PIC 10B Week 1 (Thurs)

Alex Tong Lin

January 11, 2017
1. Vectors Review

2. Pointers 101
Chapter 6: Vectors and Arrays

Chapter Goals:

► To become familiar with using vectors to collect objects.
► To be able to access vector elements and resize vectors.
► To learn how to use one- and two-dimensional arrays.
► To learn about common array algorithms.
Section 6.1: Using Vectors to Collect Data Items

▶ Suppose you have 10 items of data of the same type, e.g. a list of 10 integers. It would be a headache to create 10 different variables: n1, n2, ..., n10.

▶ This is where vectors come in: Vectors allow you to organize a list of data into a single variable.
Section 6.1: Using Vectors to Collect Data Items
How to create a vector variable

- For example, if we wanted to create a list of salaries of 10 employees, we would use a vector. The notation for this is,

```cpp
#include <vector>
std::vector<double> salaries(10);
```

- To access a value in a vector, we use the notation `salaries[i]`. For example, if we wanted to set the 5th employee’s salary to 35,000, we use,

```
salaries[4] = 35,000
```

Note the indexing starts at 0 and ends at 9.

- If you don’t specify a size, you create an empty vector for which you can resize later.
Section 6.1: Using Vectors to Collect Data Items

Visual representation of salaries

**Figure 1**  
Vector of salaries

**Figure 2**  
Vector Slot Filled with `double` Value
Section 6.1: Using Vectors to Collect Data Items

Bounds error

- You have to be careful about index values; trying to access a slot that does not exist in the vector is a serious error.
- For example, since salaries only holds 10 values, then you are not allowed to access salaries[20]. If you do, this is called a bounds error.
- Note the compiler does not catch this type of error. This is the cause of a lot of errors and even security issues. (add in examples; heartbleed, supermario)
Section 6.2: Working with Vectors

Visiting all elements of a vector

- To obtain the size of a vector, use the `size` method:
  ```java
  salaries.size();
  ```

- A common way to visit all elements of a vector:
  ```java
  int i;
  for(int i = 1; i < salaries.size(); i++)
  {
    ...
  }
  ```
Section 6.2: Working with Vectors
Adding elements and removing elements

- In order to add an element to the end of a vector, use the `push_back` command:
  
  ```
  salaries.push_back(s)
  ```

  This adds a slot to the vector at the end, and then places \( s \) in the (newly-created) last slot.

- In order to delete the last element of a vector, use `pop_back`:
  
  ```
  salaries.pop_back(s)
  ```

  Note: this does not return the last element.
Section 6.2: Working with Vectors

The salaries program

```cpp
int main()
{
    vector<double> salaries;
    cout << "Please enter salaries, 0 to quit:\n";
    bool more = true;
    while (more)
    {
        double s;
        cin >> s;
        if (s == 0)
            more = false;
        else
            salaries.push_back(s);
    }
    double highest = salaries[0];
    int i;
    for (i = 1; i < salaries.size(); i++)
    {
        if (salaries[i] > highest)
            highest = salaries[i];
    }
    for (i = 0; i < salaries.size(); i++)
    {
        cout << salaries[i];
        if (salaries[i] == highest)
            cout << " <== highest value";
        cout << "\n";
    }
    return 0;
}
```
Section 6.2: Working with Vectors

The salaries program

**User input:**

Please enter salaries, 0 to quit:

32000
54000
67500
29000
35000
80000
115000
44500
100000
65000
0

**Output:**

32000
54000
67500
29000
35000
80000
115000 <= highest value
44500
100000
65000
Section 6.2: Working with Vectors

A couple of tips and facts

**Productivity Hint 6.1:** Use a debugger to check out your vectors. Make sure you are not committing a bounds error!

**Advanced Topic 6.1:** Strings are vectors of characters.
Vector as a parameter:

```cpp
double average(vector<double> values)
{
    if (values.size() == 0) return 0;
    double sum = 0;
    for (int i = 0; i < values.size(); i++)
        sum = sum + values[i];
    return sum / values.size();
}
```

This program computes the average of floating-point numbers.
Section 6.3: Vector Parameters and Return Values

Vectors can be parameters and/or return values

Vector as return-value:

```cpp
vector<double> between(vector<double> values, double low, double high)
{
    vector<double> result;
    for (int i = 0; i < values.size(); i++)
        if (low <= values[i] && values[i] <= high)
            result.push_back(values[i]);
    return result;
}
```
Section 6.3: Vector Parameters and Return Values

Modifying a vector using a reference parameter

```cpp
void raise_by_percent(vector<double>& values, double rate)
{
    for (int i = 0; i < values.size(); i++)
        values[i] = values[i] * (1 + rate / 100);
}
```
Passing a vector into a function by value is inefficient. You should pass a vector by reference or const reference. As an example, if you don’t plan on changing the called vector, you should do

```cpp
double average(const vector<double>& values)
```

This is a useful optimization that increases performance.
Removing an element, unordered

Removing an element is easy if you care about the order of the elements. For example, suppose you want to remove an element at position $pos$ from the vector $values$. Then simply replace it with the element in the last position:

```cpp
int last_pos = values.size() - 1;
values[pos] = values[last_pos];
values.pop_back();
```

![Diagram](image.png)

**Figure 5**  Removing an Element in an Unordered Vector
Section 6.4: Removing and Inserting Vector Elements

Removing an element, ordered

But if order DOES matter, then for every element past the one you want to remove, you have to slide it back. Then erase the last slot.

```cpp
for (int i = pos; i < values.size() - 1; i++)
values[i] = values[i + 1];
values.pop_back();
```

![Diagram](Image)

**Figure 6**
Removing an Element in an Ordered Vector
Section 6.4: Removing and Inserting Vector Elements

Quality Tip 6.2

Suppose you have data of three products, and their scores.

ACMA P600 Price: 995 Score: 75
Alaris Nx686 Price: 798 Score: 57
Blackship NX-600 Price: 598 Score: 60 <= best value
Kompac 690 Price: 695 Score: 60

One solution is to keep three vectors

```cpp
vector<string> names;
vector<double> prices;
vector<int> scores;
```

And slice $i$ will correspond to one of the three products.

![Diagram of parallel vectors](image)

Figure 8 Parallel Vectors
Section 6.4: Removing and Inserting Vector Elements

Quality Tip 6.2

A better solution is to turn the concept into a class: the `Product` class.

```cpp
class Product {
public:
    ...
private:
    string name;
    double price;
    int score;
};
...
vector<Product> products
```

![Figure 9](image.png)  
Eliminating Parallel Vectors
Section 6.5: Arrays
Introduction

- A second mechanism of C++ for collecting elements of the same type is by arrays
- Arrays are a lower-level abstraction than vectors, so they are less convenient. For example, you cannot resize an array; once the size of an array has been set, you cannot change it.
- Why arrays instead of vectors? Answers:
  - Vectors are a more recent addition to C++, and so older programs use arrays. So a working knowledge of arrays is useful for reading older programs.
  - Arrays are also faster and more efficient than vectors, which can be important in some applications.
- Array indexing starts at 0, just like vectors.
Section 6.5: Arrays
Defining and Using Arrays

How to make an array

double salaries[10];

of if you wanted to initialize some values (note: you can’t do this with a vector)

double salaries[] = { 31000, 24000, 55000, 82000, 49000, 42000, 35000, 66000, 91000, 60000 };
Section 6.5: Arrays
Array capacity and array size

You can’t use a method like salaries.size() if salaries is an array. Instead, you have to keep a companion variable.

```cpp
const int SALARIES_CAPACITY = 100;
double salaries[SALARIES_CAPACITY];
...
int salaries_size = 0;
while (more && salaries_size < SALARIES_CAPACITY)
{
    cout << "Enter salary or 0 to quit: ";
    double x;
    cin >> x;
    if (cin.fail())
        more = false;
    else
    {
        salaries[salaries_size] = x;
        salaries_size++;
    }
}
```
Section 6.5: Arrays

Array as a parameter

When using an array as a parameter, you need to place an empty [] after the parameter name, as well as pass the size of the array to the function.

```cpp
double maximum(const double a[], int a_size);
{
    if (a_size == 0) return 0;
    double highest = a[0];
    int i;
    for (i = 1; i < a_size; i++)
        if (a[i] > highest)
            highest = a[i];
    return highest;
}
```

Important fact: Arrays are always passed by reference, so you don’t need to use the & character. (Good style note: add the const keyword whenever a function does not actually modify an array.)
Section 6.5: Arrays

Array as a parameter

If you want to add elements to an array, you need to pass three parameters: the array itself, the capacity, and the current size.

```cpp
void read_data(double a[], int a_capacity, int& a_size) {
    a_size = 0;
    while (a_size < a_capacity) {
        double x;
        cin >> x;
        if (cin.fail()) return;
        a[a_size] = x;
        a_size++;
    }
}
```
Section 6.5: Arrays
Array as return-type, or not

The return type of a function cannot be an array. If you want the result of a function acting on an array, you must provide an array parameter to hold the result.

```c
void between(double values[], int values_size, double low, double high, double result[], double& result_size)
```
Character arrays are arrays of values of the character type `char`. There is more about this in the book: Section 6.5.3.
Section 6.5: Arrays

Two-Dimensional Arrays

▷ If you want to store tabular data, use a two-dimensional array.

```cpp
const int POWERS_ROWS = 11;
const int POWERS_COLS = 6;
double powers[POWERS_ROWS][POWERS_COLS];
```

▷ Just as with one-dimensional arrays, you cannot change the size of a two-dimensional array.

▷ You can access individual elements by using the \texttt{m[i][j]} notation. For example \texttt{powers[3][4]}.

\textbf{Figure 11}
Accessing an Element in a Two-Dimensional Array
Section 6.5: Arrays
How two-dimensional arrays are stored

Although these arrays appear to be two-dimensional, they are still stored as a sequence of elements in memory.

```
powers[3][4] = element in 3 * POWERS_COLS + 4
```

**Figure 12**  A Two-Dimensional Array Is Stored as a Sequence of Rows
When passing a two-dimensional array to a function, you must specify the number of columns as a constant with the parameter type. The number of rows can be variable. For example,

```cpp
void print_table(const double table[][POWERS_COLS], int table_rows)
{
    const int WIDTH = 10;
    cout << fixed << setprecision(0);
    for (int i = 0; i < table_rows; i++)
    {
        for (int j = 0; j < POWERS_COLS; j++)
        {
            cout << setw(WIDTH) << table[i][j];
        }
        cout << "\n";
    }
}
```

This function can print two-dimensional arrays with arbitrary numbers of rows, but the rows must have 6 columns. You have to write a different function if you want to print a two-dimensional array with 7 columns.
Section 6.5: Arrays

Why required to specify the number of columns?

Q: When passing an array, why do you have to specify the number of columns?

Ans: The reason is because the compiler finds the element powers[i][j] by computing the offset

\[ i \times \text{POWERS}_\text{COLS} + j \]

so it needs to know the number of columns beforehand.
Quality Tip 6.3

Name the array size and the capacity consistently. It is a good habit and prevents a lot of headache later. Make sure to use const for the capacity.
Section 6.5: Arrays

Common Error 6.2

A common error is the omit the column size of a two-dimensional array parameter.

```cpp
void print(const double table[][], int table_rows,
           int table_cols)  // NO!

const int TABLE_COLS = 6;
void print(const double table[][TABLE_COLS],
           int table_rows)  // OK
```
CHAPTER SUMMARY

1. Use a vector to collect multiple values of the same type.
2. Individual values in a vector are accessed by an integer index or subscript: v[i].
3. Valid values for the index range from 0 to one less than the size of the array.
4. A bounds error, which occurs if you supply an invalid index to a vector, can have serious consequences.
5. Use the size function to obtain the current size of a vector.
6. Use the push_back member function to add more elements to a vector. Use pop_back to reduce the size.
7. Vectors can occur as the function parameters and return values.
8. Avoid parallel vectors by changing them into vectors of objects.
9. Like vectors, arrays collect elements of the same type. Once the size of an array has been set, it cannot be changed.
10. Array parameters are always passed by reference.
11. The return type of a function cannot be an array.
12. Character arrays are arrays of values of the character type char.
13. Use a two-dimensional array to store tabular data.
14. Individual elements in a two-dimensional array are accessed by double subscripts m[i][j].
**Pointers 101**

**Syntax 7.1 new Expression**

```
new type_name
new type_name(expression1, expression2, ..., expressionn)
```

**Example:**
new Time
new Employee("Lin, Lisa", 68000)

**Purpose:**
Allocate and construct a value on the heap and return a pointer to the value.

**Syntax 7.2 Pointer Variable Definition**

```
type_name* variable_name;
type_name* variable_name = expression;
```

**Example:**
Employee* boss;
Product* p = new Product;

**Purpose:**
Define a new pointer variable, and optionally supply an initial value.

**Syntax 7.3 Pointer Dereferencing**

```
*pointer_expression
pointer_expression->class_member
```

**Example:**
*boss
boss->set_salary(70000)

**Purpose:**
Access the object to which a pointer points.