Vectors and Arrays

William E. Skeith III

Section 1

Motivation

Consider the following problem: read integers from standard input, and print them back out in reverse order. With the tools we've seen so far, we actually don't have a good way to do this! If you think about it, no fixed number of post-it notes would suffice, as we have to store the entire input. Note that up until this point, the amount of space used by our programs was essentially determined *by the program source code itself*, and was *independent of the input*.[1] For example, the final version of our Fibonacci sequence program used 5 post-it notes, irrespective of the input n.

But now we need something different. No matter how many variables we declare (call this number c), our program for reversing the input won't work as soon as someone gives us an input with a list of $c+1$ or more integers. We need to be able to create post-it notes "on the fly", as the program runs. One simple way to do this is through *vectors*. Vectors give you an expandable container of post-it notes. Let's see how to use them.

Section 2

Vector Basics

You can visualize a vector as a sequence of post-it notes, for example, something like this:

$V =$ | 34 | 42 | 59 | 125 | 145 |

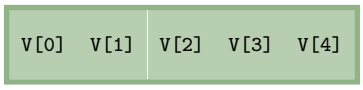
Great! That certainly looks like it could be useful. And it will be, as soon as we get answers to a few basic questions you are no doubt already asking. In particular, we need answers to the following:

(Q1) What is the name of each little box / post-it note?
(Q2) How do I actually declare one of these in a C++ program?
(Q3) How can I add (or remove) something to a vector?

Let's go through them one by one.

[1] The one exception you may have noticed was that `string` variables seem like they could be arbitrarily long. It turns out that `string`s are just specialized `vector`s!

The name of each little box (Q1). They each get a number, starting at zero, like this:

$V =$ 

(**Terminology:** the little boxes are usually called *elements* and the number associated with a box is called its *index*. So in our above example, the index of the element with the 59 would be 2, and conversely the element at index 2 has value 59.)

How to declare a vector (Q2). This actually involves a new concept: *templates*. The ability to make an expandable list of variables seems of general utility—it's something we might want to do for *many different datatypes*. We might want a vector of integers, or a vector of characters (like a string), or perhaps even a vector of vectors! Good news: when you declare a vector, you get to choose the type of the little boxes inside it. Examples:

```
vector<int> V;  /* each V[i] of type int */
vector<char> C; /* each C[i] of type char */
vector<bool> B; /* each B[i] of type bool */
vector<vector<int>> M; /* each M[i] is a vector<int>! */
```

Note the angled brackets with a datatype inside. You can put any datatype you want in there![2] And if you happen to know a list of values that you want your vector to contain initially, you can declare it using the following syntax (if you have a recent-ish C++ compiler).

```
vector<int> V = {34,42,59,125,145};
```

Adding and removing elements (Q3). Upon first declaring a vector, it will be empty:

```
vector<int> V;
```
$V =$

Important: trying to access the boxes that don't exist will end badly! We can't yet access `V[0]`, because *there is no `V[0]`*. To add an element, use the `push_back()` function like so:

```
V.push_back(34);
```
$V =$ 

As the name suggests, new values will be added to the "back" (the new element will have the largest index):

```
V.push_back(42);
V.push_back(59);
```
$V =$ 

You can also remove elements from the end of a vector with `pop_back()`. Applying it to the vector above would remove the 59 from the end:

```
V.pop_back();
```
$V =$ 

[2]Technically, any data type that is instantiatable. So things like `void` of course won't work.

Section 3

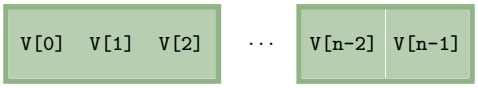
Examples

3.1 Reversing `stdin`

Let's begin by solving the problem we listed as our original motivation: read all of `stdin` (as integers), and print them back out in reverse order. As noted, we need to store the entire input before printing anything. Here's one way to do it:

```
int x; /* storage for single input */
while (cin >> x) {
    V.push_back(x);
}
```

Nothing surprising, but perhaps a new question has popped into your head: how do we know how many elements a vector has? Fortunately `V` knows its own size: you can get it via `V.size()`, so the elements look like this, where `n = V.size()`:



With that knowledge, this problem is as good as solved: just print `V[V.size()-1..0]`:

```
size_t n = V.size(); /* give V.size() a nicer name */
for (size_t i = 0; i < n; i++) {
    cout << V[n-1-i] << "\n";
}
```

Note: You may be wondering why we didn't use the following, perhaps more natural looking loop, starting `i` at `n-1` and counting backwards to `0`:

```
for (size_t i = n-1; i >= 0; i--)
    cout << V[i] << "\n";
```

The issue is that the use of the *unsigned* type `size_t` (which seemed a good choice for storing an index, since negative indexes make no sense) opened us up to some variation of the famous "Gandhi bug":[3] for the datatype `size_t`, $-1 = 2^{64} - 1$ on a 64 bit computer. See for yourself:

```
size_t n = -1;
cout << n << "\n"; /* prints 18446744073709551615.  Yikes! */
```

[3] In case you are unfamiliar, the game *Civilization* featured a character modeled after Indian civil rights activist Mahatma Gandhi. Behind the scenes, each character had an aggression meter, stored as an unsigned integer. The bug (feature?) was that actions which should make characters more peaceful had the opposite effect on Gandhi: his aggression meter started out at 1, so if it were decremented twice, it would become $-1 \equiv 2^k - 1$ and suddenly he would set out to destroy all life on Earth `x_x`

More to the point, the natural looking condition `(i >= 0)` is the same as the constant `true`. Hence, the loop above is equivalent to the following (obviously wrong, infinite) loop:

```
for (size_t i = n-1; true; i--)
    cout << V[i] << "\n";
```

Exercises. It is probably a good time to pause and test your understanding with a few simple exercises. Before reading further, try to solve the following problems.

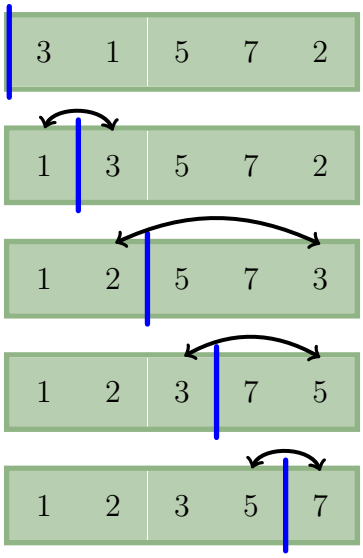
1. Write a function `bool search(const vector<int>& V, int x);` that searches vector `V` for value `x`.
2. Write a function `void reverse(vector<int>& V);` that reverses the order of the elements of `V`.
3. Given a vector of integers `V`, write code that creates a new vector `E` which contains only the even elements of `V`. That is, if `V = {1,3,5,2,7,8}`, then `E` should be `{2,8}`.
4. Given a vector `V` of *sorted* integers, write code that creates another vector `U` which is like `V`, but without duplicate elements. For example, if the input is `V = [1,2,2,3,3,3]`, then `U` should be set to contain `[1,2,3]`.

3.2 Sorting a vector

Let's try to tackle a slightly more involved problem: rearranging the elements of a vector (say of integers) so that they are in ascending order. For example, if the input was `V = {3,1,5,7,2}`, then we would like to rearrange elements so that `V = {1,2,3,5,7}`. We will use a very simple method called *selection sort*. You should be warned that it's not a very fast way to accomplish this task; we choose here it for its conceptual simplicity and ease of implementation.[4]

The selection sort algorithm can be summarized as follows: for each index `i = 0,...,n-1`, swap `V[i]` with the smallest element in `V[i..n-1]`, where `n` is the size of the vector. In more detail, we begin by finding the absolute smallest thing and placing it first (leaving whatever was in the first position somewhere else in the vector). At this point, the value in `V[0]` has reached its final destination—we never have to look at it again. Now we do the same thing to the *subvector* `V[1..n-1]`, and so on, until everything is in its right place. Before writing a program, let's sketch out a run of the algorithm in pictures. To show the value of `i`, which tells how much of the vector we've got sorted, we'll draw a blue line: *everything left of the blue line is in its right place.*

[4]However, it may be worth noting that this same concept can be made efficient: the *heap sort* algorithm is conceptually identical, with one building block (finding the smallest element) replaced by a more efficient version.



With a clear picture in hand, I think we're ready to start programming. Let's start by making a useful building block: find the location of the smallest element after the blue line. Note that it is important that we find the *location* and not just the value, since we are rearranging the elements. Here's a plausible prototype for such a function:

```
size_t indexOfSmallest(const vector<int>& V, size_t start);
```

Naturally, this function will look quite similar to computing a maximal or minimal value—it's just that we don't care about the *value* so much as we do about the index. This should be not too difficult for the reader at this point, and makes for a good exercise. Try to write this function right now before reading further (revisit lecture 1 if needed).

Here is one possible solution to the above. The invariants are listed in the comments.

```
size_t indexOfSmallest(const vector<int>& V, size_t start)
{
    size_t iMin = start; /* meaning: index of smallest element we've
                            seen so far.  (That is, from V[start..i]
                            in the loop below.) */
    for (size_t i = start+1; i < V.size(); i++) {
        if (V[i] < V[iMin]) /* found a smaller element! */
            iMin = i; /* preserve intended meaning of iMin */
    }
    return iMin;
}
```

All that remains is to perform the appropriate swaps. For each index `i`, starting from 0, we want `V[i]` $\leftrightarrow$ `V[s]`, where `s` is the index of the smallest element in the subvector `V[i..n-1]`, where `n=V.size()`. Taking a `swap` function for granted (you can find one in the STL's `<algorithm>`), our finished product might look like this:

```
void sort(vector<int>& V)
{
    for (size_t i = 0; i < V.size()-1; i++)
        swap(V[i],V[indexOfSmallest(V,i)]);
}
```

The program reads almost identically to our plain-English description of the process: *for each index `i = 0,...,n-1`, swap `V[i]` with the smallest element in `V[i...n-1]`*. Note that we stopped the loop *one before* the last element, since the last swap is guaranteed to pointless: it would always swap `V[n-1]` $\leftrightarrow$ `V[n-1]`.

Exercises.

1. Try to write selection sort from scratch. Refer back to the picture if needed, but don't look at any of the code.

2. Would our sorting algorithm still work if in the main loop we started at `V.size()-1` and went backwards to 0 like this?

   ```
   void sort_maybe(vector<int>& V)
   {
       for (size_t i = V.size()-1; i != -1; i--)
           swap(V[i],V[indexOfSmallest(V,i)]);
   }
   ```

3. Write a main program to test your sort function. Read all of `stdin` (as integers) and print it back out in sorted order.

4. Write a function that evaluates a polynomial $f(x) = \sum_{i=0}^{d} c_i x^i$ at a given point x. Your function should take as input a vector `c` for the coefficients (`c[i]` stores c_i) and the evaluation point `x`. Assume all c_i and x are integers.

5. Write an efficient search function for a *sorted* vector. The idea is to perform a comparison with an element in the middle of the vector, and then eliminate *half* of the vector from the search based on the result of the comparison. Continue in the same manner, but now acting on the remaining *subvector*. (This is called *binary search*.)

Section 4

Arrays (Vectors Behind the Scenes)

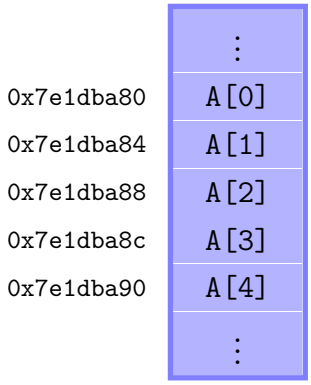
Note: for this section, we will need a tiny bit of knowledge about pointer variables. These are covered in the first section of the next lecture. It would be a good idea to browse that material right now, and then return. However, we essentially need only these two items for our discussion:

- The *address-of* operator (`&`) gives you the *memory address* of its argument. For a variable `x`, the expression `&x` will evaluate to the address of `x`.
- The *dereference* operator (`*`) takes a memory address, and gives the contents of that memory location. For a variable `p` containing an address, the expression `*p` can be used to read from or write to memory.

4.1 Arrays

Vectors are just a wrapper for a very simple data structure called an *array*. What's an array, you ask? It's a contiguous block of memory. It's the simplest thing that might warrant the name "data structure". Here's how to declare an array of 5 integers, and what it might look like in main memory:

```
int A[5];  /* A[0],...,A[4]
              are integers */
```



Notice that `A[i+1]` is always *four* memory address away from `A[i]`. This is because each value of type `int` takes 4 bytes to store, and main memory is *byte-addressable* (each address corresponds to a unique byte). Had we declared `A` as `char A[5]`, the addresses would have jumped by just one instead, since `sizeof(char)` is 1. Let's see what's inside the variable `A`. What happens if we print it out?

```
cout << A << "\n";       /* prints 0x7e1dba80 */
```

Interesting... We get a *memory address*. In particular, we get the memory address of the first element of `A`:

```
cout << &A[0] << "\n";  /* prints 0x7e1dba80 */
```

So `A` seems synonymous with the address of the first element. There is one subtle caveat to this[5], but in many cases an array variable is quite simply a pointer to the first element. Regardless, this intuition will hardly ever lead you astray, so you may as well use it. For emphasis:

Array variable ≡ pointer to first element.

[5]The caveat is that *there is no `A`* for statically allocated arrays as shown above. Note that the *elements are stored*, but there is actually no storage for the pointer `A` itself. This is the case since the address of the first element can be computed as a fixed offset from a CPU register (stack frame pointer, like `%rbp` on x86_64).

It turns out that the bracket notation for referencing elements of an array (`A[i]`) is just *syntactic sugar* for the following: add `i` to the memory address `A`, and dereference. That is:

$$\texttt{A[i]} \equiv \texttt{*(A+i)}.$$

Kind of fun consequence:

$$\texttt{A[i]} \equiv \texttt{*(A+i)} \equiv \texttt{*(i+A)} \equiv \texttt{i[A]}$$

Try it out! Write a bit of code using an array and then swap out `A[i]` with `i[A]`. Note that it works, and then **please never do it again**, haha.

Given that `A[i]` does the right thing for arrays of `char` as well as arrays of `int`, one can deduce the following fact regarding pointer arithmetic: the value of `A+i` (which is a memory address) actually depends on the *datatype* of `A`. If we add 1 to a `char` pointer, the pointer increases by 1. If we add 1 to an `int` pointer, it will actually increase by 4. In general, `A+i` will of course be `i*sizeof(A[0])` addresses ahead of `A`. Nothing complicated, but it is nice that the compiler saves us that little bit of headache.

4.2 Remarks and Warnings

Here we quickly cover a few things worth mentioning to anyone just learning about arrays in C/C++.

Writing out of bounds. There is nothing to stop you from referencing array elements that don't exist. More troubling still, is that an array (unlike a vector) typically doesn't even know its own size![6] So the programmer must be careful to keep track of these things. For example, we could do the following:

```
int A[5];
A[1000] = 0; /* writing outside of the array! x_x */
```

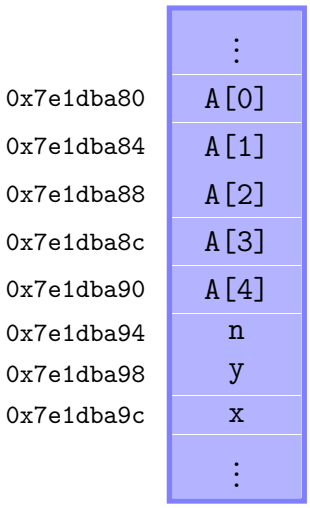
The above may cause your program to crash, or it may write to some other part of your program's memory and keep on running. Yikes! Here's a more explicitly spooky example:

```
int main()
{
    int x = 99;
    int y = 99;
    const int n = 5;
    int A[n];
    for (size_t i = 0; i < 8; i++)
```

[6]For static arrays (all we've seen so far), the `sizeof` operator will tell you the total size in bytes, so the number of elements could be recovered via `sizeof(A)/sizeof(A[0])`. But in the case of dynamically allocated arrays (covered next lecture), `sizeof(A)` will just give you the size required to store a pointer on your machine, irrespective of the size of the array to which it points.

```
        A[i] = i; /* out of bounds for i=5,6,7  */
    for (size_t i = 0; i < n; i++)
        printf("A[%lu] = %i\n",i,A[i]);
    printf("x == %i\n",x); /* prints "x == 7" o_O */
    printf("y == %i\n",y); /* prints "y == 6" o_O */
    return 0;
}
```

Notice that we never assigned anything to `x` or `y` after they were initialized to `99`, and yet both values had changed by the end of the program. We will postpone a full discussion of this phenomenon for later, but it might not be too hard to imagine how those variables were changed. The following picture of a possible memory layout[7] for such a program might be helpful:



You can see that `A[5]`, which is out of the bounds of `A` becomes a synonym for `n`, and `A[6],A[7]` become synonyms for `y,x` respectively.

While it is every bit as possible to write out of bounds with vectors, they do make it a little less likely. For one, the vector knows its own size, so you don't have to keep track of it separately (thus reducing the likelihood of mistakes). Also, vectors can help you catch this kind of problem early on during testing: with GCC, for example, if you compile with `-D_GLIBCXX_DEBUG`, a bit of extra code will run whenever you access an element to make sure you are staying in bounds. Once you're confident your program has no such bugs, you can remove that flag, and the extra checks will disappear along with it.

Character arrays (C-strings). Arrays of `char` are specially interpreted by many functions. They are the C-style way to store strings. For example, consider the following:

[7] Examples of this sort (relying on undefined behavior) can be kind of "fragile". For what it's worth, to make this "work" on GCC 8.2 it had to be compiled with flags `-O0 -fno-stack-protector`.

```
int A[5] = {0,1,2,3,4};
cout << A << "\n"; /* prints 0x7ff... */
char C[5] = {'h','i','!','!',0};
cout << C << "\n"; /* prints "hi!!" */
```

So, it seems that `cout` tries to print *the elements* when given a `char` array, but not so for arrays of `int`. **Note:** there is an important convention when dealing with character arrays: you should make sure that they end in a *zero* byte (not the character that displays a `'0'`, but the actual number `0`, as shown above). This is the (dubious[8]) way that C-strings can determine their own length: they scan for a zero (often called *null character*). Observe:

```
char s[7] = "oh hai"; /* quotes automatically add null char!
                         the 7 is not a typo! */
cout << s << "\n"; /* prints "oh hai" */
cout << (s+3) << "\n"; /* prints "hai" */
s[4] = 0; /* effectively reduces length of string to 4... */
cout << (s+1) << "\n"; /* prints "h h" */
```

4.3 Vector Internals

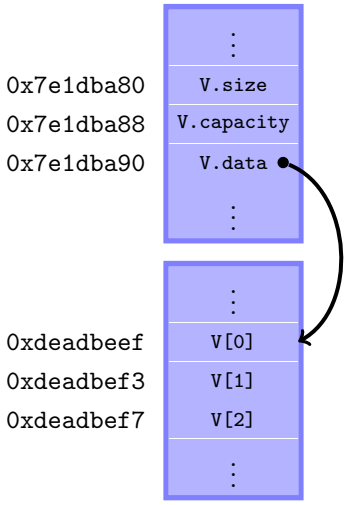
Finally, we take a quick look at how vectors are stored behind the scenes. Conceptually, a vector consists of 3 parts:[9]

- `data`: a pointer to an array it uses for the elements
- `size`: an integer (type `size_t`) for the size
- `capacity`: an integer (type `size_t`) for the *capacity*.

The *capacity* is the size of the underlying array, pointed to by `data`. The capacity is guaranteed to be at least as big as `size`, which gives the number of elements actually in use by the vector. Here is an illustration of what one might look like in main memory.

[8] It is easy to make mistakes, e.g., when copying strings of potentially different lengths. This was in fact the source of many early software security vulnerabilities.

[9] Even further behind the scenes, one can see that the STL vector (as distributed with GCC 8.2) does not actually store integers for the size and capacity, but rather stores pointers for the last element, and the end of the allocated storage, and computes size and capacity when needed.



We conclude by mentioning a few consequences of this array-based implementation.

Growing the vector. Note that the `data` pointer might change as the vector is used: once the current capacity is exhausted, the underlying array may have to be relocated to a bigger home, since

1. An array must be in a **contiguous** block of memory, and
2. The memory cells right after `V[V.size()-1]` might be occupied by something else already.

As you can imagine, this move could be costly—it would involve copying every element of the vector to new memory. Since this operation is so costly, the vector was designed so as to do it infrequently. Let's take a look at how a vector grows.[10]

```
void capacitytest()
{
    /* watch how the vector grows: */
    printf("testing capacity growth...\n");
    vector<int> V;
    size_t savedCapacity = V.capacity();
    printf("initial capacity: %lu\n",savedCapacity);
    for (size_t i = 0; i < 20; i++) {
        V.push_back(i);
        if (savedCapacity != V.capacity()) {
            savedCapacity = V.capacity();
            printf("size=%lu   \tcapacity=%lu\n",V.size(),savedCapacity);
        }
    }
}
```

[10] Your computer may behave a little differently, but the capacity will always grow by some multiplicative factor.

```
/* output:
testing capacity growth...
initial capacity: 0
size=1          capacity=1
size=2          capacity=2
size=3          capacity=4
size=5          capacity=8
size=9          capacity=16
size=17         capacity=32 */
```

The numbers on the left show the size which triggered a growth in capacity. Each time more space is needed, the capacity seems to *double*, rather than growing by some additive amount. Let's do a quick bit of analysis to see why this makes sense. Consider a sequence of n consecutive calls to the `push_back` function. We'll examine the cost of performing these operations while growing the capacity according to a few different strategies. If the space is available, `push_back` is rather cheap, so let's focus on the cost of all the copying. Call this quantity $C(n)$, measured in terms of the number of values that must be moved. First let's consider the extreme opposite strategy, in which we grow the array by one element each time. In the worst case, each time we perform this operation the array has to find a new home. Thus, if starting from an empty vector, the total cost of all the copying would be

$$C(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

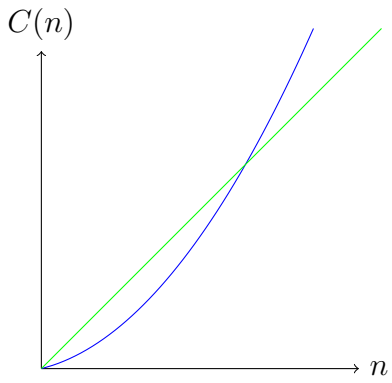
That seems like a lot of work! You would hope that adding n things to a vector would cost $\approx n$ steps, and not $\approx n^2$. However, if we use the strategy suggested by our program's output and double the capacity each time, the cost drops dramatically. Notice that we perform a copy just past each power of 2, or when $n = 2^i + 1$. A bit of thought reveals that the number of times we would have to copy the array during the sequence of `push_back` calls is $\lfloor \log_2(n-1) \rfloor + 1$. Since the i-th copy moves 2^i elements (indexing from $i = 0$), we see that

$$C(n) = \sum_{i=0}^{\lfloor \log_2(n-1) \rfloor} 2^i = 2^{\lfloor \log_2(n-1) \rfloor + 1} - 1$$

so that

$$n - 1 \leq C(n) \leq 2n - 3.$$

Much better! But perhaps you are wondering if the original additive strategy might be salvaged by growing the capacity by a bigger constant, say by adding 10 to the capacity each time instead of just 1? In general, this doesn't really help. This would reduce the work by around a factor of 10, but there is still a *quadratic* dependence on n, meaning this strategy will burn us in the long run. Even if we chose a large step for the first strategy, and a small factor for the second, the graph would still look like this at some point, with the second strategy (shown in green) winning:



Lastly, note that we could of course take an even more aggressive strategy (allocating even more memory when the capacity is reached), but then we are increasing the expected amount of wasted space, and without really helping performance: the cost of tacking on the elements to the end of the array would begin to dwarf the cost of the copying. Using a small constant factor strikes a nice balance.

Inefficient operations on vectors. Another consequence of the array-based implementation is that certain operations are inherently inefficient. In particular, there is only *one end* of a vector for which insertion and removal are efficient: the "back" (the end with the highest indexes). If you think about it, to add or remove something from the beginning or middle of the vector will involve shifting all elements after it forward (insertion) or backward (removal). For a large vector, this may be a nontrivial computation. Nevertheless, the STL implementation does provide functions for such operations: namely `insert` and `erase`. If you find yourself using these often, perhaps `vector` is not the right data structure. We'll learn others soon enough, but if the only issue is that you need to insert and remove things frequently from *both ends*, there is a nice container called the `deque`.