← course home (/table-of-contents#section_general-programming_concept_garbage-collection)

Garbage Collection

A **garbage collector** automatically frees up memory that a program isn't using anymore.

For example, say we did this in Java:

```java
public static int getMin(int[] nums) {
    // NOTE: this is *not* the fastest way to get the min!
    int[] numsSorted = Arrays.copyOf(nums, nums.length);
    Arrays.sort(numsSorted);
    return numsSorted[0];
}

int[] myNums = new int[] {5, 3, 1, 4, 6};
System.out.println(getMin(myNums));
```

Look at `numsSorted` in `getMin()`. We allocate that whole array inside our function, and once the function returns we don't need the array anymore. In fact, once the function returns we *don't have any references to it anymore*!

What happens to that array in memory? The Java garbage collector will notice we don't need it anymore and free up that space.

> How does a garbage collector know when something can be freed?
>
> One option is to start by figuring out what we *can't* free. For example, we definitely can't free local variables that we're going to need later on. And, if we have a vector, then we also shouldn't free any of the vector's elements.
>
> This is the main intuition behind one garbage collector strategy:
>
> 1. Carefully figure out what things in memory we might still be using or need later on.
> 2. Free everything else.

> This strategy is often called **tracing garbage collection**, since we usually implement the first step by tracing references from one object (say, the vector) to the next (an element within the vector).
>
> A different option is to have each object keep track of the number of things that reference it—like a variable holding the location of an array or multiple edges pointing to the same node in a graph. We call this number an object's **reference count**.
>
> In this case, a garbage collector can free anything with a reference count of zero.
>
> This strategy is called **reference counting**, since we are *counting* the number of times each object is *referenced*.

Some languages, including C++, don't have a garbage collector. So we need to manually free up any memory we've allocated (say, with operator `new`) once we're done with it:

C++

```
// make a string that can hold 15 characters
// including the terminating null byte ('\0')
char* str = new char[15];

// ... do some stuff with it ...

// we're done. free that memory!
delete[] str;
```

We sometimes call this **manual memory management**.

C++ used to only have manual memory management, but now it has smart pointers that are automatically freed when you don't need them anymore.

Manual memory management is still supported, but using smart pointers to automatically manage and free resources is generally preferred—especially when you're working with exceptions, since allocated resources are automatically freed when an exception is thrown.

C++ has two main types of smart pointers: **unique pointers** and **shared pointers**. A unique pointer is the *only* reference to its allocated resource, and as soon as the unique pointer is out of scope its resource can be deallocated. With shared pointers, multiple pointers can reference the same resource, and the resource is only freed when when *all* the references are out of scope.

← course home (/table-of-contents)