Cheatsheets / **C++ for Programmers**

codecademy

C++'s Built-In Data Structures

vectors

In C++, a vector is a data structure that stores a sequence of elements that can be accessed by index.
Unlike arrays, vectors can dynamically shrink and grow in size.
The standard `<vector>` library provide methods for vector operations:

- `.push_back()` : add element to the end of the vector.
- `.pop_back()` : remove element from the end of the vector.
- `.size()` : return the size of the vector.
- `.empty()` : return whether the vector is empty.

```
#include <iostream>
#include <vector>

int main () {
  std::vector <int> primes = {2, 3, 5, 7,
11};

  std::cout << primes[2];        //
Outputs: 5

  primes.push_back(13);
  primes.push_back(17);
  primes.pop_back();

  for (int i = 0; i < primes.size(); i++)
{
      std::cout << primes[i] << " ";
  }
  // Outputs: 2 3 5 7 11 13

  return 0;
}
```

Stacks and Queues

In C++, *stacks* and *queues* are data structures for storing data in specific orders.

Stacks are designed to operate in a **Last-In-First-Out** context (LIFO), where elements are inserted and extracted only from one end of the container.

- `.push()` add an element at the top of the stack.
- `.pop()` remove the element at the top of the stack.

Queues are designed to operate in a **First-In-First-Out** context (FIFO), where elements are inserted into one end of the container and extracted from the other.

- `.push()` add an element at the end of the queue.
- `.pop()` remove the element at the front of the queue.

```
#include <iostream>
#include <stack>
#include <queue>

int main()
{
  std::stack<int> tower;

  tower.push(3);
  tower.push(2);
  tower.push(1);

  while(!tower.empty()) {
    std::cout << tower.top() << " ";
    tower.pop();
  }
  // Outputs: 1 2 3

  std::queue<int> order;

  order.push(10);
  order.push(9);
  order.push(8);

  while(!order.empty()) {
    std::cout << order.front() << " ";
    order.pop();
  }
  // Outputs: 10 9 8

  return 0;
}
```

codecademy

Sets

In C++, a *set* is a data structure that contains a collection of unique elements. Elements of a set are index by their own values, or *keys*.

A set cannot contain duplicate elements. Once an element has been added to a set, that element cannot be modified.

The following methods apply to both `unordered_set` and `set`:

- `.insert()` : add an element to the set.
- `.erase()` : removes an element from the set.
- `.count()` : check whether an element exists in the set.
- `.size()` : return the size of the set.

```
#include <iostream>
#include <unordered_set>
#include <set>

int main()
{
  std::unordered_set<int> primes({2, 3, 5, 7});

  primes.insert(11);
  primes.insert(13);
  primes.insert(11);  // Duplicates are not inserted

  primes.erase(2);
  primes.erase(13);

  // Outputs: primes does not contain 2.
  if(primes.count(2))
    std::cout << "primes contains 2.\n";
  else
    std::cout << "primes does not contain 2.\n";

  // Outputs: Size of primes: 4
  std::cout << "Size of primes: " << primes.size() << "\n";

  return 0;
}
```

arrays

Arrays in C++ are used to store a collection of values of the same type. The size of an array is specified when it is declared and cannot change afterward.
Use `[]` and an integer index to access an array element.
Keep in mind: array indices start with `0`, not `1`!.
A multidimensional array is an “array of arrays” and is declared by adding extra sets of indices to the array name.

```
#include <iostream>

using namespace std;

int main()
{
  char vowels[5] = {'a', 'e', 'i', 'o',
'u'};

  std::cout << vowels[2];       //
Outputs: i

  char game[3][3] = {
    {'x', 'o', 'o'} ,
    {'o', 'x', 'x'} ,
    {'o', 'o', 'x'}
  };

        std::cout << game[0][2];        //
Outputs: o

  return 0;
}
```

Hash Maps

In C++, a *hash map* is a data structure that contains a collection of unique elements in the form of *key-value* pairs. Elements of a hash map are identified by key values, while the *mapped values* are the content associated with the keys.

Each element of a `map` or `unordered_map` is an object of type `pair`. A `pair` object has two member variables:

- `.first` is the value of the key
- `.second` is the mapped value

The following methods apply to both `unordered_map` and `map`:

- `.insert()` : add an element to the map.
- `.erase()` : removes an element from the map.
- `.count()` : check whether an element exists in the map.
- `.size()` : return the size of the map.
- `[]` operater:
  - If the specified key matches an element in the map, then access the mapped value associated with that key.
  - If the specified key doesn't match any element in the map, add a new element to the map with that key.

```
#include <iostream>
#include <unordered_map>
#include <map>

int main() {
  std::unordered_map<std::string, int>
country_codes;

  country_codes.insert({"Thailand", 65});
  country_codes.insert({"Peru", 51});
  country_codes["Japan"] = 81;           //
Add a new element
  country_codes["Thailand"] = 66; //
Access an element

  country_codes.erase("Peru");

  // Outputs: There isn't a code for
Belgium
  if (country_codes.count("Belgium")) {
    std::cout << "There is a code for
Belgium\n";
  }
  else {
   std::cout << "There isn't a code for
Belgium\n";
  }

  // Outputs: 81
  std::cout << country_codes["Japan"] <<
"\n";

  // Outputs: 2
  std::cout << country_codes.size() <<
"\n";

  // Outputs: Japan 81
```

```
  //          Thailand 66
  for(auto it: country_codes){
    std::cout << it.first << " " <<
it.second << "\n";
  }

  return 0;
}
```

Print Share