COMP 1633: Intro to CS II

C-Style Arrays

Charlotte Curtis

January 31, 2024

Where we left off

- `while` and `for` loops in C++
- Event controlled vs counted loops
- Some useful sentinels

> *Textbook Sections 3.3-3.4*

```
while (brain_is_working()) {
    keep_programming();
}
```

Today's topics

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

> *Textbook Chapter 7*

Side note: some handy online resources

Python tutor also does C++!

C++ for Python Programmers is an open source interactive book

> *Caution: we are doing things the ~~hard~~ low level way, so all mention of* `string` *and* `vector` *etc should not be used for now*

Python lists

- Remember the `list` type in Python?

```
cities = ["Calgary", "Vancouver", "Toronto"]
current_temp = [15, 18, 20]
```

- It's possible, but **not a good idea**, to have mixed data types

```
city_and_current_temp = ["Calgary", 15]
```

- **Arrays** in C++ are kind of like lists, but the data types **must be the same**
- We'll start by looking at "C-style" arrays

C-style arrays

- C-style arrays are a **fixed size** (length) collection of elements of the same type
- When an array is declared, memory is allocated all at once
- An array is **not** a separate data type! The general form of the declaration is:

```
data_type variable_name[array_size];
```

- For example:

```
double current_temp[3];
```

- The array size must be a **constant** (not a variable)

Arrays vs Python lists

```python
prices = [1.99, 2.99, 3.99]
print(f"The cost is ${prices[0]}")
```

```cpp
double prices[3] = {1.99, 2.99, 3.99};
cout << "The cost is $"
     << prices[0] << endl;
```

- ✓ Both store a sequence of values
- ✓ Both can be accessed by index
- ✗ Arrays have a fixed size, lists are **dynamic**
- ✗ Arrays can only store one type of value (lists *should* as well)
- ✗ Arrays are **not** objects, so they don't have methods
- ✗ Arrays are stored in **contiguous** memory

Array syntax

```
datatype array_name[array_size];
```

- `array_size` must be an **unsigned integer constant** value!
- **Declaring** an array creates a **block of memory** to store the values

```
int numbers[10];
```

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	?	?	?	?	?	?	?	?	?	?

Uninitialized variables

What is the output from the following code?

A. `0`

B. Random garbage

C. `-1`

D. `inf`

E. Runtime error

```
void uninitialized() {
    int x;
    cout << x << endl;
}
```

Array initialization

- Arrays can be **initialized** when they are declared

```
int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- However, this is the only time you can do this!

```
int numbers[10];
numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // no can do
```

- If the number of values in `{}` is less than `array_size`, the remaining values are **initialized to zero**

```
int numbers[10] = {1, 2, 3};
// numbers[3] through numbers[9] are 0
```

Inferring size from initialization

- If you **omit** the array size, the compiler will **infer** it from the initialization

```
int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // 10 elements
double other_nums[] = {1.5, 2.5, 3.5}; // 3 elements
double yet_more_nums[] = {}; // 0 elements, kinda pointless
double no_can_do[]; // sorry
```

- This could be convenient, but you probably need to know the size eventually
- With C-style arrays, it's up to the programmer to keep track of the size
- Good idea to define a **named constant** for the array size

```
const int NUM_DAYS = 10;
int numbers[NUM_DAYS];
```

Tangent: `sizeof`

- The `sizeof` operator returns the **size in bytes** of a variable or data type

```
int x = 5;
cout << sizeof(x) << endl; // prints 4
```

- The size of an array is the **total size** of the allocated memory

```
int numbers[10];
cout << sizeof(numbers) << endl; // prints 40
```

> *This is less useful than you might think*

Two meanings of `[]`

- `[]` are used to **declare** an array
- `[]` are also used to **index** into an array
  - Indexing gives the **value** at that index

```
int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout << numbers[0] << endl; // prints 1
cout << numbers[9] << endl; // prints 10
```

- The **data type** of the indexed element is the base type of the array:

```
cout << numbers[0] + numbers[9] << endl; // prints 11
```

Array indexing

- Like Python array indices start at 0 and count up
- Unlike Python, **no negative indices**!
- Any guesses what will happen here?

```
int numbers[10];
cout << numbers[0] << endl;
cout << numbers[10] << endl;
cout << numbers[-1] << endl;
```

Array operations

- After initialization, you **cannot** do any "whole array" operations, like:
  - Assigning one array to another
  - Comparing two arrays
  - Printing an array
  - Reading an array
  - Returning an array from a function

Take a guess

What do you think will happen here?

A. Compiler error

B. Runtime error

C. Prints `2`

D. Prints `NULL`

E. Prints the memory address of `primes`

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

Invalid array operations

While many array operations are **compile errors**, others are **logic errors**:

```
int primes[] = {2, 3, 5, 7, 11};
int prime_cpy = primes;       // compile time
int prime_cpy[] = primes;     // compile time
cin >> primes;                // compile time

if (primes == prime_cpy) {    // logic!
    cout << "Equal" << endl;
}
```

> *So, how do we do any of these things?*

Arrays + loops = ❤️

Array elements need to be processed **one at a time**

- The `for` loop is a natural fit for this:

```
int numbers[10];
for (int i = 0; i < 10; i++) {
    numbers[i] = 0;
}
```

- Exercise: write a program that:
  - Declares and initializes two arrays of equal length
  - Copies the values from one array to the other
  - Compares them for equality

Preview: arrays + functions

- Arrays can be passed to functions as **parameters**
- The parameter type is the same as the declaration

```
bool are_equal(int a[], int b[]);

int main() {
    const int SIZE = 10;
    int x[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int y[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    if (are_equal(x, y)) {
        cout << "Equal" << endl;
    }
}
```

> *Note that the `[]` are not passed to the function! `[]` is part of the **type***

What's missing?

If we're going to copy paste the equality code to the `are_equal` function, what **additional information** do we need to pass?

```
bool are_equal(int a[], int b[]) {
    bool equalness = true;
    // ... ?
    return equalness;
}
```

> *We're going to need the **size** of the arrays:*
>
> ```
> bool are_equal(int a[], int b[], int size);
> ```

Coming up next

- Lab: Buffer time, to work out git issues and work on assignment 1
- Lecture: more arrays, arrays + functions, multidimensional arrays
- Assignment 1: Due February 9, 2024 (Next Friday)

> *Textbook Chapter 7*