COMP 1633: Intro to CS II

More Arrays

Charlotte Curtis

February 5, 2024

Where we left off

- Arrays vs Python lists
- C-style arrays
- Array indexing
- Arrays in functions preview

> *Textbook Chapter 7*

```
int cup_sizes[] = {8, 12, 16, 20};
for (int i = 0; i < 4; i++) {
    cout << cup_sizes[i] << " oz" << endl;
}
```

Today's topics

- Passing arrays to functions, with and without `const`
- Partially filled arrays
- Sorting arrays
- Multidimensional arrays
- Preview of C-strings

> *Textbook Chapter 7, 8.1*

Passing arrays to functions

- Arrays can be passed to functions just like any other variable

```
double sum_10_elements(double arr[10]) {
    double sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += arr[i];
    }
    return sum;
}
```

- What if we give this an array with 15 elements?
- How about 5?

Better array passing

- For a flexible, reusable function, we need to pass the **size** of the array as well

> *Why wouldn't we just use a **global constant**? Or `sizeof`?*

- What if the function were declared as follows?

```
double sum_all(double arr[10], int size);
```

- The `[10]` is **ignored**, better to just write `[]`

Arrays are always passed by reference

- We might want to **modify** an array in a function

```
void add_one(double arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] += 1;
    }
}
```

- Arrays are **always passed by reference**, no `&` needed!
  - In fact, it's a compiler error to use `&` with an array
- If you only want to **read** the array, use `const`

```
double sum_all(const double arr[], int size);
```

Side tangent: `const` in function headers

- `const` in a function header means that the function **cannot modify** the parameter
- This **does not modify** the `const`-ness of the value that is passed

```
void print(const int x) {
    cout << "You passed " << x << endl; // no problem
    x = 5; // compiler error, x is const in this scope
}
...
int y = 10;
print(y); // no problem
y = 5; // also no problem
```

> *What if I changed it to `void print(const int &x)`?*

Summary of array passing

- Any size indicated in the `[]` of the function header is **ignored**
- The function receives a **pointer** to the first element of the array
  - `sizeof` information is lost
  - We'll talk more about pointers next week
- Pass the **size** of the array as a separate parameter to have flexible functions
- Arrays are always **passed by reference**, no `&` needed (or allowed)
- If you only want to **read** the array, use `const`

Returning arrays from functions

What about the following?

```
double[] get_temps() {
    const int FORECAST_DAYS = 7;
    double high_temps[FORECAST_DAYS] = {};
    for (int i = 0; i < FORECAST_DAYS; i++) {
        cout << "Enter high temp for day "
             << i + 1 << ": ";
        cin >> high_temps[i];
    }
    return high_temps;
}
```

- This is a **compiler error**! For now, just use the pass-by-reference mechanism

Array check-in 1/2

What is the output from the following code?

A. `0`

B. Random garbage

C. The address of the array

D. `4`

E. Runtime error

```
int arr[5] = {};
cout << arr[4] << endl;
```

Array check-in 2/2

What is the output from the following code?

A. `0`

B. Random garbage

C. The address of the array

D. `4`

E. Runtime error

```
int arr[5] = {};
cout << arr << endl;
```

Partially filled arrays

- Arrays of fixed-length seem quite limiting, especially coming from Python

```
high_temps = []
temp = float(input("Enter the next temperature: "))
while temp != -100:
    high_temps.append(temp)
    temp = float(input("Enter the next temperature: "))
```

- A workaround for C-style arrays is to allocate the **maximum size** you think you might need, then keep track of the **actual size** of the array
- This is called a **partially filled array**

Partially filled array example

```
const int MAX_SIZE = 30;
double high_temps[MAX_SIZE] = {};
int num_temps = 0;
double temp = 0;
cout << "Enter the next temperature: ";
cin >> temp;
while (temp != -100 && num_temps < MAX_SIZE) {
    high_temps[num_temps] = temp;
    num_temps++;
    cout << "Enter the next temperature: ";
    cin >> temp;
}
```

- Pretty verbose, but nothing is hidden
- The resulting `num_temps` is the **actual size** of the array

Searching arrays

- Searching through an array to find a value is a common task
- Example: find the **first day** with a temperature below 0

```
int first_freezing_day(const double temps[], int size);
```

- Things to consider:
  - What should be returned if there are no freezing days?
  - What should the LCV(s) be?
  - How does the loop terminate?

Sorting arrays

- Sorting algorithms are a **classic** topic in CS
- Tons of different algorithms with tradeoffs between **speed** and **memory**
- We'll look at a simple one called **selection sort** - not the fastest, but relatively easy to understand
- General algorithm:

```
Repeat until the array is sorted:
    Go through the array and find the smallest element
    Swap the smallest element with the first element
    Update the start of the array to be the next element
```

Multidimensional arrays

- So far we've only looked at **one-dimensional** arrays
- How about a **two-dimensional** array?
  - Say we want to represent a tic-tac-toe board
- Just like Python's **list of lists**

```
board = [['', '', ''],
         ['', '', ''],
         ['', '', '']]
```

```
const int ROWS = 3;
const int COLS = 3;
char board[ROWS][COLS] = {};
```

- The **first** dimension is the **rows**, the **second** is the **columns**

Multidimensional data types

Given this declaration:

```
const int ROWS = 3;
const int COLS = 3;
char board[ROWS][COLS] = {};
```

What is the type of each of the following?:

1. `board`
2. `board[0]`
3. `board[0][0]`

Multidimensional array initialization

- We can initialize a multidimensional array just like a 1D array

```
char board[ROWS][COLS] = {{' ', ' ', ' '},
                          {' ', ' ', ' '},
                          {' ', ' ', ' '}};
```

- But that gets tedious and is inflexible, a **nested loop** is probably better:

```
char board[ROWS][COLS];
for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLS; col++) {
        board[row][col] = ' ';
    }
}
```

Multidimensional array passing

- Multidimensional arrays are **passed by reference** just like 1D arrays
- An initialization function might have the following **prototype**:

```
void initialize(char board[][COLS], int size);
```

- Like 1D arrays, the **first** dimension is **ignored**, however...
- The **second** dimension **must** be specified, and it **must** be a **constant**!

> *This is probably a good place to use a global constant*

Processing row by row

Depending on the data, you might want to process one row at a time:

```
const int MAX_RECORDS = 100;
const int NUM_FIELDS = 5;
int records[MAX_RECORDS][NUM_FIELDS] = {};

for (int row = 0; row < MAX_RECORDS; row++) {
    read_record(records[row], NUM_FIELDS);
}
```

- What should the prototype for `read_record` look like?
- How could you process **column by column**?

C-string Preview

- C-style strings are **arrays of characters**
- We said that you can't do this:

```
int primes[] = {2, 3, 5, 7, 11};
cout << primes << endl;
```

- But what about this?

```
char vowels[] = {'a', 'e', 'i', 'o', 'u'};
cout << vowels << endl;
```

- We've actually been using C-strings all along!

Coming up next

- Assignment 1 due Friday
- Assignment 2 available next week: repeat of 1701 A4
- Lab: Arrays
- Next topic: C-strings + structures

> *Textbook Chapter 8.1, 8.2*