v20190913-0345

# Chapter V: Vectors (take 1)

One of the most important programming concepts is that of the "collection"—a way for a single value to contain multiple other values. Eventually, you will run across other kinds of collection in C++, but our first will be collections that act like lists, which in C++ will be called *vectors*.

The two central facts about lists in general, and specifically about the `vector` types in C++, are that

- they can hold multiple things, with no theoretical maximum size, and
- we can go through them in order.

That is, it may be that we only have five or 42 things in a particular `vector`, but there's nothing about the concept of a `vector` that prevents us from having five hundred or five billion elements, depending on what our data happens to look like. And when we process those elements, we can meaningfully think about the "first thing" or "seventh thing" and can predictably and repeatably process them in order.

## V.1 Making a vector

For now, we'll restrict ourselves to making `vector` values directly in the declare-and-init format. (We'll see other ways to process them, including reading them in from the user, in a few more chapters.) In the same way that we can declare-and-init a variable of one of the simpler types, e.g.

```
string drink = "coffee";
```

we can declare a `vector`-type variable and give its initial value:

```
vector<string> drinks = { "coffee", "tea", "water", "soda" };
```

The syntax of both these lines is exactly the same: first the type of the variable, then the variable name, a single equal sign, and the value used to init the variable. Similarly:

```
  vector<int> lengths = { 3, 72, 5, 1, 5, 54, 3, 28, 154 };
  vector<double> weights = { 4.0, 5.8, 2.75, 3.1, 12.5 };
```

The statement can run to multiple lines if it needs to (just like any other statement):

```
  vector<string> rainbow = { "red", "orange", "yellow",
                    "green", "blue", "indigo", "violet" };
```

And the value need not contain any elements at all!

```
  vector<string> zeroWords = { };
  vector<int> noNums = { };
```

Lists can be empty, after all.

### V.1.1 Required header

Vectors are standard in C++, but you need to explicitly tell the compiler you'll be using them. At the top of your program file, add the line

```
  #include <vector>
```

in addition to any other `#include` lines that you may have up there, in any program where you'll be using vectors.

### V.1.2 What's the deal with the <stuff> after the word `vector`?

Previously, we've seen types like `int` or `string`, which are complete and entire types by themselves. An `int` is an `int`, representing an integer value within a defined range and represented by a particular number of bits. But as we've already seen, saying a value is a `vector` doesn't fully specify its type, because there is also a question about the type of the *contents* of the collection. A vector... of what?

We say as a result that `vector` is not quite a complete type, but rather a *type template* or a *templated type*, which requires a *template argument* to identify the type of the contents. The details of this are not important right now, just remember: when declaring a `vector` variable, always follow the word `vector` with the type of the contents, in angle brackets,[1] as shown above. When reading code out loud or

[1]We use the term "angle brackets" here because they're enclosing something much like square brackets or curly brackets might, but on the keyboard they're exactly the same as the less-than and greater-than signs.

otherwise speaking about them, we read the content type with the word "of", so that `vector<int>` is read as "vector of int", and so on.

Omitting the content type may not always be a problem. In some cases, the type of the contents might be obvious:

```
vector nums = { 3, -1, 0, 42, 7 };  // Error?
```

Indeed, since the C++17 standard, the above line would not give an error message, or even a warning, correctly inferring that the content type must be `int` since all of the contained values are `int`s. (Compilers on older standards would give an error about a missing template argument.) However, even on a current compiler it's a good idea to habitually provide the content type:

```
vector<int> nums = { 3, -1, 0, 42, 7 };  // Better
```

Because sometimes the type can't be inferred, even on a modern compiler:

```
vector empty = { };  // ERROR: no way to infer content type
```

And the inference system can't read our minds—this might be intended as a `vector<double>` but some of the contents are int literals:

```
vector moreNums = { 5, 7.3, 9 };  // ERROR
```

But if we are explicit it works fine:

```
vector<double> moreNums = { 5, 7.3, 9 };  // Works as expected
```

In no situation does C++ permit us to mix-and-match the types of elements in a vector:

```
vector<???> mixed = { 3, 2.5, "Eli", 23, "Audra" };  // ERROR
```

In this sense, C++ vectors are said to be *homogeneous*, i.e. all the contents in any particular vector must be of the same type.

### V.1.3 Review

1. Write a declare-and-init statement for a vector containing the names of the people that live in your apartment or suite or house.
2. Write a declare-and-init statement for a vector containing the ages of the members of your immediate family.

## V.2 Processing its contents: loops

In real life, when we are giving someone simple instructions about processing a list, we might typically do so using the word "each" to stand in for each value in turn:

> *Buy each of the items on this list.*

> *Read out loud each of the names on the list.*

If the instructions for what to do with each item are more complex, we typically put the "each" first and refer back with the pronoun "it" (sometimes with additional logic):

> *With each of the words on this word list,*
> *write* ***it*** *down and*
> *say* ***it*** *out loud.*

> *For each of the items on this grocery list,*
> *check our pantry for* ***it*** *and*
> *if we are out of* ***it****,*
> *buy* ***it*** *at the store.*

This kind of pattern in natural language points the way to how we formalise processing lists of stuff in programming languages. The main difference is that a word like "it" works well for humans who have a lot of context (and can ask questions to clarify misunderstanding), but in a programming language we would like to be more precise by introducing a variable name for the "it" we'll be referring to in the instruction. And, since C++ doesn't want us to introduce new variable names without giving them a type, we will specify the type of the variable. Thus:

```
vector<string> words = { "correct", "horse", "battery", "staple" };
for (string word : words)
{
  cout << word << endl;
}
```

Here, the actual repetition instruction might be paraphrased as "for each word in the word list, print **it** out." Indeed, when we do read it out loud, we would typically actually begin, "for each `string word` in `words`...". Though the word "each" is not explicitly written in the C++, this kind of repetition is widely known as a *for-each loop*. The *loop header*, which is the line with the `for` on it, declares a local variable that will first name the first value in the list (and execute all the instructions in the block), then it will name the second value in the list (and execute all the instructions in the block), and so on. The type of this *loop control variable* should be the same as the type of the contents of the vector.

Here is another example of a for-each loop, which in this case prints out each element of the vector along with its square:

```
vector<int> nums = { 3, 0, -2, 5, 11 };
for (int n : nums)
{
  cout << n << "  " << n*n << endl;
}
```

Its output would be:

```
3  9
0  0
-2  4
5  25
11  121
```

Just as with `if` and `else` statements, the `for` controls the entire block that immediately follows it, which can contain multiple statements.

### V.2.1 Review

1. Make a vector of the names of the towns you and your friends live in, and a for-each loop that prints them all out.
2. What would be the output of this program fragment?

   ```
   vector<double> temps = { 0, 20, 25, -40 };
   for (double ctemp : temps)
   {
     cout << ctemp << "   " << ctemp * 9 / 5 + 32 << endl;
   }
   ```

## V.3 A few more problems

1. Write a program that has a vector variable with many `int` values in it, and then two loops: the first prints all the negative numbers in the vector, and the second prints all the positive numbers. (Hint: remember that blocks can contain *any* other statements, including other control statements!)