15. Vector Mutation, List Processing

CPSC 120: Introduction to Programming
Kevin A. Wortman ~ CSU Fullerton

Agenda

0. Announce
    a. Sign-in sheet
    b. Midterm 2: Wed Nov 1 (week 11)
1. Technical Q&A
2. Review: Vector
3. Vector Mutation Operations
4. List Processing Patterns

1. Technical Q&A

Technical Q&A

Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

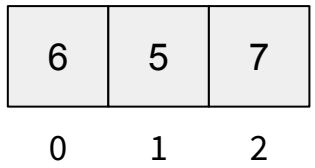
Reminder: write these questions in your notebook during lab

2. Review: Vector

Vector Layout

- **Contiguous:** elements at adjacent memory locations
- **Index:** locations numbered 0, 1, ..., *n*-1

```
std::vector<int> container{6, 5, 7};
```



Declaring a std::vector

statement:

std::vector<*data-type***>** *identifier* **{** *element ...* **};**

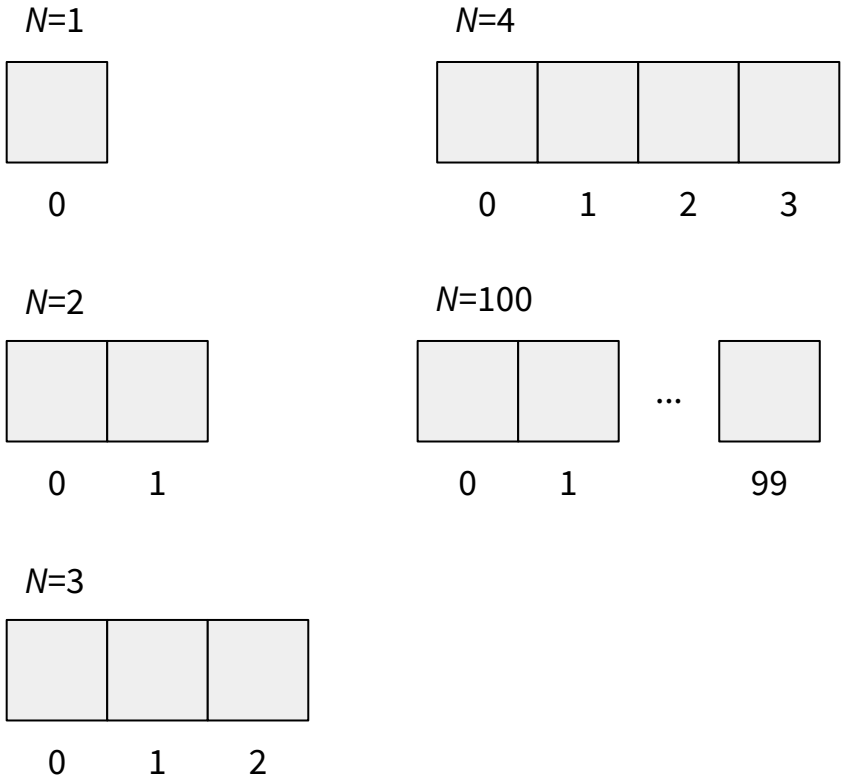
where

- *data-type* is the type of one element
- *identifier* is variable name
- *element...* are expressions of type *T*

```
std::vector<double> coords{ 1.0, 4.2 };

std::vector<int> phone{2, 7, 8, 1, 7, 1, 2};
```

Valid Indices

- **Index:** position of an element in a vector
- **Indices:** plural of index
- Let *N* = size of vector
- **First** index is 0
- **Last** index is *N* - 1



std::vector::at

- Access an element of a vector
- Member function
- Reference page: std::vector::at , observe
  - pos (index)
  - throws exception for invalid index
  - examples

std::vector::size

- Member function std::vector::size
- Returns the size of the vector
  - (number of elements)
- Needed when vector is filled at runtime
  - (command-line arguments next)

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

int main()
{
    std::vector<int> nums {1, 3, 5, 7};

    std::cout << "nums contains " << nums.size() << " elements.\n";
}
```

Empty std::vector

- **empty:** contains no elements
- size is zero
- no valid index
  - std::vector::at always throws exception
- Declare with
  - no elements between braces, **or**
  - omit braces entirely
- Classes are always initialized
  - no worry of uninitialized variable

```
// size 0
std::vector<int> scores{};

// size 0
std::vector<double> readings;
```

Syntax: For-Each Loop

statement:

for (*for-range-decl* **:** *container***)**
body-statement

container: expression for a container object

for-range-decl: elt-type elt-identifier

Semantics:

- *elt-type* must match base type of *container*
- for each element in *container*:
  - initialize new *elt-identifier*{ current element }
  - execute *body-statement*
  - *elt-identifier* destroyed

```
// prints -2-7-8-2-0-1-1
std::vector<int> digits{ 2, 7, 8, 2, 0,
  1, 1 };
for (int d : digits) {
   std::cout << "-" << d;
}
std::cout << "\n";

// prints Mon Tue Wed Thu Fri
std::vector<std::string> weekdays{"Mon",
"Tue", "Wed", "Thu", "Fri"};

for (std::string today : weekdays) {
   std::cout << today << " ";
}
std::cout << "\n";
```

3. Vector Mutation Operations

“Mutation”

- **Mutate** (v): change
- **Mutation** (n): act of changing something
- **Mutation operation** (n): function that changes a vector
- **Immutable variable**: cannot be changed; `const`

Push and Pop

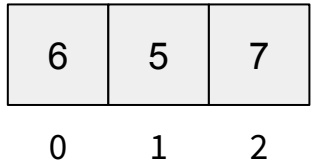
- **Push** (v): add an element to a container
- **Pop** (v): remove an element from a container
- Inspired by cafeteria dish dispenser

Image credit:
http://miseenplaceasia.com/dish-warmer-dispenser/

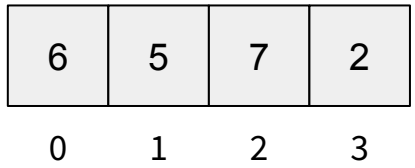
std::vector::push_back

- std::vector::push_back
- Adds one element at the back of the vector
  - Size increases by one
  - Certainly is not empty
- Why the back?
  - How people usually make lists
  - More efficient than front
  - See *CPSC 131 - Data Structures*

```
std::vector<int> values{6, 5, 7};
```



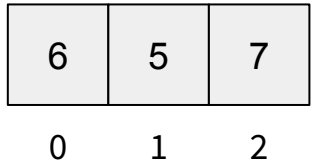

```
values.push_back(2);
```



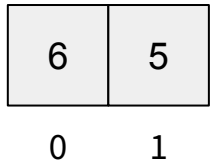
std::vector::pop_back

- `std::vector::pop_back`
- Removes the last element
- Size decreases by one
- Vector must be non-empty
  - Otherwise: undefined behavior (runtime error)
- Why the back?
  - (again)
  - More efficient than front
  - See *CPSC 131 - Data Structures*

```
std::vector<int> values{6, 5, 7};
```



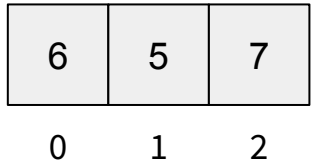

```
values.pop_back();
```



std::vector::clear

- std::vector::clear
- Erases all elements of vector
- Size becomes zero
  - Certainly is empty
- Useful to “recycle” a vector and fill it up again

```
std::vector<int> values{6, 5, 7};
```




```
values.clear();
```

(empty)

empty versus clear

- `std::vector::empty`: check whether vector is empty
  - No mutation
- `std::vector::clear`: make vector empty
  - Mutation
- Unfortunately similar, mnemonic device:
  - “Is it empty?” makes sense
  - “Is it clear?” doesn’t
  - So empty is the accessor, clear is the mutator

3. List Processing Patterns

Algorithm: Build a Vector

OUTPUT: a vector OUT of elements

1. Declare OUT as an empty vector
2. Loop for each element:
   a. Create a new element
   b. Add the element to the back of OUT (push_back)

```
std::vector<int> years; // 20th century years
for (int i = 1900; i < 2000; ++i) {
  years.push_back(i);
}

 std::vector<std::string> arguments{argv, argv + argc};

// just the inputs; skip the command name
std::vector<std::string> inputs;
for (int i = 1; i < arguments.size(); ++i) {
  inputs.push_back(arguments.at(i));
}
```

```
$ ./blackjack J 6 A

arguments: “./blackjack” “J” “6” “A”
inputs: “J” “6” “A”
```

Algorithm: Build Vector From File

(variation on **Build a Vector**)

INPUT: a filename containing data elements
OUTPUT: a vector OUT containing the elements from the file

1. Open input file (`ifstream`)
2. Declare OUT as an empty vector
3. while file is good:
   a. Read one data element
   b. if no I/O error: add element to back of OUT (`push_back`)

```
std::vector<std::string> words;
std::ifstream file("words.txt");
while (true) {
  std::string one_word;
  file >> one_word;
  if (!file) {
    break;
  }
  words.push_back(one_word);
}
```

Algorithm: Filter Vector

INPUT: a vector IN of elements that may match
OUTPUT: a vector OUT containing only the matching elements

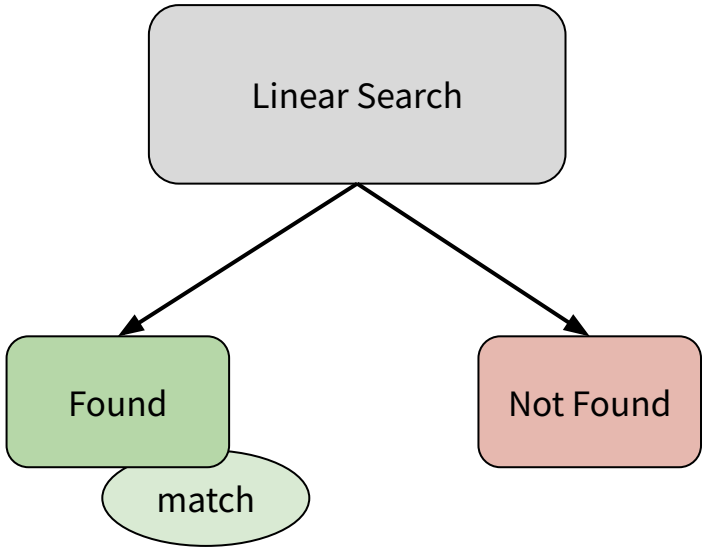
1. Declare OUT as an empty vector
2. For each element x of IN:
   a. if x is a match:
      i. Add x to back of OUT (push_back)

```
std::vector<std::string> arguments{argv, argv + argc};

std::vector<std::string> cards;
for (int i = 1; i < arguments.size(); ++i) {
  cards.push_back(arguments.at(i));
}

std::vector<std::string> aces;
for (std::string card : cards) {
  if (card == "A") {
    aces.push_back(card);
  }
}
```

```
$ ./blackjack A 3 K A

arguments: “./blackjack” “A” “3” “K” “A”
cards: “A” “3” “K” “A”
aces: “A” “A”
```

Review: Linear Search

- Linear search: algorithm for finding an element, which may not exist
  - Brute force password cracking
  - Ray tracing computer animation
- Check each element in order
- If the current one is what we want, **stop** (**success/found**)
  - Stop with `break`
- Get to the end: **failure/not-found**
  - Match does not exist
- Two outcomes



Review: Pattern: Linear Search

- `found`: `bool` variable remembers success/failure
- `match`: copy of matching element
  - only valid when `found` is `true`

```
bool found{false};
elt-type match{default-value};
for (elt-type element : container) {
 if (elt-is-match-condition) {
  found = true;
  match = element;
  break;
 }
}
// use found and match
```

```
std::vector<int> values{5, 11, -2, 8};

// find a negative value
bool found{false};
int match{0};
for (int value : values) {
  if (value < 0) {
      found = true;
      match = value;
      break;
  }
}
if (found) {
  std::cout << match << " is a negative value\n";
} else {
  std::cout << "there are no negative values\n";
}
```

Algorithm: Unique Elements

INPUT: a vector IN, which may contain duplicates
OUTPUT: a vector OUT, where each element of IN appears only once

1. Declare OUT as an empty vector
2. for each element x of IN:
    a. Linear search for x in OUT
    b. If x is not found in OUT:
        i. Add x to OUT (`push_back`)

(There is a more efficient algorithm, see *CPSC 335 Algorithm Engineering*)

Example: Unique Elements

```
std::vector<std::string> signins;
// build vector...

std::vector<std::string> unique_names;
for (std::string signin : signins) {
  bool already_in_unique_names{false};
  for (std::string name : unique_names) {
    if (signin == name) {
      already_in_unique_names = true;
      break;
    }
  }
  if (!already_in_unique_names) {
    unique_names.push_back(signin);
  }
}
```

signins: “alice” “bob” “carlos” “alice” “bob”

unique_names: “alice” “bob” “carlos”

Algorithm: Common Elements

INPUT: vector L and vector R

OUTPUT: vector OUT contains every element that is in both L and R

1. Declare OUT as an empty vector
2. For each element x of L:
    a. Linear search for x in R
    b. If x is found in R:
        i. Add x to OUT (`push_back`)

(There is a more efficient algorithm, see *CPSC 335 Algorithm Engineering*)

Example: Common Elements

```
std::vector<int> years; // 20th century years
for (int i = 1900; i < 2000; ++i) {
  years.push_back(i);
}

std::vector<int> earthquake_years; // load from file...

std::vector<int> common_years;
for (int x : years) {
  bool found{false};
  for (int y : earthquake_years) {
    if (x == y) {
      found = true;
      break;
    }
  }
  if (found) {
    common_years.push_back(x);
  }
}
```

```
years: 1900, 1901, ..., 1999

earthquake_years: 1857, 1872, 1906,
1923, 1980, 1992, 2019

common_years: 1906, 1993, 1980, 1992
```

Algorithm: Transform

INPUT: a vector IN of elements
OUTPUT: a vector OUT of elements, containing a transformed version of each element of IN

1. Declare OUT as an empty vector
2. For each element x in IN:
    a. t = *transform* x
    b. Add t to back of OUT (`push_back`)

Example: Transform

```
std::vector<std::string> arguments{argv, argv + argc};

std::vector<std::string> cards;
for (int i = 1; i < arguments.size(); ++i) {
  cards.push_back(arguments.at(i));
}

std::vector<int> points;
for (std::string card : cards) {
  int t{0};
  if (card == "A") {
  } else if ((card == "J") || (card == "Q") ||
             (card == "K")) {
    t = 10;
  } else {
    t = std::stoi(card);
  }
  points.push_back(t);
}
```

```
$ ./blackjack “J” “7” “A” “K” “A”

arguments: “./blackjack” “J” “7” “A” “K”
“A”

cards: “J” “7” “A” “K” “A”

points: 10 7 1 10 1
```