# CS 106B, Lecture 3
# Vector and Grid

reading:

*Programming Abstractions in C++*, Chapter 4-5

# Plan for Today

- Learn about two new "ADTs" or collections
  - Vector: a data structure for representing lists
  - Grid: a data structure ideal for representing two dimensional information

# Abstract Data Types (ADTs)

- **Collection**: an object that stores data;  a.k.a. "data structure"
  - the objects stored are called **elements**.
- Collections are also called ADTs: a data type described by its external functionality. Defined by its behavior, not implementation
- **Abstraction**
  - Public interface is clean, easy to use
  - Hide private messy implementation details
- First we are going to use these ADTs, then we will implement them later.

# STL vs. Stanford

- **collection**: an object that stores data;  a.k.a. "data structure"
  - the objects stored are called **elements**.
  - Also known as "ADTs" – abstract data types
- **Standard Template Library (STL)**:
  C++ built in standard library of collections.
  - `vector`, `map`, `list`, …
  - Powerful but somewhat hard to use for new coders (messy syntax) – take 106L!


- **Stanford C++ library (SPL)**:
  Custom library of collections made for use in CS 106B/X.
  - **`V`**`ector`, **`G`**`rid`, **`S`**`tack`, **`Q`**`ueue`, **`S`**`et`, **`M`**`ap`, …
  - Similar to STL, but simpler interface and error messages.
  - Note the capitalized first letter

# Plan for Today

- Learn about two new "ADTs" or collections
  - Vector: a data structure for representing lists
  - Grid: a data structure ideal for representing two dimensional information

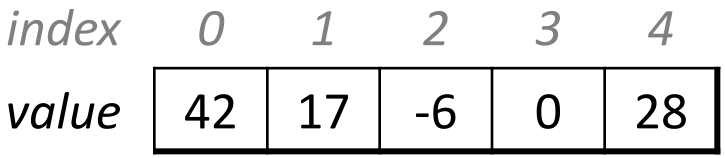
# Vectors (Lists)

```
#include "vector.h"
```

- **vector** (aka **list**): a collection of elements with 0-based **indexes**
  - like a dynamically-resizing array (Java `ArrayList` or Python `list`)
  - Include the type of elements in the <> brackets

```
// initialize a vector containing 5 integers
//          index   0   1   2   3   4
Vector<int> nums {42, 17, -6,  0, 28};

Vector<string> names;          // {}
names.add("Dog");            // {"Dog"}
names.add("Cat");          // {"Dog", "Cat"}
names.insert(0, "Bug");   // {"Bug", "Dog", "Cat"}
```

# Why not arrays?




- Arrays have fixed **size** and cannot be easily resized.
  - In C++, an array doesn't even *know* its size. (no `.length` field)
- C++ lets you index out of the array **bounds** (garbage memory) *without* necessarily crashing or warning.
- An array does not support many **operations** that you'd want:
  - inserting/deleting elements into the front/middle/back of the array, reversing, sorting the elements, searching for a given value ...

# Vector members

| | |
|---|---|
| `v.add(value);` or<br>`v += value;` or<br>`v += v1, v2, ..., vN;` | appends value(s) at end of vector |
| `v.clear();` | removes all elements |
| `v[i]` or `v.get(i)` | returns the value at given index |
| `v.insert(i, value);` | inserts given value just before the given index, shifting subsequent values to the right |
| `v.isEmpty()` | returns `true` if the vector contains no elements |
| `v.remove(i);` | removes/returns value at given index, shifting subsequent values to the left |
| `v[i] = value;` or<br>`v.set(i, value);` | replaces value at given index |
| `v.subList(start, length)` | returns new vector of sub-range of indexes |
| `v.size()` | returns the number of elements in vector |
| `v.toString()` | returns a string representation of the vector such as `"{3, 42, -7, 15}"` |
| `ostr << v` | prints v to given output stream (e.g. `cout << v`) |

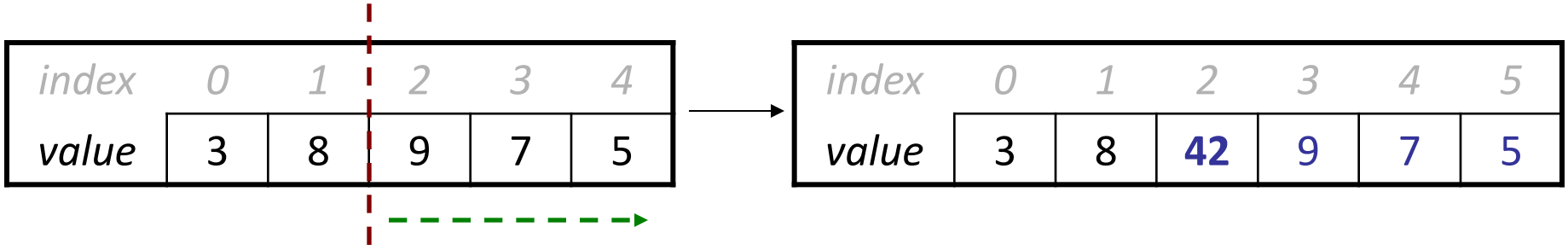
# Iterating over a vector

```
Vector<string> names {"Rafi", "Giorgi", "Sue"};

// Prints off each element on its own line
for (int i = 0; i < names.size(); i++) {
    cout << names[i] << endl;
}
// Same thing as above but backwards
for (int i = names.size() - 1; i >= 0; i--) {
    cout << names[i] << endl;
}

// "for-each" loop
for (string name : names) {
    cout << name << endl;
}
// Can't edit (insert/delete) in for-each loop
```

# Vector insert/remove

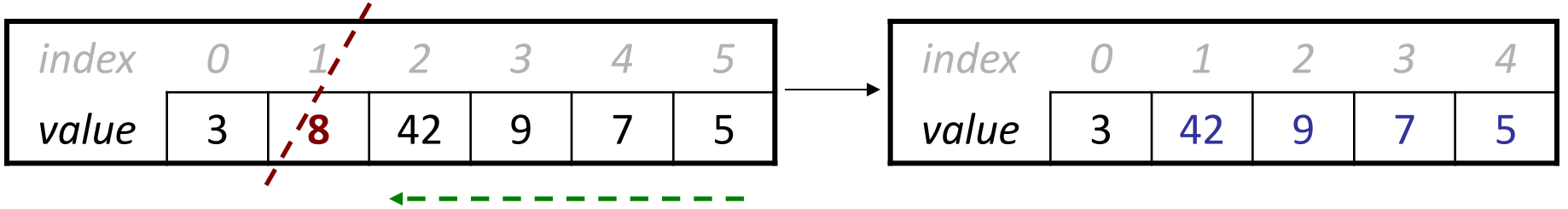
```
v.insert(2, 42);
```

- shift elements right to make room for the new element




```
v.remove(1);
```

- shift elements left to cover the space left by the removed element




*(These operations are slower the more elements they need to shift.)*

# Vector Exercises

- Write a function **countInRange** that accepts a vector<int>, a min, and a max. It returns the number of values in the vector that fall within the range inclusive.

  So if vec contained {0, 5, -21, -4, 7} and min = 2 and max = 12, the function would return 2.

- Write a function **removeAll** that accepts a vector of strings, and a target string. It removes any strings in the vector that equal the target string.

  So if vec contained {“Youre”, “a”, “hairy”,“wizard”, “hairy”} and target = “hairy”, vec should equal {“Youre”, “a”, “wizard”} .

# Exercise Solutions

```
int countInRange(const Vector<int>& vec, int min, int max) {
    int count = 0;
    for (int element : vec) {
        if (element >= min && element <= max) {
            count++;
        }
    }
    return count;
}
```

```
void removeAll(Vector<String>& vec, String target) {
    for (int i = vec.length() - 1; i >= 0; i--) {
        if (vec[i] == target) {
            vec.remove(i);
        }
    }
}
```

# Announcements

- Exam Conflicts
  - Academic or university athletic conflicts will be handled on a case by case basis.
  - Family travel is not an acceptable reason to miss an exam.

- Getting started with C++
  - Kate posted some helpful resources on Piazza and under the handouts dropdown menu of the website.

# Plan for Today

- Learn about two new "ADTs" or collections
  - Vector: a data structure for representing lists
  - Grid: a data structure ideal for representing two dimensional information

# Grid

```
#include "grid.h"
```

- Like a 2D array, but more powerful
- Must specify element type in < > (a ***template*** or a *type parameter*)

```
Grid<int> matrix(3, 4);
matrix[0][0] = 75;
...

// or specify elements in {}
Grid<int> matrix = {
    {75, 61, 83, 71},
    {94, 89, 98, 100},
    {63, 54, 51, 49}
};
```

| row \ column | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 75 | 61 | 83 | 71 |
| 1 | 94 | 89 | 98 | 100 |
| 2 | 63 | 54 | 51 | 49 |

# Grid members*

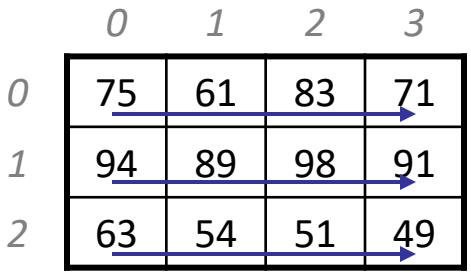
| | |
|---|---|
| `Grid<type> name(r, c);`<br>`Grid<type> name;` | create grid with given number of rows/cols; empty 0x0 grid if omitted |
| `g[r][c]` or `g.get(r, c)` | returns value at given row/col |
| `g.fill(value);` | set every cell to store the given value |
| `g.inBounds(r, c)` | returns `true` if given position is in the grid |
| `g.numCols()` or `g.width()` | returns number of columns |
| `g.numRows()` or `g.height()` | returns number of rows |
| `g.resize(nRows, nCols);` | resizes grid to new size, discarding old contents |
| `g[r][c] = value;` or<br>`g.set(r, c, value);` | stores value at given row/col |
| `g.toString()` | returns a string representation of the grid such as `"{{3, 42}, {-7, 1}, {5, 19}}"` |
| `ostr << g` | prints, e.g. `{{3, 42}, {-7, 1}, {5, 19}}` |

** (a partial list; see [http://stanford.edu/~stepp/cppdoc/](http://stanford.edu/~stepp/cppdoc/) )*

# Looping over a grid

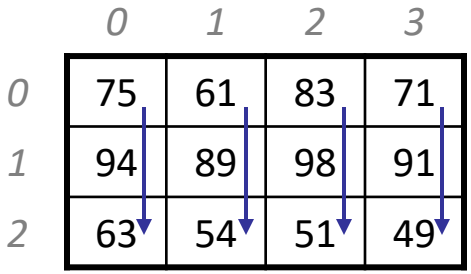
- Row-major order:

```
for (int r = 0; r < grid.numRows(); r++) {
    for (int c = 0; c < grid.numCols(); c++) {
        do something with grid[r][c];
    }
}

// "for-each" loop  (also row-major)
for (int value : grid) {
    do something with value;
}
```




- Column-major order:

```
for (int c = 0; c < grid.numCols(); c++) {
    for (int r = 0; r < grid.numRows(); r++) {
        do something with grid[r][c];
    }
}
```



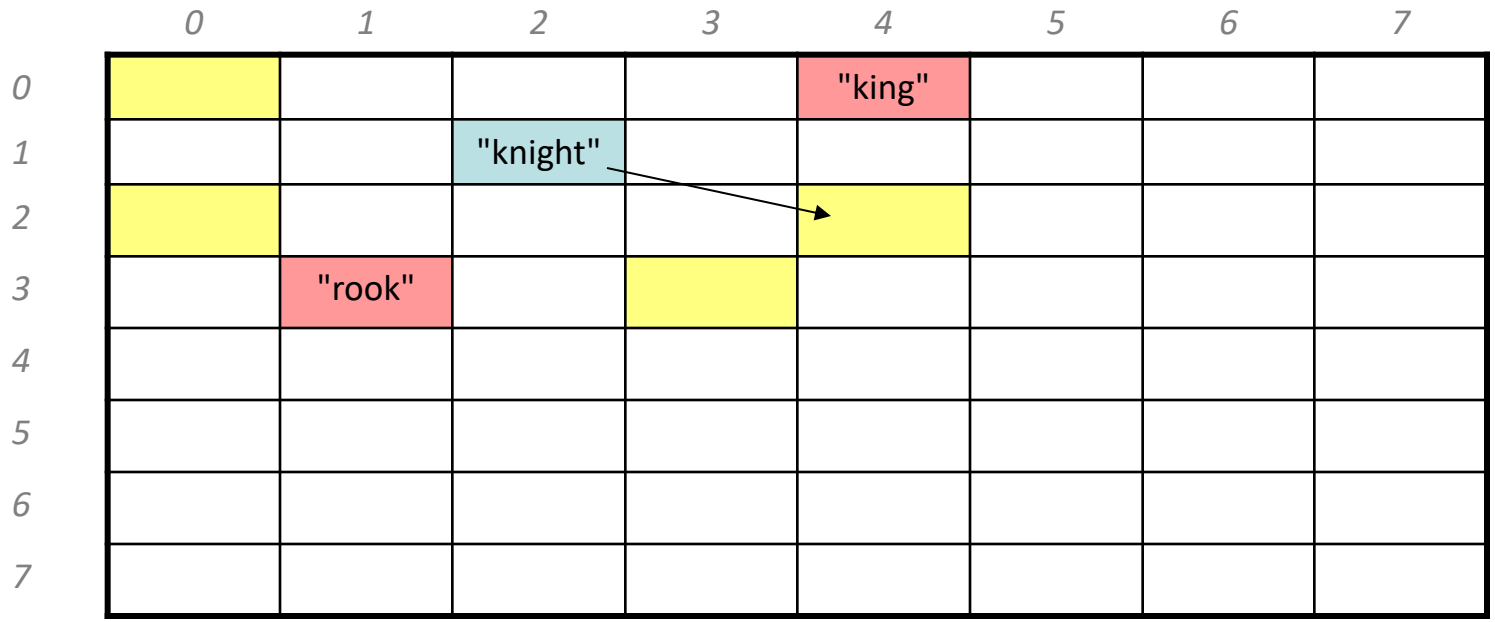
# Grid as parameter

- When a `Grid` is passed by value, C++ makes a copy of its contents.
  - Copying is slow; you should **pass by reference** with **&**
  - If the code won't modify the grid, also pass it as **`const`**

```
// Which one is best?
A) int computeSum(Grid<int> g) {
B) int computeSum(Grid<int>& g) {
C) int computeSum(const Grid<int> g) {
D) int computeSum(const Grid<int>& g) {

// Which one is best?
A) void invert(Grid<double> matrix) {
B) void invert(Grid<double>& matrix) {
C) void invert(const Grid<double> matrix) {
D) void invert(const Grid<double>& matrix) {
```

# Grid exercise

- Write a function **`knightCanMove`** that accepts a grid and two row/column pairs (*r1*, *c1*), (*r2*, *c2*) as parameters, and returns `true` if there is a knight at chess board square (*r1*, *c1*) that can legally move to empty square (*r2*, *c2*).
  - Recall that a knight makes an "L" shaped move, going 2 squares in one dimension and 1 square in the other.
  - `knightCanMove(board, 1, 2, 2, 4)` returns `true`



# Grid exercise solution

```
bool knightCanMove(Grid<string>& board, int r1, int c1,
                                        int r2, int c2) {
    if (!board.inBounds(r1, c1) || !board.inBounds(r2, c2)) {
        return false;
    }
    if (board[r1][c1] != "knight" || board[r2][c2] != "") {
        return false;
    }
    int dr = abs(r1 - r2);
    int dc = abs(c1 - c2);
    if (!((dr == 1 && dc == 2) || (dr == 2 && dc == 1))) {
        return false;
    }
    return true;
}
```

# Grid solution 2

```
bool knightCanMove(Grid<string>& board, int r1, int c1,
                                        int r2, int c2) {
    int dr = abs(r1 - r2), dc = abs(c1 - c2);
    return board.inBounds(r1, c1) && board.inBounds(r2, c2)
           && board[r1][c1] == "knight" && board[r2][c2] == ""
           && ((dr == 1 && dc == 2) || (dr == 2 && dc == 1));
}
```

# Look Ahead

- Assignment 0 due Thursday
  - If you need help with Qt stop by LaIR tonight at 8PM!

- Sections start today! Should have received an email from cs198@cs.stanford.edu
  - You can switch your section or sign up late at cs198.stanford.edu
  - Email **Kate** if you were assigned a different section than your partner