# Vectors

The data we use in our programs usually needs to be organized into one structure. In this chapter we introduce one way to structure data – the vector. With vectors, we can organize multiple data values using one name and perform several computational tasks more easily and efficiently than if the data were stored in multiple variables.

## A Motivating Example

You are a computer science teacher and you've recently given a test to 10 students. You would like to compute the average grade for the test as well as organize the test scores into ascending order so you can easily see the highest and lowest scores.

Here is what the program could look like based on what we know about C++ so far:

#include <iostream>

using namespace std;

int main()

{

const int numStudents = 10;

int grade1 = 81, grade2 = 88, grade3 = 77, grade4 = 92,

grade5 = 91, grade6 = 85, grade7 = 73, grade8 = 100,

grade9 = 84, grade10 = 98;

int totalScore = grade1 + grade2 + grade3 + grade4 +

grade5 + grade6 + grade7 + grade8 + grade9 + grade10;

double average = static\_cast<double>(totalScore) /

numStudents;

cout << "The class average is: " << average << endl;

return 0;

}

The output from this program is:

The class average is: 86.9

The problem with this code is how many times we must write out each variable for assigning grades and for computing the total of all the grades. A better way to write this program is to use a data structure that allows us to store all the grades under one name. One way to do this is to use a vector.

## How to Declare and Initialize a Vector

A *vector* is a data structure that sequentially stores a set of data of the same type. Vectors are what are known as *generic* types, which means that they can work with any data type but you must specify which data type you are going to use when you create them.

In C++, structures that are used for storing data are often called *containers*.

To use a vector, you need to specify the vector header file in the preprocessor directives:

#include <iostream>

#include <vector>

…

The next step is to declare the vector. Here is the syntax template for a vector declaration:

*vector<data-type> vector-name(optional-size, optional-default-value);*

The optional arguments inside the parentheses indicate you can specify a size, which means the initial number of elements to be stored in the vector, and a default value for all the vector elements. These arguments are optional and you can declare a new vector without these arguments.

Here are some examples of vector declarations:

vector<int> grades;

vector<string> names;

vector<double> averages(10, 0.1);

The last declaration demonstrates how to use the optional arguments to specify a size and default value.

One of the major advantages to using vectors is that they can grow and shrink in size, unlike the other primary C++ data structure – the array. This is why it's not a big deal to not specify the initial size of a vector, since you can add new elements to it no matter what the size currently is.

There will be times when you want to initialize a vector with a set of data values. You can do that using an *initializer list*, which is a list of comma-separated values delimited by curly braces. Here are some examples of using an initializer list to initialize a vector:

vector<int> grades = {81, 77, 83, 100, 98};

vector<string> names = {"Cynthia", "Jonathan", "Raymond"};

These are the primary means of initializing vectors in C++. Now let's move on to adding new data to a vector.

## Adding Data to a Vector

Data are added to a vector using the push\_back function. This function takes a data value as an argument and places the data at the back of a vector. Here is the syntax template for the push\_back function:

*vector-name.push\_back(data-value);*

Here are some examples of adding data to a vector:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> grades;

grades.push\_back(88);

grades.push\_back(77);

grades.push\_back(91);

return 0;

}

The three grades now are stored consecutively in the vector, in successive memory locations, with 88 being the first value stored, 77 begin the second value stored, and 91 being the last value stored in the vector. We can view this storage scheme graphically like this:

grades

|  |  |
| --- | --- |
| 0 | 88 |
| 1 | 77 |
| 2 | 91 |

The first column in the table represents the element positions of the vector. The first position of a vector is numbered 0, which may seem odd but is how computer scientists like to order data in data structures (you will see this again in the next chapter when we discuss arrays).

If another grade is added to this vector, it will be stored at position 3.

There are other ways of adding data to a vector but we will stick with just the push\_back function for now.

## Accessing Vector Data

The at function is the first function you can use to access vector values. This function takes an integer as an argument and returns the data value stored at the vector position indicated by the argument.

Here is the syntax template for the at function:

*vector-name.at(integer);*

Here is how we access the data we stored in the grades vector above:

int main()

{

vector<int> grades;

grades.push\_back(88);

grades.push\_back(77);

grades.push\_back(91);

cout << "First grade in vector: " << grades.at(0) << endl;

cout << "Second grade in vector: " << grades.at(1) << endl;

cout << "Third grade in vector: " << grades.at(2) << endl;

return 0;

}

The output from this program is:

First grade in vector: 88

Second grade in vector: 77

Third grade in vector: 91

The program above is somewhat inefficient as we can use a for loop to access the vector data. Here's the code:

int main()

{

vector<int> grades;

grades.push\_back(88);

grades.push\_back(77);

grades.push\_back(91);

for (int i = 0; i < 3; i++) {

cout << "Grade " << i+1 << ": " << grades.at(i) << endl;

}

return 0;

}

The output from this program is:

Grade 1: 88

Grade 2: 77

Grade 3: 91

There is a problem with this program, however. You'll note in the for loop we used the literal 3 in the condition to end the loop. This worked fine because we knew there were only 3 grades in the vector. Many times, though, we won't be sure of the true number of elements in a vector so we'll need a better way to tell the loop to access all the elements of the vector. We'll see how to do this in the next section.

## The size Function

One of the great advantages of vectors is that you can always know how many elements are in a vector because there is a function that keeps track of the vector size. The function is the size function.

Here is the syntax template for the size function:

*int vector-name.size();*

Here is a simple code fragment that demonstrates how the size function works:

vector<int> grades = {91, 87, 93};

cout << "Number of grades: " << grades.size() << endl;

This fragment displays:

Number of grades: 3

Now let's use the size function to rewrite the for loop in the earlier example:

int main()

{

vector<int> grades;

grades.push\_back(88);

grades.push\_back(77);

grades.push\_back(91);

for (int i = 0; i < grades.size(); i++) {

cout << "Grade " << i+1 << ": " << grades.at(i) << endl;

}

return 0;

}

You should always avoid using a literal value when specifying a limiting condition for a loop when working with a vector since vectors can change in size and the size function is always available to compute the number of elements in the vector.

## Accessing Vector Elements with a Range for Loop

In many situations you will want to access all the elements of a vector. You can use a regular indexed for loop to achieve this. However, using indexed loops lead to possible problems if you don't get the limiting condition correct. A better way to access all the elements of a vector is to use a range for loop.

The range for loop uses some advanced iteration techniques (iterators) internally to provide access to the first vector element through the last vector element. All you need to specify in a range for loop is a variable to hold each vector element.

Here is the syntax template for the range for loop:

*for (data-type variable : vector-name) {*

*loop body;*

*}*

Now let's use the range for loop to display all the elements of a vector:

int main()

{

vector<int> grades = {81, 77, 92, 100, 82, 73};

for (int grade : grades) {

cout << grade << " ";

}

return 0;

}

Here is the output from this program:

81 77 92 100 82 73

## Indexed Access to Vector Elements

The elements in a vector can also be accessed by their index address. As elements are added to a vector they are stored via an index position. The first element of a vector is at index position 0, the next element at index position 1, and so on. We can use these indexes to access vector elements.

Index access of a vector is done using the index operator ([]). The syntax template for this task is:

*vector-type vector-name[index];*

The *vector-type* is the data type of the vector.

You can use this index access anywhere you need a vector element, such as inside an expression or in an assignment statements, for two examples. The following short program will demonstrate how to use a for loop to access all the elements of a vector using indexed access:

int main () {

vector<int> numbers = {1,2,3,4,5};

for (int i = 0; i < numbers.size(); i++) {

cout << numbers[i] << " ";

}

return 0;

}

The output from this program is:

1 2 3 4 5

We can use this type of access to display the vector in reverse order, from the last element to the first element. Here's the program:

int main () {

vector<int> numbers = {1,2,3,4,5};

for (int i = numbers.size()-1; i >= 0; i--) {

cout << numbers[i] << " ";

}

return 0;

}

Notice how the for loop is set up. The indexing variable is set to the size of the vector minus 1. This is because, while the size of the vector is 5 elements, the last index position is 4 since the indexing starts at 0.

Here is the output from this program:

5 4 3 2 1

You will not normally use indexed access with vectors. It is better to use the at function when you need access to specific elements, but this technique is available for special circumstances.

## Accessing the First and Last Vector Elements

A primary reason for using a data structure such as a vector is for the special functions they provide. Two examples of this are the front and back vector functions. These functions are used to access the first element of a vector and the last element, respectively.

Here are the syntax templates for these functions:

*data-type vector-name.front();*

*data-type vector-name.back();*

Now let's look at a small example that uses these two functions with a vector of names:

int main()

{

vector<string> siblings = {"Meredith", "Allison", "Mason"};

cout << "The first name is: " << siblings.front() << endl;

cout << "The last name is: " << siblings.back() << endl;

return 0;

}

Here is the output from this program:

The first name is: Meredith

The last name is: Mason

If you continue in Computer Science you will see lots of situations where using the front and back functions play important roles in complex problem solving situations.

## Vector Utility Functions

We have covered the primary functions you need to work with vectors successfully. However, there are a couple functions that can perform useful tasks. These functions are empty and clear.

The empty function is a Boolean function that returns true if the target vector does not have any data in it and false if the vector does have data in it. Here is the syntax template for the empty function:

*bool vector-name.empty();*

This function is useful for checking to make sure a vector isn't empty before performing some task on it. Here is an example of this use of the empty function:

int main()

{

vector<int> numbers;

if (numbers.empty()) {

for (int i = 1; i <= 10; i++) {

numbers.push\_back(i);

}

}

return 0;

}

The program above will only add data to the vector if the vector is currently empty.

This next example only accesses the specified vector element if the vector is not empty:

int main()

{

vector<int> numbers = {2,4,6,8,10};

if (!numbers.empty()) {

int element;

cout << "Which element do you want to access? ";

cin >> element;

cout << "The value at position " << element

<< " is " << numbers.at(element) << endl;

}

return 0;

}

Here is the output from one run of the program:

Which element do you want to access? 2

The value at position 2 is 6

The next useful utility function is clear. This function removes all the elements of a vector and sets its size back to 0. Here is the syntax template for the clear function:

*void vector-name.clear();*

Here is a program that demonstrates how the clear function works:

int main()

{

vector<int> numbers = {2,4,6,8,10};

cout << "Size of numbers vector: " << numbers.size() << endl;

numbers.clear();

cout << "Size of numbers vector after clearing: "

<< numbers.size() << endl;

return 0;

}

Here is the output from this program:

Size of numbers vector: 5

Size of numbers vector after clearing: 0

## Generating Vector Data with Random Numbers

As you design a program that uses a vector to store data, you will want to test your program using data. It's hard, though, to generate enough data to thoroughly test your program on your own. A good solution for numeric vectors is to use random number generation to generate that data.

We’ve covered random number generation already so let's just jump right into it. Here is a program that generates 100 random integers and stores them in a vector:

#include <iostream>

#include <vector>

#include <random>

#include <ctime>

using namespace std;

int main () {

vector<int> numbers;

default\_random\_engine defEngine(time(0));

uniform\_int\_distribution<int> intDistro(0,100);

const int vecSize = 100;

for (int i = 1; i <= vecSize; i++) {

numbers.push\_back(intDistro(defEngine));

}

for (const int n : numbers) {

cout << n << " ";

}

return 0;

}

Here is the output from this program:

0 13 76 46 53 22 4 68 68 94 38 52 83 3 5 53 67 0 38 6 42 69 59 93 85 53 9 66 42 70 91 76 26 4 74 33 63 76 100 36 24 99 72 76 65 7 63 89 27 44 77 48 24 27 36 16 49 90 91 6 91 50 52 32 99 49 26 9 95 7 50 38 27 92 53 46 95 5 76 77 83 12 1 69 87 63 74 73 100 89 23 30 35 51 59 85 41 84 27 41

The output isn't pretty but you can clearly see 100 randomly generated numbers coming directly from the vector.

Use this technique when you need to test a program that uses vector storage.

## Vectors as Function Arguments

Functions can have vectors as parameters for the purpose of computing something with the vector data or even creating a vector. In this section we'll look at examples of both these tasks.

Our first example will use a void function to populate a vector with data. We'll use random number generation to create the data. An important point to note is that the vector must be passed by reference to permanently store the data in the vector.

Here is the function definition and a program that uses the function:

void createVector(vector<int> &vec, int n) {

default\_random\_engine defEngine(time(0));

uniform\_int\_distribution<int> intDistro(0,100);

for (int i = 1; i <= n; i++) {

vec.push\_back(intDistro(defEngine));

}

}

int main () {

vector<int> numbers;

createVector(numbers, 20);

for (const int n : numbers) {

cout << n << " ";

}

return 0;

}

Here is the output from one run of this program:

17 97 1 64 80 2 93 96 88 36 56 10 29 43 14 37 8 68 28 1

Now let's combine this function with another function that computes the average of the numbers in the vector. Here is the function definition and a program that tests the function:

void createVector(vector<int> &vec, int n) {

default\_random\_engine defEngine(time(0));

uniform\_int\_distribution<int> intDistro(0,100);

for (int i = 1; i <= n; i++) {

vec.push\_back(intDistro(defEngine));

}

}

int computeAverage(vector<int> vec) {

int total = 0;

for (int n : vec) {

total += n;

}

return static\_cast<double>(total) / vec.size();

}

int main () {

vector<int> numbers;

createVector(numbers, 20);

for (const int n : numbers) {

cout << n << " ";

}

cout << endl << endl;

double numbersAvg = computeAverage(numbers);

cout << "The average of the vector is: "

<< numbersAvg << endl;

return 0;

}

The output from one run of this program is:

18 2 42 87 75 29 41 15 32 59 48 54 12 90 24 59 95 1 15 64

The average of the vector is: 43

The beauty of writing programs in this manner is that the main part of the program is relatively short, with just a few declarations and a few function calls. An experienced C++ programmer can look at this program and understand what it does without having to read the function definitions. However, if more detail is needed, the functions can be examined to understand exactly what they are doing.

## Glossary

**vector**: a container that stores data in successive memory locations

**container**: another name for a C++ structure that stores data

**generic structure**: a container that can store data of any data type

## Exercises

1. Create a vector of five strings. Display the names in order using a range for loop and backward using an index for loop.
2. Rewrite the program in Exercise 1 using functions to display the vector in order and backwards.
3. Create a vector of ten grades, randomly generated. Write a function that displays the lowest grade in the vector and another function that displays the highest grade in the vector.
4. Create a function that creates a vector of five strings that are entered by the user when the function is called. Create another function that displays the vector element with the most characters. Use the functions in a program.
5. Using a function create a vector of 100 integers. Create a function that determines the median value in the vector. Use the functions in a program.