Why Vectors

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 student01, student02, student03, student04,
        student05, student06, student07, student08, student09,
        student10, student11; // And so on . . . to student45
    // Read grades from stream
    grades >> student01 >> student02 >> student03 >> student04
        >> student05 >> student06 >> student07 >> student08
        >> student09 >> student10 >> student11; // and so on
}
```

Now, imagine how **long 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**.

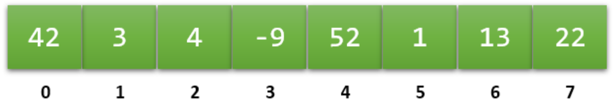
Meet the *vector*

The solution this problem is to use **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 carton, you can't put a `string` in a `vector` of `int`. Because of this, we say a that `vector` is a **homogenous** collection.

When you create a vector, its **elements** are stored right next to each other in memory; we say that the **elements are contiguous**. Each element (except the first and last) has a **predecessor** and a **successor**, so we say that the collection is **linear**.

Each element uses **exactly the same amount of memory** as any other, which allows your computer to locate an individual element instantaneously, using simple arithmetic, rather than by searching.



When creating a vector, you give the **entire collection a name**, just like you would any other simple variable. You access vector elements by specifying an **index** or **subscript** representing its **position** in the collection. The first element has the 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 **offsets** 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 which 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.

vector Variables

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

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

Classes with a base-type specification are called **parameterized** classes, and they are implemented, in C++, using a technique called **class templates**. This means that the classes, **`vector<int>`**, **`vector<Card>`**, and **`vector<string>`** are **independent classes** which each share a common general structure.

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

Creating a **`vector`** variable is similar to creating an **`int`** variable:

```
int n;  // create an integer
vector<int> iVec;     // an empty vector that 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** as in Java, where you might expect to write something like this:

```
vector<int> ivec = new vector<int>(); // Illegal
vector<int&< v1; // Illegal
const vector<int>& = ...;  // OK
```

You **cannot** create a **`vector`** of references, but a **`vector`** may, itself, be a reference (usually when used as a parameter).

vector Initialization

A newly-created **vector** is **empty**; 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 specifies the initial 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 constructed by implicitly calling its **default constructor**.

You may want to initialize all elements with **a value other than zero**. Suppose, for example, you want five copies of the **string** **"(none)"** or twenty copies of the **int** value **137**. To do this, **specify both** the number of elements, and default value for each element elements like this:

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

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

List and Copy Initialization

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

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

Lastly, you can initialize one **vector**, from 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, place the subscript in **square brackets** after the variable name. (The square brackets are called the **subscript operator**).

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

Both the **subscript operator** and the `at()` member function **return a reference** to the selected element, so they may appear on either side of an assignment.

vector Size

You can find out how many elements a `vector` contains by using the `size()` member function. The first element in the `vector` is at subscript `0`, while the last is at `v.at(v.size() - 1)`. C++11 (and later) added two additional member functions, `front()` and `back` which return a reference to the first and last elements.

Calling `front()` or `back()` on an empty `vector` **is undefined**. You can find out if a `vector` contains elements by calling its `empty()` member function, or by checking if its `size()` is greater than zero.

Undefined Behavior

Subscripting a `vector` past its bounds is an error which invokes **undefined behavior** in C++. Undefined behavior means that the compiler is completely free to ignore the error (which is what usually happens). However, the compiler is also free to format your hard disk, send your credit-card credentials to Timbucktu, or, to 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.

Range Checking

Programmers coming from other languages often gravitate to the subscript operator since it is similar in syntax to the array operations which the vector emulates. However, if you use the subscript operator, 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 what happens in C++:

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

You **will not get a compiler error or a runtime error**, as you would in Java or Python, even though you are accessing (and even writing to) 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 would get with Java or Python, telling you that your index is out of bounds. If you **are very lucky**, your operating system 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 this deficiency 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. Combined with C++ **exception handling**, your code will be safer and have fewer bugs.

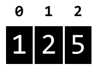
You can also modify the vector class so that subscripts do throw exceptions. That's what Bjarne Stroustrup does in Section 4.4.1.2 of the Tour of C++ (page 104).

Growing & Shrinking

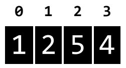
Unlike the built-in array type, the size of a `vector` is not fixed; it can grow or shrink at runtime. The **`push_back()`** member function appends a new element **to the end** of the **`vector`**. If the **`vector`** is full, **it is expanded**. Here's an example:

```
vector<int> v;       // size is 0
v.push_back(1);      // size is now 1
v.push_back(2);      // size is now 2
v.push_back(5);      // size is now 3
```

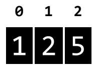
If **`v`** is an empty **`vector<int>`**, executing the code above adds these three elements to the end of **`v`**. Afterwards, **`v`** looks like the illustration here.



You 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.After calling `v.pop_back()` on the **`vector`**, its contents and size are back where it started.



Vectors & Loops

The modern C++ range-based loops work with `vector` as well as with `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; cannot 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`. When you modify `e` you are actually modifying the element inside the `vector v`.
3. If `v` is a `vector<string>`, for example, you **don't want to make a copy** of each of the elements. And, if **you want to prevent any changes**, then use this version of the range-based ***for*** loop.

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 the vector elements here
}
```

Some notes about this loop:

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

Next, let's look at a few common vector algorithms.

Common Algorithms

The real advantage a `vector`, as opposed to using 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 several kinds:

- Algorithms that **need only 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.
- Algorithms where the **position** of the elements in the `vector` is significant or must be noted.
- Algorithms which need to process multiple **`vector`**s, using the same index, or algorithms which need to process only some of the elements.

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 are in the `vector`? How many numbers between one and a hundred? How many short words? All those are **counting** algorithms. Here's some pseudocode that explains the algorithm:

```
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 type `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 this loop counts **all** of the readings, you don't need a counter, but can use the `vector` size instead.

Here's another example, 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& word : words)
{
  sum += word.size();
}
double avg_word_size = 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 application of this algorithm which finds the highest temperature in the readings you saw before:

```cpp
auto largest = temperatures.front();
for (auto current : temperatures)
{
    if (current > largest)
    {
        largest = current;
    }
}
```

This will **fail** if there are no items in `temperatures`; 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 counter-controlled ***`for`*** loop is the loop of choice, because the **`size()`** member function creates a natural bounds, 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 lottery balls, which are now randomly ordered.

Filling & Shuffling

You can automatically fill a `vector` with any value you like when you create it by using one of the constructors. 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 kNumbers = 50;
vector<int> lotter(kNumbers);     // sized vector
for (int i = 0; i < kNumbers; ++i)
{
    v.at(i) = (i + 1);
}
```

Once you have the `vector` filled, its time to randomly rearrange the elements, a process called **shuffling**. The best 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 balls.

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 & 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 DO THIS
```

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, and that is very inefficient.