# **Review: C++ Arrays**

* In many programming languages (like C++) you have two types of arrays:

1. Statically declared (compile-time)
2. Dynamically declared (run-time)

## **C++ Static Arrays**

* Statically declared arrays are fixed in size.
* They are allocated memory at compile time and their size is fixed (i.e., their size cannot be changed later).

**Static array syntax**: **type name[size];**

* For example: two int arrays are declared, one initialized, one not.

int a[10];

int b[5] {8, 20, 25, 9, 14};

* The **a** array has 10 elements with subscripts numbered from 0 to 9, filled with garbage.
* The **b** array has 5 elements with subscripts numbered from 0 to 4, filled with the five given values. They accessed in the usual way, e.g., a[5] or b[2].
* When declaring a static array, the length of the array (between the square brackets) must be a compile-time constant.
* This is because the length of a static array must be known at compile time.
* A memory picture of these arrays show 10 int garbage values and 5 valid int values:

\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_

a| -|-->|...|...|...|...|...|...|...|...|...|...|

-- --- --- --- --- --- --- --- --- --- ---

0 1 2 3 4 5 6 7 8 9

\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_

b| -|-->| 8 | 20| 25| 9 | 14|

-- --- --- --- --- ---

* You can declare a static array using:
  1. A literal constant
  2. A const/constexpr symbolic constant
  3. An enumerator
  4. A macro

## **C++ Dynamic Arrays**

* Dynamically declared arrays are also fixed in size but have the flexibility to be assigned to a new array with a smaller/larger size that is a non-constant or calculated at runtime.
* You cannot change the actual size of the array itself, but dynamic arrays allow the flexibility to allocate a new array that's larger, copy the values you want to keep, delete the original array, and change the array pointer to point to the new array.
* Here are the steps to initialize a new dynamic array (and copy over an old array’s values):
  1. Allocate a new[] array and store it in a temporary pointer.
  2. Copy over the previous values that you want to keep.
  3. Delete[] the old array.
  4. Change the member variables, ptr and size to point to the new array and hold the new size.

**Dynamic array syntax**: **type\* name = new type[size]**

* The memory picture is identical to the static array, but you can initialize a new array and change what the pointer points to.
* The following code fragment illustrates how an array, arr, which initially has length 10, can be expanded as needed:

Text

Description automatically generated

## **Java vs. C++ Arrays**

* Java arrays are close to C++ dynamic arrays but have a simpler syntax.
* In Java, there is no distinction between a fixed-size and dynamic array.
* Their syntax is a mix of the two, but they are essentially dynamic arrays without the pointers (the pointers are implicit).
* You can set the size of an array at runtime like the following:

int **actualSize** = . . .;

**Employee[]** staff = **new Employee**[**actualSize**];

## **Problem: Fixed Size**

* Like C++ arrays, each Java array you initialize has a fixed size.
* What if we wanted to keep a single array, and dynamically modify its size at runtime? Neither C++ nor Java arrays can actually do this.
* If you needed to resize an array, you would have to create a new array with a larger size and copy over the contents from the previous array to the new one.
* This forces programmers into uncomfortable trade-offs.
* How much space should we initialize the array with?
* Should we overestimate or underestimate?
* Do we want to waste 90 entries when we initially only need 10?
* In Java you can deal with this common situation by using another Java class, called ArrayList.

# **ArrayList<E>**

## **ArrayList Description**

* The Java API provides a class that is similar to an array but can automatically adjust its capacity.
* Internally, each ArrayList object uses an array to store its values.
* As a result, an ArrayList provides the same fast random access as a regular array.
* But unlike with an array, with an ArrayList you can make simple requests to add or remove values, and the ArrayList takes care of all of the details for you:
  + If you add values to the list, it makes the array bigger
  + If you remove values, it makes the array smaller
  + If these operations happen in the middle of the list, it handles any element shifting that needs to be done.

## **ArrayList Initialization**

**Import Package**

* The ArrayList class is part of the java.util package, so to include it in a program, you must include an import declaration:

import java.util.\*; // for ArrayList

**Generics**

* Remember that you can declare arrays of different types.
  + If you want an array of int values, you declare a variable of type int[].
  + For an array of String values, you use the type String[].
* This is a special syntax that works just for arrays, but the ArrayList class has ***almost*** the same flexibility.
* If you read the API documentation for ArrayList, you’ll see that it is actually listed as ArrayList<E>. This is an example of a **generic class** in Java.
* Generic classes are only compatible with classes, and not primitives. That’s why Java has such classes as Integer, Character, etc..
* Therefore, generic classes require **type parameters**.

**Type Parameters**

* The syntax for constructing an ArrayList is more complicated than what we’ve seen before because of the required **type parameter**.
* To specify the type of the element objects that the ArrayList holds, you append a class name enclosed in angle brackets, such as ArrayList<Employee>.
* For example, you could construct an ArrayList of Strings as follows:

// construct a list of strings (long form)

**ArrayList<String> list = new ArrayList<String>();**

* This code constructs an empty list of strings.

**<E> Is A Part of The Type**

* The syntax is complicated, but it will be easier to remember if you keep in mind that the <String> notation is actually part of the type.
* This isn’t simply an ArrayList, it is an ArrayList<String>, often read as “an ArrayList of String.”
* Notice how the type appears on both sides of the = sign, when you declare the variable and when you call the constructor.
* If you think in terms of the type being ArrayList<String>, you’ll see that this line of code isn’t all that different from the code used to construct a regular object like a Point.
* In both cases, you write the variable’s type on the left side of the = sign before its name and after the = sign and the keyword new (which calls that type’s constructor to create an object of that type):

// construct a normal object

Point p = new Point();

**Simpler <E> Notation**

* It can be cumbersome to list the element type <String> twice, so Java version 7 introduced a new shorter syntax for declaring collections called the “diamond operator” whereby the element type may be omitted on the right side of the statement and replaced by <>, such as:

// construct a list of strings (short form with diamond operator)

ArrayList<String> list = new ArrayList<>();

* The diamond operator syntax is shorter and more convenient, and its behavior is identical to that of the longer code.

**Compile-Time Checking**

* Java will make sure that you add values of an appropriate type.
* In this case, because you requested an ArrayList<String>, you can add String elements to the list.
* If you try to add a value of a different type, such as an integer or Point, the code will generate a compiler error.