easy) - https://open.kattis.com/problems/armystrengtheasy
6. Army Strength (Hard) - https://open.kattis.com/problems/armystrengthhard
7. Black Friday - https://open.kattis.com/problems/blackfriday
8. Bacon, Eggs and Spam - https://open.kattis.com/problems/baconeggsandspam

1.15.1   sorting vectors with two keys

1. Roll Call - https://open.kattis.com/problems/rollcall
2. Cooking Water - https://open.kattis.com/problems/cookingwater

1.16   Summary

- this chapter covered C++ vector container STL
- vector is an easier alternative to C-array
- vector is and advanced type that you can create/instantiate bojects from
- the type of the data must be mentioned as a template parameter while declaring vectors
- provides many out-of the box common operations in the form of member functions or methods
- vector can be passed to functions; returning a large vector may not be effective due to copying of data
- probems and sample solutions

[ ]: