Cheat Sheat : C++ Syntax

Table of Contents

Hello World

You can write a simple Hello World program in C++ like this:

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

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

But you can also write the same program like in C:

```
#include <iostream>

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

Variables

This is the same as in C. But in C++ you can initialize variables like in C or like this:

```
    [type] [name]([value]);
```

- Example:

```
int a = 5; // like in C
float b(3.14); // typical C++ initialization b = 3.14
char c('a'); // typical C++ initialization c = 'a'
```

Data Types

- **Basic Data Types**: Same as in C. But in C++ you can use `bool` for boolean values. | Type | Description | Size | Range | | --- | --- | --- | --- | | bool | Boolean value | 1 byte | true = 1, false = 0 |

- **Derived Data Types**: Same as in C. But in C++ you can use `string` for strings and you have also `class`.

  - **String**:

    ```
    std::string [name] = "[value]";
    ```

    - Example:

      ```
      std::string myString = "Hello";
      ```

    or

    ```
    using namespace std;
    string [name] = "[value]";
    ```

    - Example:

      ```
      using namespace std;
      string myString = "Hello";
      ```

  - **Class**:

    ```
    class [name] {
        [members]
    };
    ```

    - Example:

      ```
      class MyClass {
          public:
              int myNum;
              string myString;
      };
      ```

Operators

All operators are the same as in C.

Control Structures

All control structures are the same as in C.

Functions

Functions are the same as in C. But in C++ you can use default arguments and function overloading.

- **Default Arguments**:

```
[return type] [name]([type] [name] = [value]) {
    [code]
}
```

  - Example:

```
int myFunction(int x = 5) {
    return x;
}
```

- **Function Overloading**:

```
[return type] [name]([type] [name]) {
    [code]
}
[return type] [name]([type] [name], [type] [name]) {
    [code]
}
```

  - Example:

```
int myFunction(int x) {
    return x;
}
int myFunction(int x, int y) {
    return x + y;
}
```

Classes

Classes are like structures in C. But you have more features like `public`, `private` and `protected` that you can use to protect your data, not like in C where everything is public.

- **Class Declaration

```
class [name] {
    public:
        [members]
    private:
        [members]
    protected:
        [members]
};
```

  - Example:

```
class MyClass {
    public:
        int myNum;
        string myString;
    private:
        int myPrivateNum;
};
```

- **Class Definition**

```
[return type] [name]::[member] {
    [code]
}
```

  - Example:

```
int MyClass::myFunction() {
    return myNum;
}
```

- **Constructor**

```
[name]([parameters]) {
    [code]
}
```

  - Example:

```
MyClass::MyClass(int x, string y) {
    myNum = x;
    myString = y;
}
```

The constructor has always the same name as the class.

- **Destructor**

```
~[name]() {
    [code]
}
```

  - Example:

```
MyClass::~MyClass() {
    std::cout << "Destructor called" << std::endl;
}
```

The destructor has always the same name as the class but with a ~ in front of it. You can also use destructors to free memory, close files, etc. This is important when you use dynamic memory allocation in your class.

- **Inheritance**

```
class [name] : [visibility] [base class] {
    [members]
};
```

  - Example:

```
class MyOtherClass : public MyClass {
    public:
        int myOtherNum;
};
```

All the members of the base class are inherited by the derived class but the derived class can also have its own members. If you want to access directly the members of the base class you need to use `protected` or `public` inheritance. `private` inheritance is also possible but you can't access the members of the base class.

- **Polymorphism** Polymorphism is the ability to have many methods with the same name but with different implementations. If you want to use polymorphism you need to use virtual functions in the base class.

  - Example:

```
class MyClass {
    public:
        virtual void myFunction();
};

class MyOtherClass : public MyClass {
    public:
        void myFunction() override;
};

void MyClass::myFunction() {
    std::cout << "MyClass" << std::endl;
}

void MyOtherClass::myFunction() {
    std::cout << "MyOtherClass" << std::endl;
}
```

In this example, the `myFunction` method is a virtual function in the base class. The `override` keyword is used to tell the compiler that the function is supposed to override a function in the base class. If the function does not exist in the base class, the compiler will give you an error.

Input/Output

Input and output are the same as in C. But in C++ you can use `cin` and `cout` for input and output.

- **Input**

```
std::cin >> [variable];
```

  - Example:

```
int x;
std::cin >> x;
```

```
```

- **Output**

```
std::cout << [value];
```

  - Example:

```
int x = 5;
std::cout << x;
```

Pointers

Pointers are the same as in C. But in C++ you can use `nullptr` instead of `NULL`. And you can also use `this` to refer to the current object, like in classes.

- **this**

```
[return type] [name]([parameters]) {
    this->[member] = [value];
}
```

  - Example:

```
void MyClass::myFunction(int x) {
    this->myNum = x;
}
```

- **nullptr**

```
[type] *[name] = nullptr;
```

  - Example:

```
int *ptr = nullptr;
```

Memory Management

Memory management is the same as in C. But in C++ you can use `new` and `delete` to allocate and deallocate memory.

- **new**

```
[type] *[name] = new [type];
```

  - Example:

```
int *ptr = new int;
```

`new` is like `malloc` in C but it also calls the constructor of the object if it exists.

- **delete**

  ```
  delete [name];
  ```

  - Example:

    ```
    delete ptr;
    ```

  `delete` is like `free` in C but it also calls the destructor of the object if it exists.

Preprocessor Directives

Preprocessor directives are the same as in C. But in C++ you can use `#include <iostream>` to include the input/output library.

Standard Template Library (STL)

The Standard Template Library (STL) is a library that provides you with a lot of useful functions and classes.

- **Vector**
  - **include**

    ```
    #include <vector>
    ```

  - **Declaration**

    ```
    std::vector<[type]> [name];
    ```

    - Example:

      ```
      std::vector<int> myVector;
      ```

  - **Initialization**

    ```
    std::vector<[type]> [name] = {[values]};
    ```

    - Example:

      ```
      std::vector<int> myVector = {1, 2, 3, 4, 5};
      ```

  - **Add Element**

    ```
    [name].push_back([value]);
    ```

    - Example:

      ```
      myVector.push_back(6);
      ```

  - **Remove Element**

```
[name].pop_back();
```

- Example:

```
myVector.pop_back();
```

- **Access Element**

```
[name][index];
```

- Example:

```
std::cout << myVector[0];
```

or you can use the `at` method:

```
[name].at([index]);
```

- Example:

```
std::cout << myVector.at(0);
```

This method is safer because it checks if the index is in the range of the vector and if you store pointers in the vector it will obligate you to use this method.

- **Size**

```
[name].size();
```

- Example:

```
std::cout << myVector.size();
```

- **Iterate**

```
for ([type] [name] : [name]) {
    [code]
}
```

- Example:

```
for (int i : myVector) {
    std::cout << i << std::endl;
}
```

or you can use the `begin` and `end` methods:

```
for (std::vector<int>::iterator it = myVector.begin(); it != myVector.end(); ++it) {
    std::cout << *it << std::endl;
}
```

or you can use the `rbegin` and `rend` methods to iterate in reverse:

```
for (std::vector<int>::reverse_iterator it = myVector.rbegin(); it != myVector.rend(); ++it) {
    std::cout << *it << std::endl;
}
```

- **Map** This is like a dictionary in Python.
    - **include**

    ```
    #include <map>
    ```

    - **Declaration**

    ```
    std::map<[key type], [value type]> [name];
    ```

        - Example:

        ```
        std::map<int, std::string> myMap;
        ```

    - **Initialization**

    ```
    std::map<[key type], [value type]> [name] = {[{key, value}]};
    ```

        - Example:

        ```
        std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}, {3, "three"}};
        ```

    - **Add Element**

    ```
    [name][key] = [value];
    ```

        - Example:

        ```
        myMap[4] = "four";
        ```

    - **Remove Element**

    ```
    [name].erase([key]);
    ```

        - Example:

        ```
        myMap.erase(4);
        ```

    - **Access Element**

    ```
    [name][key];
    ```

        - Example:

        ```
        std::cout << myMap[1];
        ```

    - **Size**

```
[name].size();
```

- Example:

```
std::cout << myMap.size();
```

- **Iterate**

```
for (std::pair<[key type], [value type]> [name] : [name]) {
    [code]
}
```

- Example:

```
for (std::pair<int, std::string> i : myMap) {
    std::cout << i.first << " => " << i.second << std::endl;
}
```

- **Set** This is like a list in Python. But it doesn't allow duplicates.
  - **include**

```
#include <set>
```

  - **Declaration**

```
std::set<[type]> [name];
```

- Example:

```
std::set<int> mySet;
```

  - **Initialization**

```
std::set<[type]> [name] = {[values]};
```

- Example:

```
std::set<int> mySet = {1, 2, 3, 4, 5};
```

  - **Add Element**

```
[name].insert([value]);
```

- Example:

```
mySet.insert(6);
```

  - **Remove Element**

```
[name].erase([value]);
```

    - Example:

      ```
      mySet.erase(6);
      ```

  - **Size**

    ```
    [name].size();
    ```

    - Example:

      ```
      std::cout << mySet.size();
      ```

  - **Iterate**

    ```
    for ([type] [name] : [name]) {
        [code]
    }
    ```

    - Example:

      ```
      for (int i : mySet) {
          std::cout << i << std::endl;
      }
      ```

- **Queue** This is a queue data structure. It is like a list but you can only add elements at the end and remove elements from the beginning. So it is a FIFO (First In First Out) data structure.
  - **include**

    ```
    #include <queue>
    ```

  - **Declaration**

    ```
    std::queue<[type]> [name];
    ```

    - Example:

      ```
      std::queue<int> myQueue;
      ```

  - **Initialization**

    ```
    std::queue<[type]> [name];
    ```

    - Example:

      ```
      std::queue<int> myQueue;
      ```

  - **Add Element**

    ```
    [name].push([value]);
    ```

    - Example:

```
myQueue.push(1);
```

- **Remove Element**

```
[name].pop();
```

  - Example:

```
myQueue.pop();
```

- **Access Element**

```
[name].front();
```

  - Example:

```
std::cout << myQueue.front();
```

- **Size**

```
[name].size();
```

  - Example:

```
std::cout << myQueue.size();
```

- **Stack** This is a stack data structure. It is like a list but you can only add elements at the end and remove elements from the end. So it is a LIFO (Last In First Out) data structure.
  - **include**

```
#include <stack>
```

  - **Declaration**

```
std::stack<[type]> [name];
```

    - Example:

```
std::stack<int> myStack;
```

  - **Initialization**

```
std::stack<[type]> [name];
```

    - Example:

```
std::stack<int> myStack;
```

  - **Add Element**

```
[name].push([value]);
```

    - Example:

    ```
    myStack.push(1);
    ```

  - **Remove Element**

  ```
  [name].pop();
  ```

    - Example:

    ```
    myStack.pop();
    ```

  - **Access Element**

  ```
  [name].top();
  ```

    - Example:

    ```
    std::cout << myStack.top();
    ```

  - **Size**

  ```
  [name].size();
  ```

    - Example:

    ```
    std::cout << myStack.size();
    ```

- **Algorithm** This is a library that provides you with a lot of useful functions.
  - **include**

  ```
  #include <algorithm>
  ```

  - **Sort**

  ```
  std::sort([begin], [end]);
  ```

    - Example:

    ```
    std::vector<int> myVector = {5, 2, 3, 1, 4};
    std::sort(myVector.begin(), myVector.end());
    ```

  - **Reverse**

  ```
  std::reverse([begin], [end]);
  ```

    - Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::reverse(myVector.begin(), myVector.end());
```

- **Find**

```
std::find([begin], [end], [value]);
```

  - Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::vector<int>::iterator it = std::find(myVector.begin(), myVector.end(), 3);
if (it != myVector.end()) {
    std::cout << "Element found in myVector: " << *it << std::endl;
} else {
    std::cout << "Element not found in myVector" << std::endl;
}
```

- **Binary Search**

```
std::binary_search([begin], [end], [value]);
```

  - Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
if (std::binary_search(myVector.begin(), myVector.end(), 3)) {
    std::cout << "Element found in myVector" << std::endl;
} else {
    std::cout << "Element not found in myVector" << std::endl;
}
```

- **Copy**

```
std::copy([begin], [end], [destination]);
```

  - Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::vector<int> myVectorCopy(5);
std::copy(myVector.begin(), myVector.end(), myVectorCopy.begin());
```

- **Max Element**

```
std::max_element([begin], [end]);
```

  - Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::vector<int>::iterator it = std::max_element(myVector.begin(), myVector.end());
std::cout << "Max element in myVector: " << *it << std::endl;
```

- **Min Element**

```
std::min_element([begin], [end]);
```

- Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::vector<int>::iterator it = std::min_element(myVector.begin(), myVector.end());
std::cout << "Min element in myVector: " << *it << std::endl;
```

- **Count**

```
std::count([begin], [end], [value]);
```

- Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
int count = std::count(myVector.begin(), myVector.end(), 3);
std::cout << "Number of 3 in myVector: " << count << std::endl;
```

- **Sum**

```
std::accumulate([begin], [end], [initial value]);
```

- Example:

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
int sum = std::accumulate(myVector.begin(), myVector.end(), 0);
std::cout << "Sum of myVector: " << sum << std::endl;
```

Miscellaneous

- **Namespaces** Namespaces are used to avoid name conflicts. You can use the `using` keyword to avoid writing the namespace every time.
  - **Declaration**

```
namespace [name] {
    [code]
}
```

- Example:

```
namespace myNamespace {
    int myNum = 5;
}
```

- **Access**

```
[namespace]::[member];
```

- Example:

```
std::cout << myNamespace::myNum;
```

    - **Using**

      ```
      using namespace [namespace];
      ```

        - Example:

          ```
          using namespace myNamespace;
          std::cout << myNum;
          ```

- **Enums** Like in C, enums are used to define a set of named integer constants. But in C++ you can use `enum class` to avoid name conflicts.
    - **Declaration**

      ```
      enum class [name] {
          [members]
      };
      ```

        - Example:

          ```
          enum class Color {
              RED,
              GREEN,
              BLUE
          };
          ```

    - **Access**

      ```
      [name] [variable] = [name]::[member];
      ```

        - Example:

          ```
          Color myColor = Color::RED;
          ```