C++ Cheat Sheet

Overview

This document aims to act as a quick reference guide for C++ syntax.

Compilation

--

Simple Values/Literals

Numbers

- Integer literals: **0, 10, 12345, -4, -200**
- Double precision floating point literals: **0.0, 0.1, 100.24, 3.14159**
- Normal precision floating point literals: **0.0f, 0.1f, 100.24f, 3.14159f**
- Hexadecimal integers: **0x0, 0x10, 0xFF, 0xC0FFEE**

Strings

In C++ string literals are actually **const char** pointers.

“Hello this is a string”,
“First line\nSecond line”

There is a helper class in the standard library **std::string**. This is in the include **<string>**. For example:

std::string myString = “Some text string”;

Expressions

int someResult = 24 + 42; // assign 66 to someResult

double someDecimalNumber = 13.0 * 12.0 + 4.0; // assigns 160.0 to someDecimalNumber

Variables

Declaring a variable in C++ involves specifying a type and variable name, so to define an integer called j**ohnsAge:**

int johnsAge;

To initialise the value you use usual equals-sign assignment:

johnsAge = 28;

You can combine these steps and initialise a variable as soon as it's declared:

int bobsAge = 32;

Variables can be used in expressions simply by using their name:

int ageGap = bobsAge – johnsAge; // calculate age difference

Containers

The only “built in” container in C++ is C-style arrays, however the standard library contains many useful container classes such as **std::vector** (a dynamically sized array), **std::list** (a linked list) and a **std::map** (a dictionary of keys mapped to values) amongst others. Here are a few examples:

1. *std::vector* and *std::list* have similar interfaces

   Vectors are useful for storing a collection of values, you can declare them like so:

   ```
   std::vector<int> myVectorOfIntegers; // container of integers
   ```

   ```
   std::list<double> myListOfDoubles; // container of doubles
   ```

   To add a value to the back of a vector or list:

   ```
   myVectorOfIntegers.push_back(24);
   ```

   ```
   myListOfDoubles.push_back(48.3);
   ```

   The size of these containers can be fetched with the **size** method:

   ```
   int size = myVectorOfIntegers.size();
   ```

   Getting a value out can be done in the same style as a C-style array:

   ```
   int thirdValue = myVectorOfIntegers[2]; // extract 3rd value from vector
   ```

2. *std::map*:

   This type represents a key-value map, where the key is commonly a string and the value is of any type. This sort of collection is often explained using a phonebook example:

   ```
   std::map<std::string, std::string> phonebook; // declare the map of strings to strings
   ```

   ```
   phonebook[“John”] = “01234 567890”; // store a phone number under the key “John”
   ```

   ```
   phonebook[“Harry”] = “01234 890567”; // store a phone number under the key “Harry”
   ```

   To retrieve a value from a map you can pass the key in using the *sub-script* notation:

   ```
   std::string number = phonebook[“John”]; // get John's phone number out of the book
   ```

   To see if a key exists in a map you can do:

   ```
   auto found = phonebook.find(“John”);
   ```

   ```
   if(found == phonebook.end()) std::cout << “Didn't find John's number” << std::endl;
   ```

   ```
   else std::cout << “Found John's number: “ << found->second << std::endl;
   ```

Control Flow

Conditional Statements

In C++ conditional logic can be implemented with if-statements as follows:

```
if(true)
{
        std::cout << “True is True! Who'd have thought?” << std::endl;
}
else
{
        std::cout << “True isn't True? How can this be!?” << std::endl;
}
```

Which would print:

True is True! Who'd have thought?

This guide won't explain how all of the types can or can not be evaluated to t**rue** or f**alse**, but here is a quick list of *boolean operators* that produce t**rue/false** outputs depending on their arguments:

- For testing equality:

  ```
  1 == 1
  true
  1 == 2
  false
  ```

- Less than:

  ```
  1 < 2
  true
  1 < 1
  false
  ```

- Greater than:

  ```
  1 > 4
  false
  4 > 1
  frue
  ```

- Checking for difference:

  ```
  1 != 1
  false
  2 != 3
  true
  ```

For Loops

To make a piece of code repeat 5 times, such as

```
for(int x = 0; x < 5; x++)
{
	std::cout << "Hello!" << std::endl;
}
```

Would output:

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

As you can see, a for loop declares variables (**int x = 0**), gives a condition that explains what must be true for the loop to continue (**x < 5**) and dictates what should happen after each iteration (**x++**).

While Loops

In C++ a *while-loop* is similar again to most other languages, you can use one to print numbers 0 to 5 as follows:

```
int counter = 0;
while(counter < 6)
{
	std::cout << counter << std::endl;
	counter = counter + 1;
}
```

Which would output:

```
0
1
2
3
4
5
```

Functions

To define a function, such as squaring a given number:

```
double square(double x)
{
	return (x * x);
}
```

Then to call this function:

```
double result = square(10); // assigns 100 to result
```

A function does not have to return a value:

```
void say_hello_to(std::string person)
{
	std::cout << “Hello, ” << person << std::endl;
}
```

```
say_hello_to(“John”)
```

Would output:

Hello, John

A function also does not need parameters:

```
int fetch_answer()
{
	return 42;
}
```

```
fetch_answer(); // returns 42
```

Classes / Objects

Using the 3D co-ordinate example to show off C++ classes:

```
class Coordinate
{
public:
	Coordinate(double x, double y, double z)
		: m_X(x), m_Y(y), m_Z(z)
	{
	}
private:
	double m_X, m_Y, m_Z;
}
```

This defines a class called *Coordinate*. We define a *method* of this class called *Coordinate*. This is a special function, in C++ when a method has the same name as its parent class it is considered a *constructor*. In this case we are declaring a *constructor* for the *Coordinate* class that takes three parameters: **x**, **y**, and **z**. Then follows an initialiser list (between the **:** and **{** symbols) which can only be used in a constructor. An initialiser list allows you to initialise *member* variables at the time of construction, so in this case the **m_X, m_Y,** and **m_Z** *member* variables are initialised to have the values of **x, y** and **z** respectively.

To create an instance of our *Coordinate* class:

```
Coordinate origin(0, 0, 0);
Coordinate my_point(100, 15, 82);
```

The *members* in this class have been declared *private* which means they cannot be accessed from outside the class. If they were defined as *public* (like the *constructor* was) then it would be possible to access them as:

int x = my_point.m_X; // this would give a compilation error if m_X is *private* or *protected*

However, as they *private* you have to define *accessor methods* (also known as *getter methods*). These are *public* functions in the class that return the value of a *member variable* so that you can read the value but not have access to it in order to change it.

We could therefore change the **Coordinate** class to look more like:

```
class Coordinate
{
public:
        Coordinate(double x, double y, double z)
                : m_X(x), m_Y(y), m_Z(z)
        {
        }
        inline double x() const { return m_X; }
        inline double y() const { return m_Y; }
        inline double z() const { return m_Z; }
private:
        double m_X, m_Y, m_Z;
}
```

This adds three *accessor methods* **x, y** and **z.** You can call these much like ordinary functions:

```
double origin_x = origin.x();
```

The *inline* keyword suggests (note: not forces) the compiler should put the body of the function "in-situ" at the calling site, rather than generating a call instruction. This is convenient for simply accessor functions because usually they are just a memory access anyway, so the overhead of calling a function would be excessive.

The *const* keyword between the parameter list and code body tells the compiler that this method is not allowed to modify the internal state of the object it is called on. That is to say, a method denoted as *const* cannot change the value of its *member variables*.