C++ List

A list is similar to a [vector](#) in that it can store multiple elements of the same type and dynamically grow in size.

However, two major differences between lists and vectors are:

1. You can add and remove elements from both the beginning and at the end of a list, while vectors are generally optimized for adding and removing at the end.

2. Unlike vectors, a list does not support random access, meaning you cannot directly jump to a specific index, or access elements by index numbers.

To use a list, you have to include the <list> header file:

```
// Include the list library
#include <list>
```

Create a List

To create a list, use the list keyword, and specify the **type** of values it should store within angle brackets <> and then the name of the list, like: list<*type*> *listName*.

Example

```
// Create a list called cars that will store strings
list<string> cars;
```

If you want to add elements at the time of declaration, place them in a comma-separated list, inside curly braces {}:

Example

```
// Create a list called cars that will store strings
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Print list elements
for (string car : cars) {
  cout << car << "\n";
}
```

[Try it Yourself »](#)

Note: The type of the list (string in our example) cannot be changed after its been declared.

Access a List

You cannot access list elements by referring to index numbers, like with arrays and vectors.

However, you can access the first or the last element with the .front() and .back() functions, respectively:

Example

```
// Create a list called cars that will store strings
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
// Get the first element
cout << cars.front();  // Outputs Volvo

// Get the last element
cout << cars.back();  // Outputs Mazda
```

Try it Yourself »

Change a List Element

You can also change the value of the first or the last element with the .front() and .back() functions

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Change the value of the first element
cars.front() = "Opel";

// Change the value of the last element
cars.back() = "Toyota";

cout << cars.front(); // Now outputs Opel instead of Volvo
cout << cars.back();  // Now outputs Toyota instead of Mazda
```

Try it Yourself »

Add List Elements

To add elements to a list, you can use .push_front() to insert an element at the beginning of the list and .push_back() to add an element at the end:

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Add an element at the beginning
cars.push_front("Tesla");

// Add an element at the end
cars.push_back("VW");
```

Try it Yourself »

Remove List Elements

To remove elements from a list, use .pop_front() to remove an element from the beginning of the list and .pop_back() to remove an element at the end:

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Remove the first element
cars.pop_front();

// Remove the last element
cars.pop_back();
```

[Try it Yourself »]

List Size

To find out how many elements a list has, use the .size() function:

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.size();  // Outputs 4
```

[Try it Yourself »]

Check if a List is Empty

Use the .empty() function to find out if a list is empty or not.

The .empty() function returns 1 (*true*) if the list is empty and 0 (*false*) otherwise:

Example

```
list<string> cars;
cout << cars.empty();  // Outputs 1 (The list is empty)
```

[Try it Yourself »]

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.empty();  // Outputs 0 (not empty)
```

[Try it Yourself »]

Loop Through a List

You cannot loop through the list elements with a traditional for loop combined with the .size() function, since it is not possible to access elements in a list by index:

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.size(); i++) {
  cout << cars[i] << "\n";
```

```
}
```

The simplest way to loop through a list is with the **for-each** loop:

Example

```
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (string car : cars) {
  cout << car << "\n";
}
```

C++ Stack

A stack stores multiple elements in a specific order, called **LIFO**.

LIFO stands for **Last in, First Out**. To vizualise LIFO, think of a pile of pancakes, where pancakes are both added and removed from the top. So when removing a pancake, it will always be the last one you added. This way of organizing elements is called LIFO in computer science and programming.

Unlike vectors, elements in the stack are not accessed by index numbers. Since elements are added and removed from the top, you can only access the element at the top of the stack.

To use a stack, you have to include the <stack> header file:

```
// Include the stack library
#include <stack>
```

Create a Stack

To create a stack, use the stack keyword, and specify the **type** of values it should store within angle brackets <> and then the name of the stack, like: stack<*type*> *stackName*.

```
// Create a stack of strings called cars
stack<string> cars;
```

Note: The type of the stack (string in our example) cannot be changed after its been declared.

Note: You cannot add elements to the stack at the time of declaration, like you can with vectors:

```
stack<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Add Elements

To add elements to the stack, use the .push() function, after declaring the stack:

Example

```
// Create a stack of strings called cars
stack<string> cars;

// Add elements to the stack
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");
```

The stack will look like this (remember that the last element added is the top element):

Mazda (top element)
Ford
BMW
Volvo

Access Stack Elements

You cannot access stack elements by referring to index numbers, like you would with arrays and vectors.

In a stack, you can only access the top element, which is done using the .top() function:

Example

```
// Access the top element
cout << cars.top();  // Outputs "Mazda"
```

Try it Yourself »

Change the Top Element

You can also use the .top function to change the value of the top element:

Example

```
// Change the value of the top element
cars.top() = "Tesla";

 // Access the top element
cout << cars.top();  // Now outputs "Tesla" instead of "Mazda"
```

Try it Yourself »

Remove Elements

You can use the .pop() function to remove an element from the stack.

This will remove the last element that was added to the stack:

Example

```
// Create a stack of strings called cars
stack<string> cars;

// Add elements to the stack
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

// Remove the last added element (Mazda)
cars.pop();

// Access the top element (Now Ford)
cout << cars.top();
```

Try it Yourself »

Get the Size of the Stack

To find out how many elements a stack has, use the .size() function:

Example

```
cout << cars.size();
```

Try it Yourself »

Check if the Stack is Empty

Use the .empty() function to find out if the stack is empty or not.

The .empty() function returns 1 (*true*) if the stack is empty and 0 (*false*) otherwise:

Example

```
stack<string> cars;
cout << cars.empty(); // Outputs 1 (The stack is empty)
```

Try it Yourself »

Example

```
stack<string> cars;

cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

cout << cars.empty();  // Outputs 0 (not empty)
```

Try it Yourself »

Stacks and Queues

Stacks are often mentioned together with Queues

C++ Queue

A queue stores multiple elements in a specific order, called **FIFO**.

FIFO stands for **First in, First Out**. To visualize FIFO, think of a queue as people standing in line in a supermarket. The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO in computer science and programming.

Unlike vectors, elements in the queue are not accessed by index numbers. Since queue elements are added at the end and removed from the front, you can only access an element at the front or the back.

To use a queue, you have to include the <queue> header file:

```
// Include the queue library
#include <queue>
```

Create a Queue

To create a queue, use the queue keyword, and specify the **type** of values it should store within angle brackets <> and then the name of the queue, like: queue<*type*> *queueName*.

```
// Create a queue of strings called cars
queue<string> cars;
```

Note: The type of the queue (string in our example) cannot be changed after its been declared.

Note: You cannot add elements to the queue at the time of declaration, like you can with vectors:

```
queue<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Add Elements

To add elements to the queue, you can use the .push() function after declaring the queue.

The .push() function adds an element at the end of the queue:

Example

```
// Create a queue of strings
queue<string> cars;

// Add elements to the queue
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");
```

The queue will look like this:

Volvo (front (first) element)
BMW
Ford
Mazda (back (last) element)

Access Queue Elements

You cannot access queue elements by referring to index numbers, like you would with arrays and vectors.

In a queue, you can only access the element at the front or the back, using .front() and .back() respectively:

Example

```
// Access the front element (first and oldest)
cout << cars.front();  // Outputs "Volvo"

// Access the back element (last and newest)
cout << cars.back();  // Outputs "Mazda"
```

Try it Yourself »

Change Front and Back Elements

You can also use .front and .back to change the value of the front and back elements:

Example

```
// Change the value of the front element
cars.front() = "Tesla";

// Change the value of the back element
cars.back() = "VW";

// Access the front element
cout << cars.front();  // Now outputs "Tesla" instead of "Volvo"

// Access the back element
cout << cars.back();  // Now outputs "VW" instead of "Mazda"
```

Try it Yourself »

Remove Elements

You can use the .pop() function to remove an element from the queue.

This will remove the front element (the first and oldest element that was added to the queue):

Example

```
// Create a queue of strings
queue<string> cars;

// Add elements to the queue
cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

// Remove the front element (Volvo)
```

```
cars.pop();

// Access the front element (Now BMW)
cout << cars.front();
```

[Try it Yourself »]

Get the Size of a Queue

To find out how many elements there are in a queue, use the .size() function:

Example

```
cout << cars.size();
```

[Try it Yourself »]

Check if the Queue is Empty

Use the .empty() function to find out if the queue is empty or not.

The .empty() function returns 1 (*true*) if the queue is empty and 0 (*false*) otherwise:

Example

```
queue<string> cars;
cout << cars.empty(); // Outputs 1 (The queue is empty)
```

[Try it Yourself »]

Example

```
queue<string> cars;

cars.push("Volvo");
cars.push("BMW");
cars.push("Ford");
cars.push("Mazda");

cout << cars.empty();  // Outputs 0 (not empty)
```

[Try it Yourself »]

Stacks and Queues

Queues are often mentioned together with Stacks, which is a similar data structure

C++ Deque

In the previous page, your learned that elements in a queue are added at the end and removed from the front.

A deque (stands for **d**ouble-**e**nded **queue**) however, is more flexible, as elements can be added and removed from both ends (at the front and the back). You can also access elements by index numbers.

To use a deque, you have to include the <deque> header file:

```
// Include the deque library
#include <deque>
```

Create a Deque

To create a deque, use the deque keyword, and specify the **type** of values it should store within angle brackets <> and then the name of the deque, like: deque<*type*> *dequeName*.

Example

```
// Create a deque called cars that will store strings
deque<string> cars;
```

If you want to add elements at the time of declaration, place them in a comma-separated list, inside curly braces {}:

Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Print deque elements
for (string car : cars) {
  cout << car << "\n";
}
```

Try it Yourself »

Note: The type of the deque (string in our example) cannot be changed after its been declared.

Access a Deque

You can access a deque element by referring to the index number inside square brackets [].

Deques are 0-indexed, meaning that [0] is the first element, [1] is the second element, and so on:

Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars[0];  // Outputs Volvo

// Get the second element
cout << cars[1];  // Outputs BMW
```

Try it Yourself »

You can also access the first or the last element of a deque with the .front() and .back() functions:

Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars.front();

// Get the last element
cout << cars.back();
```

Try it Yourself »

To access an element at a specified index, you can use the .at() function and specify the index number:

Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the second element
cout << cars.at(1);

// Get the third element
cout << cars.at(2);
```

Try it Yourself »

Note: The .at() function is often preferred over square brackets [] because it throws an error message if the element is out of range:

Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Try to access an element that does not exist (will throw an exception)
cout << cars.at(6);
```

Try it Yourself »

Change a Deque Element

To change the value of a specific element, you can refer to the index number:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
// Change the value of the first element
cars[0] = "Opel";

cout << cars[0];  // Now outputs Opel instead of Volvo
```

[Try it Yourself »]

However, it is safer to use the .at() function:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Change the value of the first element
cars.at(0) = "Opel";

cout << cars.at(0);  // Now outputs Opel instead of Volvo
```

[Try it Yourself »]

Add Deque Elements

To add elements to a deque, you can use .push_front() to insert an element at the beginning of the deque and .push_back() to add an element at the end:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Add an element at the beginning
cars.push_front("Tesla");

// Add an element at the end
cars.push_back("VW");
```

[Try it Yourself »]

Remove Deque Elements

To remove elements from a deque, use .pop_front() to remove an element from the beginning of the deque and .pop_back() to remove an element at the end:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Remove the first element
cars.pop_front();

// Remove the last element
cars.pop_back();
```

[Try it Yourself »]

Deque Size

To find out how many elements a deque has, use the .size() function:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.size();  // Outputs 4
```

[Try it Yourself »]

Check if a Deque is Empty

Use the .empty() function to find out if a deque is empty or not.

The .empty() function returns 1 (*true*) if the deque is empty and 0 (*false*) otherwise:

Example

```
deque<string> cars;
cout << cars.empty();  // Outputs 1 (The deque is empty)
```

[Try it Yourself »]

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.empty();  // Outputs 0 (not empty)
```

[Try it Yourself »]

Loop Through a Deque

You can loop through the deque elements by using a for loop combined with the .size() function:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.size(); i++) {
  cout << cars[i] << "\n";
}
```

[Try it Yourself »]

You can also use a **for-each loop** (introduced in C++ version 11 (2011), which is cleaner and more readable:

Example

```
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (string car : cars) {
  cout << car << "\n";
}
```

Try it Yourself »

Tip: It is also possible to loop through deques with an iterator, C++ Set

A set stores unique elements where they:

- Are sorted automatically in ascending order.
- Are unique, meaning equal or duplicate values are ignored.
- Can be added or removed, but the value of an existing element cannot be changed.
- Cannot be accessed by index numbers, because the order is based on sorting and not indexing.

To use a set, you have to include the <set> header file:

```
// Include the set library
#include <set>
```

Create a Set

To create a set, use the set keyword, and specify the **type** of values it should store within angle brackets <> and then the name of the set, like: set<*type*> *setName*.

Example

```
// Create a set called cars that will store strings
set<string> cars;
```

If you want to add elements at the time of declaration, place them in a comma-separated list, inside curly braces {}:

Example

```
// Create a set called cars that will store strings
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Print set elements
for (string car : cars) {
  cout << car << "\n";
}
```

The output will be:

```
BMW
Ford
Mazda
Volvo
```

Try it Yourself »

As you can see from the result above, the elements in the set are sorted automatically. In this case, alphabetically, as we are working with strings.

If you store integers in the set, the returned values are sorted numerically:

Example

```
// Create a set called numbers that will store integers
set<int> numbers = {1, 7, 3, 2, 5, 9};

// Print set elements
```

```
for (int num : numbers) {
  cout << num << "\n";
}
```

The output will be:

1
2
3
5
7
9

Try it Yourself »

Note: The type of the set (e.g. string and int in the examples above) cannot be changed after its been declared.

Sort a Set in Descending Order

By default, the elements in a set are sorted in ascending order. If you want to reverse the order, you can use the greater<*type*> functor inside the angle brackets, like this:

Example

```
// Sort elements in a set in descending order
set<int, greater<int>> numbers = {1, 7, 3, 2, 5, 9};
// Print the elements
for (int num : numbers) {
  cout << num << "\n";
}
```

The output will be:

9
7
5
3
2
1

Try it Yourself »

Note: The type specified in greater<*type*> must match the type of elements in the set (int in our example).

Unique Elements

Elements in a set are unique, which means they cannot be duplicated or equal.

For example, if we try to add "BMW" two times in the set, the duplicate element is ignored:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "BMW", "Mazda"};

// Print set elements
for (string car : cars) {
  cout << car << "\n";
}
```

The output will be:

BMW
Ford
Mazda
Volvo

Try it Yourself »

Add Elements

To add elements to a set, you can use the .insert() function:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Add new elements
cars.insert("Tesla");
cars.insert("VW");
cars.insert("Toyota");
cars.insert("Audi");
```

Try it Yourself »

Remove Elements

To remove specific elements from a set, you can use the .erase() function:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Remove elements
cars.erase("Volvo");
cars.erase("Mazda");
```

Try it Yourself »

To remove all elements from a set, you can use the .clear() function:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Remove all elements
cars.clear();
```

[Try it Yourself »]

Find the Size of a Set

To find out how many elements a set has, use the .size() function:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.size();  // Outputs 4
```

[Try it Yourself »]

Check if a Set is Empty

Use the .empty() function to find out if a set is empty or not.

The .empty() function returns 1 (*true*) if the set is empty and 0 (*false*) otherwise:

Example

```
set<string> cars;
cout << cars.empty();  // Outputs 1 (The set is empty)
```

[Try it Yourself »]

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.empty();  // Outputs 0 (not empty)
```

[Try it Yourself »]

Loop Through a Set

You can loop through a set with the **for-each loop**:

Example

```
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (string car : cars) {
  cout << car << "\n";
}
```

[Try it Yourself »]

Tip: It is also possible to loop through sets with an [iterator], C++ Map

A map stores elements in "**key/value**" pairs.

Elements in a map are:

- Accessible by keys (not index), and each key is unique.
- Automatically sorted in ascending order by their keys.

To use a map, you have to include the <map> header file:

```
// Include the map library
#include <map>
```

Create a Map

To create a map, use the map keyword, and specify the **type** of both the key and the value it should store within angle brackets <>. At last, specify the name of the map, like: map<*keytype, valuetype*> *mapName*:

Example

```
// Create a map called people that will store strings as keys and integers as values
map<string, int> people
```

If you want to add elements at the time of declaration, place them in a comma-separated list, inside curly braces {}:

Example

```
// Create a map that will store the name and age of different people
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
```

Access a Map

You cannot access map elements by referring to index numbers, like you would with [arrays]() and [vectors]().

Instead, you can access a map element by referring to its key inside square brackets []:

Example

```
// Create a map that will store the name and age of different people
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Get the value associated with the key "John"
cout << "John is: " << people["John"] << "\n";

// Get the value associated with the key "Adele"
cout << "Adele is: " << people["Adele"] << "\n";
```

[Try it Yourself »]()

You can also access elements with the .at() function:

Example

```
// Create a map that will store the name and age of different people
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
```

```
// Get the value associated with the key "Adele"
cout << "Adele is: " << people.at("Adele") << "\n";

// Get the value associated with the key "Bo"
cout << "Bo is: " << people.at("Bo") << "\n";
```

[Try it Yourself »]

Note: The .at() function is often preferred over square brackets [] because it throws an error message if the element does not exist:

Example

```
// Create a map that will store the name and age of different people
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Try to access an element that does not exist (will throw an exception)
cout << people.at("Jenny");
```

[Try it Yourself »]

Change Values

You can also change the value associated with a key:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Change John's value to 50 instead of 32
people["John"] = 50;

cout << "John is: " << people["John"];  // Now outputs John is: 50
```

[Try it Yourself »]

However, it is safer to use the .at() function:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Change John's value to 50 instead of 32
people.at("John") = 50;

cout << "John is: " << people.at("John");  // Now outputs John is: 50
```

[Try it Yourself »]

Add Elements

To add elements to a map, it is ok to use square brackets []:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Add new elements
people["Jenny"] = 22;
people["Liam"] = 24;
people["Kasper"] = 20;
people["Anja"] = 30;
```

Try it Yourself »

But you can also use the .insert() function:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Add new elements
people.insert({"Jenny", 22});
people.insert({"Liam", 24});
people.insert({"Kasper", 20});
people.insert({"Anja", 30});
```

Try it Yourself »

Elements with Equal Keys

A map cannot have elements with equal keys.

For example, if we try to add "Jenny" two times to the map, it will only keep the first one:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Trying to add two elements with equal keys
people.insert({"Jenny", 22});
people.insert({"Jenny", 30});
```

Try it Yourself »

To sum up; values can be equal, but keys must be unique.

Remove Elements

To remove specific elements from a map, you can use the .erase() function:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Remove an element by key
```

```
people.erase("John");
```

[Try it Yourself »]

To remove all elements from a map, you can use the .clear() function:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Remove all elements
people.clear();
```

Find the Size of a Map

To find out how many elements a map has, use the .size() function:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
cout << people.size();  // Outputs 3
```

[Try it Yourself »]

Check if a Map is Empty

Use the .empty() function to find out if a map is empty or not.

The .empty() function returns 1 (*true*) if the map is empty and 0 (*false*) otherwise:

Example

```
map<string, int> people;
cout << people.empty(); // Outputs 1 (The map is empty)
```

[Try it Yourself »]

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
cout << people.empty();  // Outputs 0 (not empty)
```

[Try it Yourself »]

Note: You can also check if a specific element exists, by using the .count(*key*) function.

It returns 1 (*true*) if the element exists and 0 (*false*) otherwise:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
cout << people.count("John");  // Outputs 1 (John exists)
```

[Try it Yourself »]

Loop Through a Map

You can loop through a map with the **for-each** loop. However, there are a couple of things to be aware of:

- You should use the auto keyword (introduced in C++ version 11) inside the for loop. This allows the compiler to automatically determine the correct data type for each key-value pair.
- Since map elements consist of both keys and values, you have to include .first to access the keys, and .second to access values in the loop.
- Elements in the map are sorted automatically in ascending order by their keys:

Example

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

for (auto person : people) {
  cout << person.first << " is: " << person.second << "\n";
}
```

The output will be:

Adele is: 45
Bo is: 29
John is: 32

Try it Yourself »

If you want to reverse the order, you can use the greater<*type*> functor inside the angle brackets, like this:

Example

```
map<string, int, greater<string>> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

for (auto person : people) {
  cout << person.first << " is: " << person.second << "\n";
}
```

The output will be:

John is: 32
Bo is: 29
Adele is: 45

Try it Yourself »

Tip: It is also possible to loop through maps with an iterator, C++ Iterators

Iterators are used to access and iterate through elements of data structures (vectors, sets, etc.), by "pointing" to them.

It is called an "iterator" because "iterating" is the technical term for **looping**.

To iterate through a vector, look at the following example:

Example

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Create a vector iterator called it
vector<string>::iterator it;

// Loop through the vector with the iterator
for (it = cars.begin(); it != cars.end(); ++it) {
  cout << *it << "\n";
}
```

Try it Yourself »

Example explained

1. First we create a vector of strings to store the names of different car manufactures.
2. Then we create a "vector iterator" called it, that we will use to loop through the vector.
3. Next, we use a for loop to loop through the vector with the iterator. The iterator (it) points to the first element in the vector (cars.begin()) and the loop continues as long as it is not equal to cars.end().
4. The increment operator (++it) moves the iterator to the next element in the vector.
5. The dereference operator (*it) accesses the element the iterator points to.

Note: The type of the iterator must match the type of the data structure it should iterate through (string in our example)

What is begin() and end()?

begin() and end() are **functions** that **belong to data structures**, such as vectors and lists. They **do not belong to the iterator** itself. Instead, they are used with iterators to access and iterate through the elements of these data structures.

- begin() returns an iterator that points to the first element of the data structure.
- end() returns an iterator that points to one position after the last element.

To understand how they work, let's continue to use vectors as an example:

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

vector<string>::iterator it;
```

Begin Examples

begin() points to the first element in the vector (index 0, which is "Volvo"):

Example

```
// Point to the first element in the vector
it = cars.begin();
```

[Try it Yourself »]()

To point to the second element (BMW), you can write cars.begin() + 1:

Example

```
// Point to the second element
it = cars.begin() + 1;
```

[Try it Yourself »]()

And of course, that also means you can point to the third element with cars.begin() + 2:

Example

```
// Point to the third element
it = cars.begin() + 2;
```

[Try it Yourself »]()

End Example

end() points to one position **after** the last element in the vector (meaning it doesn't point to an actual element, but rather indicates that this is the end of the vector).

So, to use end() to point to the last element in the cars vector (Mazda), you can use cars.end() - 1:

Example

```
// Point to the last element
it = cars.end() - 1;
```

[Try it Yourself »]()

Why do we say "point"?

Iterators are like "[pointers]()" in that they "point" to elements in a data structure rather than returning values from them. They refer to a specific position, providing a way to access and modify the value when needed, without making a copy of it. For example:

Example

```
// Point to the first element in the vector
it = cars.begin();

// Modify the value of the first element
*it = "Tesla";

// Volvo is now Tesla
```

[Try it Yourself »]()

The auto Keyword

In C++ 11 and later versions, you can use the auto keyword instead of explicitly declaring and specifying the type of the iterator.

The auto keyword allows the compiler to automatically determine the correct data type, which simplifies the code and makes it more readable:

Instead of this:

```
vector<string>::iterator it = cars.begin();
```

You can simply write this:

```
auto it = cars.begin();
```

Try it Yourself »

In the example above, the compiler knows the type of it based on the return type of cars.begin(), which is vector<string>::iterator.

The auto keyword works in for loops as well:

```
for (auto it = cars.begin(); it != cars.end(); ++it) {
  cout << *it << "\n";
}
```

Try it Yourself »

For-Each Loop vs. Iterators

You can use a **for-each** loop to just loop through elements of a data structure, like this:

Example

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Print vector elements
for (string car : cars) {
  cout << car << "\n";
}
```

Try it Yourself »

When you are just reading the elements, and don't need to modify them, the for-each loop is much simpler and cleaner than iterators.

However, when you need to add, modify, or remove elements **during iteration**, iterate in reverse, or skip elements, you should use iterators:

Example

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
// Loop through vector elements
for (auto it = cars.begin(); it != cars.end(); ) {
  if (*it == "BMW") {
    it = cars.erase(it); // Remove the BMW element
  } else {
    ++it;
  }
}

// Print vector elements
for (const string& car : cars) {
  cout << car << "\n";
}
```

[Try it Yourself »]

Iterate in Reverse

To iterate in reverse order, you can use rbegin() and rend() instead of begin() and end():

Example

```
// Iterate in reverse order
for (auto it = cars.rbegin(); it != cars.rend(); ++it) {
  cout << *it << "\n";
}
```

[Try it Yourself »]

Iterate Through other Data Structures

Iterators are great for code reusability since you can use the same syntax for iterating through vectors, lists, deques, sets and maps:

List Example

```
// Create a list called cars that will store strings
list<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Loop through the list with an iterator
for (auto it = cars.begin(); it != cars.end(); ++it) {
  cout << *it << "\n";
}
```

[Try it Yourself »]

Deque Example

```
// Create a deque called cars that will store strings
deque<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Loop through the deque with an iterator
for (auto it = cars.begin(); it != cars.end(); ++it) {
  cout << *it << "\n";
```

```
}
```

[Try it Yourself »]

Set Example

```
// Create a set called cars that will store strings
set<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Loop through the set with an iterator
for (auto it = cars.begin(); it != cars.end(); ++it) {
  cout << *it << "\n";
}
```

[Try it Yourself »]

Map Example

```
// Create a map that will store strings and integers
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };

// Loop through the map with an iterator
for (auto it = people.begin(); it != people.end(); ++it) {
  cout << it->first << " is: " << it->second << "\n";
}
```

[Try it Yourself »]

Iterator Support

The examples above shows how to iterate through different data structures that support iterators (vector, list, deque, map and set support iterators, while **stacks and queues do not**).

Algorithms

Another important feature of iterators is that they are used with different algorithm functions, such as sort() and find() (found in the <algorithm> library), to sort and search for elements in a data structure.

For example, the sort() function takes iterators (typically returned by begin() and end()) as parameters to sort elements in a data structure from the beginning to the end.

In this example, the elements are sorted alphabetically since they are strings:

Example

```
#include <iostream>
#include <vector>
#include <algorithm>  // Include the <algorithm> library
using namespace std;

int main() {
  // Create a vector called cars that will store strings
  vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

  // Sort cars in alphabetical order
  sort(cars.begin(), cars.end());
```

```
  // Print cars in alphabetical order
  for (string car : cars) {
    cout << car << "\n";
  }

  return 0;
}
```

[Try it Yourself »]

And in this example, the elements are sorted numerically since they are integers:

Example

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // Create a vector called numbers that will store integers
  vector<int> numbers = {1, 7, 3, 5, 9, 2};

  // Sort numbers numerically
  sort(numbers.begin(), numbers.end());

  for (int num : numbers) {
    cout << num << "\n";
  }

  return 0;
}
```

[Try it Yourself »]

To reverse the order, you can use rbegin() and rend() instead of begin() and end():

Example

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // Create a vector called numbers that will store integers
  vector<int> numbers = {1, 7, 3, 5, 9, 2};

  // Sort numbers numerically in reverse order
  sort(numbers.rbegin(), numbers.rend());

  for (int num : numbers) {
    cout << num << "\n";
  }
```

```
  return 0;
}
```

Try it Yourself »

C++ Algorithms

In the previous chapters, you learned that data structures (like vectors, lists, etc) are used to store and organize data.

Algorithms are used to solve problems by sorting, searching, and manipulating data structures.

The <algorithm> library provides many useful functions to perform these tasks with iterators.

To use these functions, you must include the <algorithm> header file:

```
// Include the algorithm library
#include <algorithm>
```

Sorting Algorithms

To sort elements in a data structure, you can use the sort() function.

The sort() function takes iterators (typically a *start iterator* returned by begin() and an *end iterator* returned by end()) as parameters:

Example

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Sort cars alphabetically
sort(cars.begin(), cars.end());
```

Try it Yourself »

By default, the elements are sorted in ascending order. In the example above, the elements are sorted alphabetically since they are strings.

If we had a vector of integers, they would be sorted numerically:

Example

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically
sort(numbers.begin(), numbers.end());
```

Try it Yourself »

To reverse the order, you can use rbegin() and rend() instead of begin() and end():

Example

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically in reverse order
sort(numbers.rbegin(), numbers.rend());
```

Try it Yourself »

To only sort specific elements, you could write:

Example

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically, starting from the fourth element (only sort 5, 9, and 2)
sort(numbers.begin() + 3, numbers.end());
```

[Try it Yourself »]

Searching Algorithms

To search for specific elements in a vector, you can use the find() function.

It takes three parameters: *start_iterator*, *end_iterator*, *value*, where *value* is the value to search for:

Example

Seach for the number **3** in "numbers":

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Search for the number 3
auto it = find(numbers.begin(), numbers.end(), 3);
```

[Try it Yourself »]

To search for the first element that is **greater than** a specific value, you can use the upper_bound() function:

Example

Find the first value greater than **5** in "numbers":

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort the vector in ascending order
sort(numbers.begin(), numbers.end());

// Find the first value that is greater than 5 in the sorted vector
auto it = upper_bound(numbers.begin(), numbers.end(), 5);
```

[Try it Yourself »]

The upper_bound() function is typically used on sorted data structures. That's why we first sort the vector in the example above.

To find the smallest element in a vector, use the min_element() function:

Example

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Find the smallest number
auto it = min_element(numbers.begin(), numbers.end());
```

[Try it Yourself »]

To find the largest element, use the max_element() function:

Example

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Find the largest number
auto it = max_element(numbers.begin(), numbers.end());
```

[Try it Yourself »]

Modifying Algorithms

To copy elements from one vector to another, you can use the copy() function:

Example

Copy elements from one vector to another:

```
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Create a vector called copiedNumbers that should store 6 integers
vector<int> copiedNumbers(6);

// Copy elements from numbers to copiedNumbers
copy(numbers.begin(), numbers.end(), copiedNumbers.begin());
```

[Try it Yourself »]

To fill all elements in a vector with a value, you can use the fill() function:

Example

Fill all elements in the numbers vector with the value 35:

```
// Create a vector called numbers that will store 6 integers
vector<int> numbers(6);

// Fill all elements in the numbers vector with the value 35
fill(numbers.begin(), numbers.end(), 35);
```

C++ Namespaces

Namespaces

A **namespace** is a way to group related code together under a name. It helps you avoid naming conflicts when your code grows or when you use code from multiple sources.

Think of a namespace like a folder: you can have a variable named x in two different folders, and they won't clash.

Why Use Namespaces?

- To avoid name conflicts, especially in larger projects
- To organize code into logical groups
- To separate your code from code in libraries

Basic Namespace Example

Here we define a variable called x inside a namespace called MyNamespace:

```
namespace MyNamespace {
  int x = 42;
}

int main() {
  cout << MyNamespace::x;
  return 0;
}
```

Try it Yourself »

We use MyNamespace::x to access the variable inside the namespace.

The using namespace Keyword

If you don't want to write the namespace name every time you access the variable, you can use the using keyword:

```
namespace MyNamespace {
  int x = 42;
}

using namespace MyNamespace;

int main() {
  cout << x;  // No need to write MyNamespace::x
  return 0;
}
```

Try it Yourself »

However, be careful: In large programs, using using namespace can cause name conflicts. It's often better to use the full name like MyNamespace::x instead.

The std Namespace

In C++, things like cout, cin, and endl belong to the Standard Library.

These are all part of a namespace called std, which stands for **standard**. That means you normally have to write std::cout, std::cin, and so on.

To make your code shorter, you can add:

using namespace std;

This lets you use cout, cin, and endl without writing std:: every time.

Without using namespace std

```
#include <iostream>

int main() {
  std::cout << "Hello World!\n";
  return 0;
}
```

You must type std:: before cout.

With using namespace std

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!\n";
  return 0;
}
```

Now you can use cout without writing std:: every time.

Should You Always Use It?

For small programs and learning, using namespace std is fine.

But in large projects, it is better to write std:: before each item. This prevents conflicts if different libraries have functions or variables with the same name.

In short: using namespace std; is helpful for beginners, but use it with care in big programs.

C++ Reference Documentation

A list of C++ keywords and popular libraries can be found here:

Keywords <iostream> <fstream> <cmath> <string> <cstring> <ctime> <vector> <algorithm>

C++ Keywords

A list of useful keywords in C++ can be found in the table below.

Keyword	Description
and	An alternative way to write the logical && operator
and_eq	An alternative way to write the &= assignment operator
auto	Automatically detects the type of a variable based on the value you assign to it
bitand	An alternative way to write the & bitwise operator
bitor	An alternative way to write the | bitwise operator
bool	A data type that can only store true or false values
break	Breaks out of a loop or a switch block
case	Marks a block of code in switch statements
catch	Catches exceptions generated by try statements
char	A data type that can store a single character
class	Defines a class
compl	An alternative way to write the ~ bitwise operator
const	Defines a variable or parameter as a constant (unchangeable) or specifies that a class method class
continue	Continues to the next iteration of a loop
default	Specifies the default block of code in a switch statement
delete	Frees dynamic memory
do	Used together with while to create a do/while loop

double	A data type that is usually 64 bits long which can store fractional numbers
else	Used in conditional statements
enum	Declares an enumerated type
false	A boolean value equivalent to 0
float	A data type that is usually 32 bits long which can store fractional numbers
for	Creates a for loop
friend	Specifies classes and functions which have access to private and protected members
goto	Jumps to a line of code specified by a label
if	Makes a conditional statement
int	A data type that is usually 32 bits long which can store whole numbers
long	Ensures that an integer is at least 32 bits long (use *long long* to ensure 64 bits)
namespace	Declares a namespace
new	Reserves dynamic memory
not	An alternative way to write the logical ! operator
not_eq	An alternative way to write the != comparison operator
or	An alternative way to write the logical || operator
or_eq	An alternative way to write the |= assignment operator
private	An access modifier which makes a member only accessible within the declared class
protected	An access modifier which makes a member only accessible within the declared class and its c
public	An access modifier which makes a member accessible from anywhere
return	Used to return a value from a function
short	Reduces the size of an integer to 16 bits

signed	Specifies that an int or char can represent positive and negative values (this is the default so t
sizeof	An operator that returns the amount of memory occupied by a variable or data type
static	Specifies that an attribute or method belongs to the class itself instead of instances of the clas Specifies that a variable in a function keeps its value after the function ends
struct	Defines a structure
switch	Selects one of many code blocks to be executed
template	Declares a template class or template function
this	A variable that is available inside class methods and constructors which contians a pointer to a
throw	Creates a custom error which can be caught by a try...catch statement
true	A boolean value equivalent to 1
try	Creates a try...catch statement
typedef	Defines a custom data type
unsigned	Specifies that an int or char should only represent positive values which allows for storing nur
using	Allows variables and functions from a namespace to be used without the namespace's prefix
virtual	Specifies that a class method is virtual
void	Indicates a function that does not return a value or specifies a pointer to a data with an unspe
while	Creates a while loop
xor	An alternative way to write the ^ bitwise operator
xor_eq	An alternative way to write the ^= assignment operator

C++ iostream objects

The <iostream> library provides objects which can read user input and output data to the console or to a file.

A list of all iostream objects can be found in the table below.

Object	Description
cerr	An output stream for error messages
clog	An output stream to log program information
cin	An input stream that reads keyboard input from the console by default
cout	An output stream which writes output to the console by default
wcerr	The same as cerr but outputs wide char (wchar_t) data rather than char data
wclog	The same as clog but outputs wide char (wchar_t) data rather than char data
wcin	The same as cin but interprets each input character as a wide char (wchar_t)
wcout	The same as cout but outputs wide char (wchar_t) data rather than char data

C++ fstream Library (File Streams)

❮ PreviousNext ❯

C++ fstream classes

The <fstream> library provides classes for reading and writing into files or data streams.

A list of useful fstream classes can be found in the table below.

Class	Description
filebuf	A lower level file handling class used internally by the fstream, ifstream and ofstream classes
fstream	A class that can read and write to files
ifstream	A class that can read from files
ofstream	A class that can write to files

C++ Math Functions

The <cmath> library has many functions that allow you to perform mathematical tasks on numbers.

A list of all math functions can be found in the table below:

Function	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
acosh(x)	Returns the hyperbolic arccosine of x
asin(x)	Returns the arcsine of x, in radians
asinh(x)	Returns the hyperbolic arcsine of x
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y, x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordina
atanh(x)	Returns the hyperbolic arctangent of x
cbrt(x)	Returns the cube root of x
ceil(x)	Returns the value of x rounded up to its nearest integer
copysign(x, y)	Returns the first floating point x with the sign of the second floating point y
cos(x)	Returns the cosine of x (x is in radians)
cosh(x)	Returns the hyperbolic cosine of x
exp(x)	Returns the value of E^x
exp2(x)	Returns the value of 2^x
expm1(x)	Returns e^x-1
erf(x)	Returns the value of the error function at x
erfc(x)	Returns the value of the complementary error function at x

fabs(x)	Returns the absolute value of a floating x
fdim(x)	Returns the positive difference between x and y
floor(x)	Returns the value of x rounded down to its nearest integer
fma(x, y, z)	Returns x*y+z without losing precision
fmax(x, y)	Returns the highest value of a floating x and y
fmin(x, y)	Returns the lowest value of a floating x and y
fmod(x, y)	Returns the floating point remainder of x/y
frexp(x, y)	With x expressed as $m*2^n$, returns the value of m (a value between 0.5 and 1.0) and writes the pointer y
hypot(x, y)	Returns sqrt(x^2 +y^2) without intermediate overflow or underflow
ilogb(x)	Returns the integer part of the floating-point base logarithm of x
ldexp(x, y)	Returns $x*2^y$
lgamma(x)	Returns the logarithm of the absolute value of the gamma function at x
llrint(x)	Rounds x to a nearby integer and returns the result as a long long integer
llround(x)	Rounds x to the nearest integer and returns the result as a long long integer
log(x)	Returns the natural logarithm of x
log10(x)	Returns the base 10 logarithm of x
log1p(x)	Returns the natural logarithm of x+1
log2(x)	Returns the base 2 logarithm of the absolute value of x
logb(x)	Returns the floating-point base logarithm of the absolute value of x
lrint(x)	Rounds x to a nearby integer and returns the result as a long integer
lround(x)	Rounds x to the nearest integer and returns the result as a long integer
modf(x, y)	Returns the decimal part of x and writes the integer part to the memory at the pointer y

nan(s)	Returns a NaN (Not a Number) value
nearbyint(x)	Returns x rounded to a nearby integer
nextafter(x, y)	Returns the closest floating point number to x in the direction of y
nexttoward(x, y)	Returns the closest floating point number to x in the direction of y
pow(x, y)	Returns the value of x to the power of y
remainder(x, y)	Return the remainder of x/y rounded to the nearest integer
remquo(x, y, z)	Calculates x/y rounded to the nearest integer, writes the result to the memory at the pointer z
rint(x)	Returns x rounded to a nearby integer
round(x)	Returns x rounded to the nearest integer
scalbln(x, y)	Returns $x*R^y$ (R is usually 2)
scalbn(x, y)	Returns $x*R^y$ (R is usually 2)
sin(x)	Returns the sine of x (x is in radians)
sinh(x)	Returns the hyperbolic sine of x
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of x (x is in radians)
tanh(x)	Returns the hyperbolic tangent of x
tgamma(x)	Returns the value of the gamma function at x
trunc(x)	Returns the integer part of x

C++ string Functions

The <string> library has many functions that allow you to perform tasks on strings.

A list of all string functions can be found in the table below.

Function	Description
append()	Adds characters or another string to the end of the current string
at()	Returns the character at a specified index, with bounds checking
back()	Accesses the last character in the string
begin()	Returns an iterator pointing to the first character of the string
c_str()	Returns a C-style null-terminated string
clear()	Removes all characters, making the string empty
compare()	Compares the string with another string and returns the result
copy()	Copies characters from the string into a character array
data()	Returns a pointer to the string's internal character array
empty()	Checks whether the string is empty
end()	Returns an iterator pointing just past the last character
erase()	Deletes part of the string by position and length
find()	Finds the first occurrence of a character or substring
front()	Accesses the first character in the string
insert()	Inserts characters or a substring at a specified position
length()	Returns the number of characters in the string
max_size()	Returns the maximum number of characters of a string
operator[]	Returns the character at a given index
pop_back()	Removes the last character from the string

push_back()	Adds a single character to the end of the string
replace()	Replaces part of the string with new content
rfind()	Finds the last occurrence of a character or substring
resize()	Changes the size of the string, either trimming or padding it
size()	Alias of length(); returns the string's length
substr()	Returns a portion of the string, starting at a given index and length
swap()	Exchanges the contents of two strings

C++ cstring Functions

The <cstring> library has many functions that allow you to perform tasks on arrays and C-style strings.

Note that C-style strings are different than regular strings. A C-style string is an array of characters, created with the char type. To learn more about C-style strings, read our C Strings Tutorial.

A list of all **cstring** functions can be found in the table below.

Function	Description
memchr()	Returns a pointer to the first occurrence of a value in a block of memory
memcmp()	Compares two blocks of memory to determine which one represents a larger numeric value
memcpy()	Copies data from one block of memory to another
memmove()	Copies data from one block of memory to another accounting for the possibility that the bloc
memset()	Sets all of the bytes in a block of memory to the same value
strcat()	Appends one C-style string to the end of another
strchr()	Returns a pointer to the first occurrence of a character in a C-style string
strcmp()	Compares the ASCII values of characters in two C-style strings to determine which string has a
strcoll()	Compares the locale-based values of characters in two C-style strings to determine which stri
strcpy()	Copies the characters of a C-style string into the memory of another string
strcspn()	Returns the length of a C-style string up to the first occurrence of one of the specified charact
strerror()	Returns a C-style string describing the meaning of an error code
strlen()	Return the length of a C-style string
strncat()	Appends a number of characters from a C-style string to the end of another string
strncmp()	Compares the ASCII values of a specified number of characters in two C-style strings to deterr
strncpy()	Copies a number of characters from one C-style string into the memory of another string
strpbrk()	Returns a pointer to the first position in a C-style string which contains one of the specified ch
strrchr()	Returns a pointer to the last occurrence of a character in a C-style string

strspn()	Returns the length of a C-style string up to the first character which is not one of the specified
strstr()	Returns a pointer to the first occurrence of a C-style string in another string
strtok()	Splits a string into pieces using delimiters
strxfrm()	Convert characters in a C-style string from ASCII encoding to the encoding of the current local

C++ ctime Functions

The <ctime> library has a variety of functions that allow you to measure dates and times.

Function	Description
asctime()	Returns a C-style string representation of the time in a tm structure
clock()	Returns a number representing the amount of time that has passed while the program is runn
ctime()	Returns a C-style string representation of the time in a timestamp
difftime()	Returns the time difference between two timestamps
gmtime()	Converts a timestamp into a tm structure representing its time at the GMT time zone
localtime()	Converts a timestamp into a tm structure representing its time in the system's local time zone
mktime()	Converts a tm structure into a timestamp
strftime()	Writes a C-style string representing the date and time of a tm structure with a variety of form
time()	Returns a timestamp representing the current moment in time

C++ vector Library

The <vector> library has many functions that allow you to perform tasks on vectors.

A list of popular vector functions can be found in the table below.

Function	Description
assign()	Fills a vector with multiple values
at()	Returns an indexed element from a vector
back()	Returns the last element of a vector
begin()	Returns an iterator pointing to the beginning of a vector
capacity()	Returns the number of elements that a vector's reserved memory is able to store
clear()	Removes all of the contents of a vector
data()	Returns a pointer to the block of memory where a vector's elements are stored
empty()	Checks whether a vector is empty or not
end()	Returns an iterator pointing to the end of a vector
erase()	Removes a number of elements from a vector
front()	Returns the first element of a vector
insert()	Inserts a number of elements into a vector
max_size()	Returns the maximum number of elements that a vector can have
pop_back()	Removes the last element of a vector
push_back()	Adds an element to the end of a vector
rbegin()	Returns a reverse iterator pointing to the last element of a vector
rend()	Returns a reverse iterator pointing to a position right before the first element of a vector
reserve()	Reserves memory for a vector
resize()	Changes the size of a vector, adding or removing elements if necessary

shrink_to_fit()	Reduces the reseved memory of a vector if necessary to exactly fit the number of elements
size()	Returns the number of elements in a vector
swap()	Swaps the contents of one vector with another

C++ algorithm Library

The <algorithm> library has many functions that allow you to modify ranges of data from data structures.

A list of useful functions in the algorithm library can be found below.

Function	Description
adjacent_find()	Finds a pair of consecutive elements with the same value in a data range
all_of()	Checks if all of the elements in a data range match a condition
any_of()	Checks if at least one element in a data range matches a condition
binary_search()	An efficient algorithm for finding if a value exists in a sorted data range
copy()	Copies the values from a data range into a different data range
count()	Counts the number of times that a value occurs in a data range
count_if()	Counts the number of elements in a data range that match a condition
fill()	Writes a value into every element of a data range
find()	Finds the first element of a data range with a specified value
find_first_of()	Finds the first element of a data range which matches one of several specified values
find_if()	Finds the first element of a data range which matches a condition
find_if_not()	Finds the first element of a data range which does not match a condition
for_each()	Runs a function on every element in a data range
includes()	Checks if all of the values in a sorted data range exist in another sorted data range
is_permutation()	Checks if a data range is a permutation of another
is_sorted()	Checks if a data range is sorted
is_sorted_until()	Finds the position in a data range at which elements are no longer sorted

lower_bound()	Finds the first element at or above a specified lower bound in a sorted data range
max_element()	Finds the element with the highest value in a data range
merge()	Merges the values of two data ranges into a new data range
min_element()	Finds the element with the lowest value in a data range
none_of()	Checks if none of the elements in a data range match a condition
random_shuffle()	Randomly rearranges the elements in a data range
replace()	Replaces all occurrences of a value in a data range with a different value
replace_copy()	Creates a copy of a data range with all occurrences of a specified value replaced with a differe
replace_copy_if()	Creates a copy of a data rage where all values that match a condition are replaced with a diffe
replace_if()	Replaces all values in a data range that match a condition with a different value
reverse()	Reverses the order of elements in a data range
reverse_copy()	Creates a copy of a data range with the elements in reverse order
search()	Finds a specified sequence of values in a data range
sort()	Sorts the values of a data range in ascending order
swap()	Swaps the values of two variables
swap_ranges()	Swaps the values of two data ranges of the same size
upper_bound()	Finds the first element above a specified upper bound in a sorted data range