C++ Arrays, Pointers
Review: Java and C++ are very similar

- **Similar in:**
  - Syntax: Java used syntax similar to C++ to ease adoption
  - Principles: Both are object-oriented languages
  - Execution: Many similarities when run on a machine
    - Compiled down to similar assembly language

- **Different in goals:**
  - Java designed for: safety and portability
  - C++ designed for: performance and control

As a result, C++ exposes aspects of execution that Java hides
What we’ve talked about so far

- Main function: where everything starts
  - #include: “” for things you write, <> for things you didn’t

- Printing things out with std::cout and std::endl
  - Namespaces, using, scope resolution operator (::)

- Object declaration
  - .h (declarations) include guards public/private regions
  - .cpp (function implementation)

- Constructors
  - Don’t rely on default constructors; primitives uninitialized
  - Initializer lists: for init-ing children –and– calling other constructors
What we’ve talked about so far, cont.

- Allocating objects, two ways:
  - On the stack: uses same notation as primitives
    - “Deallocated” when they leave scope
  - On the heap: returns a pointer to the allocated thing
    - Thing *thing = new Thing();
    - Need to manually delete this memory.

- Useful tools for looking at memory
  - & (address of operator)
  - sizeof(thing) says how big “thing” is
Arrays in C++ vs. Java (Primitives)

- Arrays of primitives in C++ are similar to those in Java
  - Laid out sequentially
  - Unlike Java, they don’t know how big they are
    - They don’t prevent you from accessing elements that don’t exist
    - It is common in C/C++ to keep an int length with an array

```cpp
int array[10];
cout << sizeof(int) << " " << sizeof(array) << endl;
cout << &array[0] << " " << &array[1] << endl;
```
Arrays in C++ vs. Java (Objects)

- In Java, arrays of objects are **arrays of references** to objects
  - We had to do:
    ```java
    Double [] doubles = new Double[100];
    for (int i = 0; i < doubles.length; i++) {
        doubles[i] = new Double(i);
    }
    ```
  - Also, arrays were objects themselves (heap allocated)
Arrays in C++ vs. Java (Objects)

- In C++, arrays of objects can be **arrays of objects**
  - Allocated in place, just like primitive values
    ```cpp
    ExpressionValue eValueArray[10];
    cout << sizeof(ExpressionValue) << " " << sizeof(eValueArray) << endl;
    cout << &eValueArray[0] << " " << &eValueArray[1] << endl;
    ```
  - Arrays can be **stack or heap allocated**
    - If heap allocated, it returns a pointer to the type
      ```cpp
      ExpressionValue *eValueArray2 = new ExpressionValue[10];
      ```
So let’s talk more about pointers

- **Declaring**: use a `*` in declaration
  - `int *myIntPtr;`
- **Assignment**: must match type
  - `myIntPtr = &myInt; // & is address of operator`
  - `myIntPtr = new int; // new returns a pointer`
- **Copying**: from one pointer to another
  - `int *myOtherIntPtr = myIntPtr;`
- **De-referencing**: use `*` in expression to get to the value
  - `*myOtherIntPtr = 7; // assigning value pointed to`
  - `int justAnInt = *myIntPtr; // getting value of thing pointed to`
What is happening here? (Draw a picture!)

```
int myInt = 8;
int *myIntPointer;
myIntPointer = &myInt;
myIntPointer = new int;
int *myOtherIntPointer = myIntPointer;
*myOtherIntPointer = 7;
int justAnInt = *myIntPointer;
cout << justAnInt << endl;
```

A) 7  
B) 8
More pointer puzzles

```c
int a = 1, b = 2, c = 3;
int *p1 = &a;
int *p2 = &b;
int *p3 = &c;
p1 = p2; /* Correct */
*p1 = *p3; /* Correct */
*p3 = 5; /* Correct */
```

Output:
```
1 3 5
```

Options:
A) 5 5 5
B) 3 2 5
C) 1 2 5
D) 1 3 5
E) 5 1 3
C++ pointers point to arrays or individuals

```cpp
int *anotherIntPtr = new int[10];
for (int i = 0; i < 10; i++) { anotherIntPtr[i] = i; }

anotherIntPtr = &(anotherIntPtr[2]);
cout << *anotherIntPtr << endl;

for (int i = 0; i < 8; i++) {
    cout << anotherIntPtr[i] << endl;
}
int justAnInt = 7;
anotherIntPtr = &justAnInt;
cout << *anotherIntPtr << " ";
cout << anotherIntPtr[0] << endl;
```

![Diagram showing the use of pointers and arrays in C++](image)
C++ pointers point to arrays or individuals

- As a result, you need to tell `delete` if what you are deleting is an array:
  - Use `delete []` for arrays, `delete` for single things

```cpp
ExpressionValue *eValuePtr = new ExpressionValue();
delete eValuePtr;

ExpressionValue *eValueArray2 = new ExpressionValue[10];
delete [] eValueArray2;
```