Interview Cake

← course home (/table-of-contents)

# In-Place Algorithm

An **in-place** function modifies data structures or objects outside of its own stack frame (i.e.: stored on the process heap or in the stack frame of a calling function). Because of this, the changes made by the function remain after the call completes.

> In-place algorithms are sometimes called **destructive**, since the original input is "destroyed" (or modified) during the function call.

> **Careful: "In-place" does *not* mean "without creating any additional variables!"** Rather, it means "without creating a new copy of the input." In *general*, an in-place function will only create additional variables that are $O(1)$ space.

An **out-of-place** function doesn't make any changes that are visible to other functions. Usually, those functions copy any data structures or objects before manipulating and changing them.

In many languages, **primitive** values (integers, floating point numbers, or characters) are copied when passed as arguments, and more complex **data structures** (vectors, heaps, or hash tables) are passed by reference. In C++, arguments that are pointers or references can be modified in place.

Here are two functions that do the same operation on a vector, except one is in-place and the other is out-of-place:

C++

```
void squareVectorInPlace(vector<int>& intVector)
{
    for (size_t i = 0; i < intVector.size(); ++i) {
        intVector[i] *= intVector[i];
    }


    // NOTE: no need to return anything - we modified
    // intVector in place
}

vector<int> squareVectorOutOfPlace(const vector<int>& intVector)
{
    // we create a new vector with the size of the input vector
    vector<int> squaredVector(intVector.size());

    for (size_t i = 0; i < intVector.size(); ++i) {
        int item = intVector[i];
        squaredVector[i] =  item * item;
    }

    return squaredVector;
}
```

**Working in-place is a good way to save time and space.** An in-place algorithm avoids the cost of initializing or copying data structures, and it usually has an $O(1)$ space cost.

**But be careful: an in-place algorithm can cause side effects.** Your input is "destroyed" or "altered," which can affect code *outside* of your function. For example:

C++

```
vector<int> originalVector = {2, 3, 4, 5};
squaredVector = squareVectorInPlace(originalVector);

cout << "original vector: [";
for (size_t i = 0; i < originalVector.size(); ++i) {
    if (i > 0) {
        cout << ", ";
    }
    cout << originalVector[i];
}
cout << "]" << endl;
// prints: original vector: [4, 9, 16, 25], confusingly!
```

**Generally, out-of-place algorithms are considered safer because they avoid side effects.** You should only use an in-place algorithm if you're space constrained or you're *positive* you don't need the original input anymore, even for debugging.

← course home (/table-of-contents)

Next up: Dynamic Array → (/concept/dynamic-array?course=fc1&section=array-and-string-manipulation)