Chapter

16

The *vector* Library Type

Variables are named 'boxes' that hold a value. But, when you want to manage **a collection** of similar and **closely related** values, using individual, named variables is unwieldy. Imagine using named variables to track the scores for each student in this section of CS150; it would be nigh-on impossible.

For instance, to print the statistics for one exam, for one section, you'll need to write a function like this:

```
void printClassStatistics(istream& grades)
{
    // Grade for each student
    double s01, s02, s03, s04, s05, s06, s07,
        s08, s09, s10, s11; // And so on . . . to s45
    // Read grades from stream
    grades >> s01 >> s02 >> s03 >> s04 >> s05 >> s06
        >> s07 >> s08 >> s09 >> s10 >> s11; // and so on
}
```

Now, imagine how l**ong and unwieldy** it would become when it came time to add the scores for a quiz. You'd need a separate statement for each student variable.

If you think "**There must be a better way!**" you're right; **there is**.

Meet the *vector*

The solution is **the C++ library** class named **`vector`**. The most used of the standard library's **collection classes**., a **`vector`**:

- Stores **multiple variables**, of any type, in a list.
- Each variable (or **element**) must be **exactly the same type**. Just as you can't put eggs in a muffin tin or bake muffins in an egg crate, you can't put a **`string`** in a **`vector`** of **`int`**; we say a vector is a **homogenous** collection.

Contiguous Allocation

When you create a `vector`, its **elements** are stored right next to each other in memory; we say that the **elements are contiguous** and that the collection is **linear**. Each element uses **exactly the same amount of memory**, which allows your compiler to locate an individual element instantaneously, using simple arithmetic, rather than by searching.

Indexed Access

When creating a `vector`, you give the **entire collection a name**, just like you would any other simple variable. I think of it like the name of a **neighborhood** like Newport Heights.

In most neighborhoods, houses don't have names. Instead, you find a particular house by referring to its **street number**. Similarly, you access `vector` elements by specifying an **index** or **subscript** representing its position in the collection. Just like with the `string` type, the first element has subscript `0`, the next `1`, and so on.

Why Start with Zero?

Should numbering start with zero or one? If you think of a subscript as a **counting number**—#1, # 2, etc...—then zero-based subscripts make no sense. But subscripts are not counting numbers, but **an offset** from the beginning of the collection.

If you tell the compiler to retrieve `v.at(5)`, you are asking it to go to start of the `vector` named `v`, then "walk past" five elements and bring you the element it finds there. If you tell it to access `v.at(0)`, the compiler knows it doesn't have to skip any elements at all, and it brings you the first element.

Creating *vector* Objects

All **library collection classes specify the kind of values they contain** by including the type name in angle brackets following the class name. For example:

- `vector<int>` represents a `vector` that contains integers
- `vector<Card>` is a `vector` of playing cards (a user-defined type)
- `vector<string>` is a `vector` containing `string` objects.

The type in angle-brackets is the **base type**. Classes with a base-type specification are called **parameterized classes**, which, in C++ are implemented with **class templates**.

The types `vector<int>`, `vector<Card>`, and `vector<string>` classes are **independent classes** that share a common general structure. Just like **function templates**, the **`vector` template** can create a whole family of classes, in which the only difference is what type of value the `vector` contains.

Creating *vector* Variables

To use the standard collections, **include the appropriate header** (`<vector>`). The `vector`, like the `string` class, is in the `std` `namespace`.

Creating and using a `vector` object is similar to creating and using an `int`:

```
int n;                  // create an integer
vector<int> iVec;       // stores integers
vector<double> dVec;    // stores doubles
vector<string> sVec;    // stores strings
```

The variable, `iVec` is a `vector` of integers. There is no separate instantiation step.

You cannot create a `vector` of `references`, a vector may be a reference.

```
vector<const Star&> v1;  // Illegal
const vector<Star>& v2 = ...;  // OK
```

Initialization

A newly-created `vector` is **empty**, by default; it contains no elements. To create a sized `vector`, specify the initial size (**in parentheses**) when the `vector` is created. For example, to create a `vector` that initially holds fifteen elements, write:

```
vector<int> iVec(15);
```

This only changes the starting size; you may add additional elements later. All elements are **default-initialized**. For primitive types, such as `int`, that means they are set to `0`. For class types, such as `string`, each element is initialized by its **default constructor**.

Value Initialization

In some cases, you want to initialize all elements with **a value other than zero**. Suppose you want five copies of the string "`(none)`" or twenty copies of the value `137`. To do this, **specify both** the number and default value for the elements, like this:

```
vector<int> v137(10, 137);
vector<string> vNone(5, "(none)");
```

This syntax is **only legal** when initially **creating** a `vector`. In addition, you **must use parentheses**; if you use braces, something else will happen.

List Initialization

In C++11 (or later), you can specify a **initial list** of values. Write the values, separated by commas, and **surrounded by braces**. As mentioned, **this doesn't work in C++ 98**.

```
vector<string> months{"Jan", "Feb", "Mar"};
```

Copy Initialization

Lastly, you can initialize one `vector`, using an existing `vector`. The new `vector` is a **completely independent copy** of the original, not an alias, as in Java. Here's how:

```
vector<int> v1{. . .};
. . .
vector<int> v2(v1);
```

Element Access

The variables stored inside a vector are called its elements. To access an individual element, you use its **subscript** (or **index**). This is called **selecting** the element. To **select** an element, pass the subscript as an argument to the `vector` `at()` member function, or, placing the subscript in **square brackets** after the `vector` name.

```
1  cout << v[1] << endl;    // subscript operator
2  cout << v.at(1) << endl;    // at member function
```

Accessing the second element in v.

For instance, you can print element `1` in `v` by using either of these. Both the **subscript operator** and the `at()` member function **return a reference** to the selected element. Both may appear on either side of an assignment.

Many programmers prefer the square-brackets **subscript operator** because it is similar to the array operations which the `vector` emulates. However, if you use the square-bracket subscripts, your program will **do no range checking**!

Before you go any further, **read over that last line again**. Most of you probably **cannot imagine** a language that does not do some sort of range checking. Let me illustrate:

```
vector<int> v(4);       // 4 elements
cout << v[4];           // Out of bounds
v[4] = 27;
cout << v[4];
```

You **will not** get a compiler error **or a runtime error**, even though you are accessing an element that is outside of the `vector` bounds.

This is an error, though. Often, `cout` will print the value stored in the location where the fifth element of `v` **would be stored**, if it existed. If that is the case, on your platform, then the assignment will happily store the value `27` in the same location, **regardless of what is currently stored there**.

If you think "Well, that's not so bad!", then what about this?

```
v[1075935] = 27;
```

Again, you'll get **no nice stack trace** like you get with Java, telling you your index is out of bounds. If you **are very lucky**, the O/S will shut down your application rudely with a **segmentation fault**. If you aren't, you will **silently corrupt a portion of your own application**, producing a bug that shows up days, weeks or months later. **Not good**.

Using the *at* Member Function

Fortunately, you can fix that by using `at()`.When you use `at()`, the compiler generates code to check out-of-bound subscripts; you don't have to rely on accidentally stepping on the toes of your operating system to find your errors.

Other than the slight performance hit, I can't think of any reason not to **always** use `at()` instead of the subscript operator[1]. Combined with C++ **exception handling**, your code will be safer and have fewer bugs.

---

[1] You can also modify the vector class so that subscripts do throw exceptions. That's what Bjarne Stroustrup does in Section 4.4.1.2 of the Tour of C++ (page 104).

Undefined Behavior

Subscripting a vector past its bounds is an error which invokes **undefined behavior**. Undefined behavior means that the compiler is completely free to ignore it (which is what usually happens). However, the compiler is also free to format your hard disk, send your credit-card to China, or make demons fly out of your nose.

C++ also has **implementation-defined** and **unspecified** behavior. Implementation-defined means the compiler **must pick** a particular implementation, and document it, such as the size of an integer. Unspecified means that the compiler **must do something reasonable**, but need not document what it does. The order in which arguments are evaluated when passed to a function is unspecified; it may be different each time.

First and Last

C++11 (and later) added two additional member functions, `front` and `back` that return a reference to the first and last elements. Calling `front` or `back` on an empty `vector` **is undefined**. If you are lucky, the operating system will catch it. If not, then your hard disk may get reformatted, or your bank account emptied. **Don't do it.**

Processing *vector* Objects

Unlike the built-in array type, the size of a `vector` is not fixed; it can grow or shrink at runtime. The `push_back()` member function appends a new element **to the end** of the `vector`. If the `vector` is full, **it is expanded**.

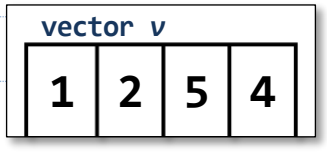
If *v* is an empty `vector` of integers, executing:

```
v.push_back(1);
v.push_back(2);
v.push_back(5);
```

adds these three elements to the end of *v*. Afterwards, *v* looks like the illustration here.

Hou can add additional elements at any time. Later, for example, you could call

```
v.push_back(4);
```



which would add the value `4` to the end of the `vector`, like this.

The `pop_back()` member function removes the element at the end of the `vector` and decreases its size. If the `vector` is empty, calling `pop_back()` is undefined behavior.

```
v.pop_back();
```

After calling `pop_back()` on the previous `vector`, it's back where it started.

Vectors and Loops

The C++11 **range-based** loops work with `vector` as well as `string`. This loop automatically visits every element in the vector:

```
for (auto e : v) {}          // e is a copy
for (auto& e : v) {}         // no copy; may modify
for (const auto& e : v) {}   // no copy; no modify
```

1. The **local variable** `e` is initialized with **a copy** of the next value in `vector` `v`.
2. Here, `e` **is a reference** to the next element in `v`, which **may be** modified.
3. If `v` is a `vector<string>`, you **don't want to make a copy** of each of the elements, and, **you want to prevent any changes**, write this loop.

Later this week, you'll learn how to work with **iterators** and the built-in algorithms from the standard library, which are often an alternative to using loops.

Counter-controlled Loops

The **general pattern** for **manually** iterating through a `vector` looks like this:

```
for (size_t i{0}, len{v.size()}; i < len; ++i)
{
    // Process vector elements here
}
```

Some notes about this loop:

- Instead of calling `v.size()` each time in the loop, call it once and s**ave the value in a variable**; your loop initializer will thus have **two** variables.
- Use `size_t` to avoid the lengthy declaration of `size_type`.
- **At all costs** avoid comparing an `int` to the value returned from `v.size()`. **Mixing signed and unsigned numbers is error prone**.

Right now, let's look at a few common `vector` algorithms.

Common *vector* Algorithms

The real advantage a **vector**, as opposed to individual discrete variables, is that it allows you to apply the same processing to **all of the elements** by using a loop. We can divide this processing into two kinds:

- Algorithms that **need only to read** the values contained in the **vector**. These algorithms solve many counting and calculating problems.
- Algorithms that **may modify** the elements of the **vector** as it is processed. This includes initialization, sorting, and otherwise rearranging items. It also includes algorithms where the position of the elements in the vector is significant or must be noted.

We'll use the **range-based *for*** loop whenever possible. While you always **can** use the counter-controlled ***for*** loop if you like, it's just more work.

Counting Algorithms

Often, you'll need to **count the number of elements** which meet a **particular condition**. How many negative numbers exist? How many numbers between one and a hundred? How many short words? All those are counting algorithms. Here's some pseudocode:

```
counter <- 0
examine each item in the collection
    if the item meets the condition then
        count the item
```

It takes longer to describe this in English than it does to write it in C++. Here's a loop that **counts the positive numbers** in a **vector** named **v**:

```
int positive{0};
for (auto e : v)
{
   if (e > 0) ++positive;
}
```

Cumulative Algorithms

These are the algorithms that **accumulate** or **compute a running sum**. These algorithms include averaging and more complex algorithms like standard deviation and variance. Here is the pseudocode for computing an average:

```
counter <- 0
accumulator <- 0
examine each item in the collection
    if the item meets the condition then
        count the item
        add the item to the accumulator
if the counter is > 0 then
    average <- accumulator / counter
```

Here's a loop that calculates an **average daily temperature** from a list of readings.

```
double sum{0.0};
for (auto t : temperatures)
{
    sum += t;
}
double avg = sum / temp.size(); // nan if no elements
```

Because `sum` is a `double`, this loop sets `avg` to `nan` if there are no elements in the `vector`, using that as an **error code**. If both were `int` then the program would crash from the division by zero. In addition, since it counts **all** of the readings, you don't need a counter, but can use the `vector` size instead.

Processing vector<string>

Here's a second loop which computes the **average word size** in a `vector<string>`. Because you don't want to make unnecessary copies of each element, nor to inadvertently modify an element, the loop variable is `const string&`.

```
double sum{0};
for (const string& w : words) { sum += w.size(); }
double avgWordSize = sum / words.size();
```

Extreme Values

An **extreme value** is the **largest** or **smallest** in a sequence of values. Here's the algorithm for largest. The algorithm for smallest is similar:

```
largest <- first item
examine each item in the collection
    if the current item is larger than largest then
        largest <- current item
```

Here's an example finding the highest temperature in the readings you saw before:

```
auto largest = temps.front();
for (auto temp : temps)
{
    if (temp > largest)
    {
        largest = temp;
    }
}
```

This will **fail** if there are no items in temps; you should **guard the loop** with an `if`. Also, using the range-*for* loop is slightly less efficient than it could be, since it examines the first element twice. Using a traditional counter-controlled *for* loop is only slightly more efficient.

If you are finding the **largest** **for a condition**, then the element found at the `front()` will not necessarily be the largest. Instead, set `largest` to a very small value and check both your condition and for `largest` in the `if` statement.

Modifying Algorithms

When the `vector` elements may be modified, or where the positions of the elements is important, the *for* loop is the loop of choice, because the `size()` member function creates a natural bounds for a counter-controlled loop, and because the loop index can do double-duty as the subscript to access the `vector` elements.

The elements in a `vector` often need to be rearranged for a variety of reasons. You may want to **sort** the names in a list, **update** the prices in a price list, randomly **shuffle** the cards in a deck, or **reverse** the characters in a `string`.

Consider this problem: you have a big glass globe filled with 50 "lotto" balls. Each ball is numbered. You want to **pick three of the balls** for a lottery game called "Pick 3 Lotto".

- The numbers have to be **randomly** selected between 1 and 50 (inclusive)
- No number can appear twice

You may be tempted to start with this code, using the **rand()** function from **<cstdlib>**:

```
int n1{1 + rand() % 50};
int n2{1 + rand() % 50};
int n3{1 + rand() % 50};
```

Unfortunately, this generates duplicates, (that is, 2 of the 3 numbers will be the same), **about 8% of the time**, which is an impossibility in the game you're trying to simulate. (That's why you can't use this method for selecting cards from a deck of cards.)

Instead, the best way to solve this problem is to put all of the lottery balls (numbers) into a vector, **shuffle** the **vector**, and then pick the first three (randomly ordered) elements.

Filling a Vector

You can automatically **fill a vector with any value** you like when you create it. To fill a **vector** with a sequence of **different values** when that sequence is dependent on the loop counter, use a counter-controlled loop like this:

```
const int NUMBERS = 50;
vector<int> lottery(NUMBERS);
for (int i{0}; i < NUMBERS; ++i)
{
    v.at(i) = (i + 1);
}
```

Shuffling

Which brings us to the question: **how do you shuffle the numbers in a list**? The algorithm, known as the **Fisher-Yates** or **Knuth** shuffle, works like this:

1. Take the **last** ball in the **vector** (or card in the deck), and exchange it with any other ball. After this exchange, this ball will never be swapped again. It will also be guaranteed **not** to be itself.
2. Then, take the next-to-last ball, and exchange it with any of the remaining.

Continue on until the first ball has been swapped. Here's the shuffle algorithm in code:

```
for (size_t i = lottery.size(); i > 0; --i)
{
    size_t j = rand() % i;

    int temp = lottery.at(j);
    lottery.at(j) = lottery.at(i);
    lottery.at(i) = temp;
}
```

Vectors and Functions

A **vector** is a **library type,** which means you should follow the same rules for passing parameters as you learned for the **string** library type:

```
int count(const vector<int>& v) ... // input parameter
void mod(vector<int>& v) ... // output or input-output
int bad(vector<int> bad) ... // By value. DON'T
```

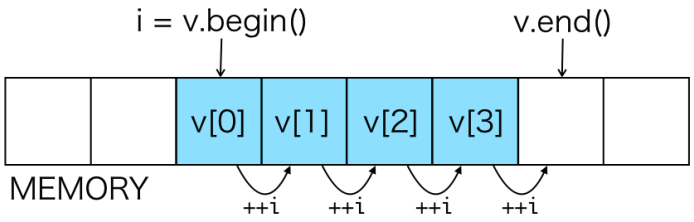
Pass **by reference** for output parameters or **const reference** for input parameters. **Never pass by value**, since that makes a copy of each element in the **vector** when you call the function.

Iterators & Algorithms

An **iterator** is a way of specifying a position inside a container, regardless of what kind of container you are using; subscripts only apply to array-like, contiguous containers. The library has several functions that create and manipulate iterators:

- **begin(c)** returns an iterator pointing to the **first element** in a collection. Most collections also have a member function named **begin()**.
- **end(c)** returns an iterator to the position just **past the end** of the collection. As with **begin()**, most collections also have an **end()** member function.
- **cbegin(c)** and **cend(c)** return **constant iterators**, used when you intend to look at, but not change, the elements in a collection
- **++iter moves the iterator** named **iter** to the next element
- ***iter retrieves the element** that **iter** is "pointing" at. This is called **dereferencing the iterator**, and is the same syntax which we'll use with pointers later in the semester.

Here's a diagram that illustrates these relationships with a vector.



Note particularly that the **end()** iterator is not included in the range of valid elements. This is called a **half-open interval**, which is written as **[begin, end)**. Iterators have long and confusing type names. Avoid that by using **auto** like this:

```
vector<int> v{1, 2, 3};
for (auto i = cbegin(v); i != cend(v); ++i)
{
    cout << *i << " ";
}
cout << endl;
```

In addition to using `++` to move an iterator forward, you can use `--` to move it backward, and, you can use addition and subtraction to move it by several positions. For instance, `begin(v) + 2` is an iterator that points to `v[2]` in the illustration above.

Inserting & Erasing

You may **insert an element** at an arbitrary position in a vector by using an iterator. Given the vector `v`, you can insert `9` into the second position like this:

```
vector<int> v{1, 2, 3};
v.insert(begin(v) + 1, 9);
```

Because this changes the vector, you must use `begin(v)`, not `cbegin(v)`. When an element is inserted, all of the elements after it are shifted towards the end of the vector by one to make room for the new element.

You can `insert()` an element at the `end()` of a `vector`, but if the insert is past the end, the result **is undefined**.

Erasing Elements

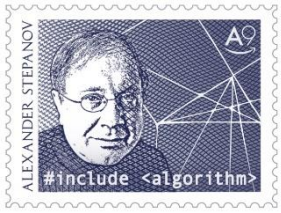
The `pop_back()`, `erase()` and `clear()` **remove** elements. Here's how they are used:

```
vector<int> v{1, 2, 3, 4, 5, 6};
v.pop_back();                       // [1, 2, 3, 4, 5]
v.erase(begin(v));                  // [2, 3, 4, 5]
v.erase(begin(v)+1, begin(v)+3);    // [2, 5]
v.clear();                          // []
```

Notice that if you call `erase()` with one iterator, **only that element** is erased. If you call it with **two iterators**, then all of the elements in the half-open range are erased: that is, the elements from the begin iterator, up to, but not including the end iterator.

The `erase()` member function returns an iterator to the element **following** the last element which was erased.

Introducing the STL



One part of the C++ standard library is known as the STL. The STL is a collection of data structures and algorithms developed at Hewlett Packard in the 1990s by computer scientist Alexander Stepanov. In 1998, the STL was integrated into the C++ standard library.

The classes in the STL are **generic classes**, built using **templates**. Its containers, like `vector`, can **store objects of any type**, and its algorithms work on all containers. Instead of one `count()` function for `vector` and another for `list`, one function works for both.

Building the generic STL library on templates makes it **flexible** and high-performance, better than what you could build yourself. Having a standard library of data structures frees you from constantly **"reinventing the wheel"**, making your code more reliable.

The standard library has a number of algorithms (in the header `<algorithm>`) that are designed to simplify common programming tasks. In general, you should prefer calling a standard algorithm to hand-writing your own loops.

Count Matches

The `count()` algorithm returns the number of items that match a given value.

```
vector<int> v{1, 4, 4, 5, 4, 7, 9, 3};
auto matches = count(cbegin(v), cend(v), 4);
```

Here, matches will be set to 3, since the value 4 is found three times. Notice that the example uses `cbegin()` and `cend()`, since the algorithm **will not modify the elements**. Since that's the case, you could use `begin()` and `end()` as well.

Count for a Condition & Lambdas

To count for a condition, such as **all of the odd numbers**, you need to use the `count_if()` algorithm instead of `count()`. Then, you'll need to write a **predicate function**, and pass the function name as the third argument. Here's an example:

```
bool isOdd(int n) { return n % 2 == 1; }
. . .
vector<int> v{1, 4, 4, 5, 4, 7, 9, 3};
auto odds = count_if(cbegin(v), cend(v), isOdd);
```

This loop goes through the list, passing each element to `isOdd()`. If the function returns `true`, then it counts that element.

You can also make your code a little shorter, by replacing the named function with an **anonymous, inline function**, called a **lambda**. A lambda:

- Begins with a set of brackets `[]`. These take the place of the keyword **lambda** used in Python, or the keyword **function** used in JavaScript.
- Followed by a parameter list.
- Followed by the body of the function.

Here's a similar example, this time **counting the even numbers**, using a lambda in place of a named function.

```
vector<int> v{1, 4, 4, 5, 4, 7, 9, 3};
auto evens = count_if(cbegin(v), cend(v),
    [] (int e) { return e % 2 == 0; } // lambda
);
```

Lambdas can also be **stored in variables** so you can reuse the same function:

```
auto isEven = [] (int e) { return e % 2 == 0; };
```

Extreme Values

You can find the **extreme values** in a range by using **`min_element()`**, **`max_element()`** or **`minmax_element()`**. These algorithms **return an iterator**, so you must dereference the iterator to get the value.

Here's an example showing the use of all three functions. Notice, the you can **apply the dereferencing operator** to the function when you call it to get the value, as done here for **`min_element()`**, or, you can save the iterator itself, and dereference that.

```
vector<int> v{1, 4, 4, 5, 4, 7, 9, 3};
auto little = *min_element(cbegin(v), cend(v)); // deref
cout << "The smallest is " << little << endl;

auto big = max_element(cbegin(v), cend(v)); //iterator
cout << "The biggest is " << *big << endl;// dereference
```

The **`minmax_element()`** function returns a structure (a **`std::pair`**) of iterators with members **`first`** and **`second`**. Here's how that works:

```
auto both = minmax_element(cbegin(v), cend(v));
cout << "Smallest is " << *both.first
    << ", largest->" << *both.second << endl;
```

There are more than 100 pre-written algorithms in the standard library. In general, using an algorithm will be simpler, less error prone, and higher-performing that writing your own loops. We'll be covering more of them as the semester goes by.

You can see a list of all of the algorithms online at cppreference.com.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.