CPP Quicknotes

Karl Statz

January 10, 2024

Contents

1 Compiler/Environment Setup **1**

2 Basics **2**

3 Control Flow **3**

4 Container Types **4**
4.1 std::vector<T> 4
4.2 std::set<T> 5
4.3 std::map<T> 6
4.4 std::iterator 6

5 Functions **6**
5.1 Value/Reference Semantics 6

1 Compiler/Environment Setup

MSVC is Microsoft's C and C++ compiler. Figuring out which standard your version of MSVC supports is an exercise in madness leaving one with a strong desire to live in a shack in the Wyoming wilderness. Using the following Version Documentation we can read the tea leaves that the following are true

- If you are using a version post September 2020 It should support the C11 and c17 standards
- If you have the same for c++ you support the latest/greatest c++

Visual studio is Microsoft's Flagship IDE for a suite of language (C#, Visual Basic, C++, F# etc). Download it from this link and follow this guide to configure it for C/C++ development.

C++ is a bit bare bones as far as intellisense (Microsoft's proprietary code completion engine) so I highly recommend Resharper C++ as a visual studio plugin that should be free with a student email address.

If you are feeling anti-microsoft the Resharper folks (Jetbrains) have a full c++ editor named Clion that is free to students and is immensely popular across the industry.

The third option, and perhaps my favorite, is Visual Studio Code. This is microsoft's free, open source (ish) text editor built on Electron with a rich plugin ecosystem it is a very viable coding environment for basically every language.

There are other options of course, depending on your preferences and how stubborn you are. If you are on a Mac CLion and Visual Studio Code are the two best options. If you are on linux, like me, you are beyond saving and likely have strong preferences for editor, compiler tool chain and pretty much everything else.

2 Basics

What would a C++ cheat sheet/quick guide be without a Hello World example:

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello World" << std::endl;
}
```

This is a fully complete c++ program that, in theory should print "Hello World". Lets go from top to bottom to figure out what all this means

```
#include <iostream>
```

the #include statement is a pre-processor directive to include a header file in your program. In this case we are including the iostream header.This link is an exhaustive list of headers included with the c++ Standard Library. In this case iostream contains the cout object. The call semantics for c++ are a bit unique and a stark contrast to C.

```
//std is the namespace std (standard)
//:: is the scope operator
//cout is the name of the object getting called
//<< is the pipe operator
//"Hello World" is the data sent over the pipe to cout
//std::endl this is a portable endline call that both appends \n, \r depending on plat
//small note, std::endl also flushes the stream
std::cout << "Hello World" << std::endl;
```

This is a very different beast than c's printf method.

```
#include <cstdio>
int main()
{
    printf("%s", "Hello World");
}
```

Notice we, in c++, include cstdio instead of stdio.h. This delves into a problem that a lot of young c++ programmers run into which is how to not stomp all over the global namespace. the c++ style includes for c headers use namespaces instead of just inserting into the global namespace

```
#include <iostream>
using std;

int  main()
{
   cout << "hello world" << endl;
}
```

This might look cleaner, but using std will import all of std into the global namespace which can cause collisions if you have the same name for things in your code (custom string implementations come to mind)

3 Control Flow

In c++ we have all the normal control flow types that exist in c

```
if(predicate) {
    //do something
}
else if(otherPredicate) {
    //do something else
}
else {
    // Do something else
}
```

Looping is also, roughly the same in c++

```
do {
} while(predicate);

while(predicate) {

}

for(int i = 0; i < LIMIT; i++) {
}
```

If you feel like the only solution to your control flow problem is a GOTO statement, you are wrong and haven't thought about it properly.

4 Container Types

One of the massive benefits to c++ is it's immense standard library. C is a spartan language with few, if any, creature comforts. In c you get arrays, which are a thin abstraction over a contiguous block of memory. With a few tricks c++ is able to have much richer system of containers for data

4.1 std::vector<T>

this container type is resizable container. If you are familiar with c# it is closest to the 'List<T>' type. Below is the basic usage. Note the universal initializer syntax (c++11 and beyond). Notice the type between angle brackets. You must type a vector. In this case it is a vector of integers.

```
#include <vector>

std::vector<int> my_ints = { 1, 2, 3, 4, 5, 6 };

my_ints.pusn_back(7); //appending to a vector is a amortized constant time (O(1)) time
```

Vectors can generally be treated as magic arrays but with some glaring caveats. The first is they cannot read your mind on how much data you are going to store in them. If you dont specify the initial capacity of a vector is usually 0 (the standard doesnt specify, but i tried a few implementations and they all defaulted to 0). This means that whenever you add data to the vector it is forcing a resize. This, in small vector's is bad but not catastrophic, but when your vector grows to the hundreds or thousands of elements every resize is a performance hit. The trick is to reserve space in the vector this will pre-size the container avoiding any resizing penalties (remember if it is allocated to the heap this means it will do more heap allocations behind the scenes)

```
std::vector<int> v;
v.reserve(100); //this is highly dependant on your use case for vectors, 100 is a plac
```

iterating through a vector is slightly different than an array. The tricky bit is there are two ways to do it

```
for(std::vector<int>::iterator it = v.begin(); i != v.end(); it++)
{
    int i = *it; //iterators are an abstraction over a pointer
}
```

Above is the most verbose way and is required if you need the underlying iterator, if you are just looping through a vector there is a more succinct syntax.

```
for(int i : v)
{
    //i here is the dereferenced value pointed at by the iterator
}
```

to iterate through a vector is slightly different than a c array.

4.2 std::set<T>

sets are a special data structure the general usecase for sets is to hold unique elements. There are two flavors of set in the c++ standard library, ordered (undecorated std::set) and std::unordered$_{\text{set}}$<T>. the base set type has the added caveat that the members of the set are in order. This adds insertion time since when you insert into the set there is a first pass to make sure the item is unique and then there is a sorting pass to make sure the set is ordered. All things being equal inserting into a unordered$_{\text{set}}$ is going to be faster than inserting into a set. For the folks that think in big O notation unordered set are is Ø(log n) and inserting into an unordered set is Ø(1). There is a quick rule for Big O notation that if you see that it is logarithmic complexity there is a binary tree somewhere in there.

4.3 std::map<T>

The map container in C++ follows the same semantics as the set container(s). The Map container is a generic hashmap implementation and is useful if you have unique keys tied to typed values. There exists the same caveat about ordered and unordered maps to keep in mind when using them.

4.4 std::iterator

The key thing to understand about c++ container types is the concept of an iterator. Iterators are an abstraction over the general pointer that you would use to iterate through an array. You can increment them (++i) and dereference them (*i).

5 Functions

Functions in c++ are syntactically similar to c but you can overload them.

```
int foo(int x);
int foo(int x, int y); //this will fail to compile wiht a name collision
```

in contrast, take the following c++ snippet

```
int foo(int x);
int foo(int x, int y);//compiles fine
```

5.1 Value/Reference Semantics

There are 3 ways to pass arguments to a function (ignoring RValue references and std::move for the time being)

- By value
- By pointer

- By reference

passing by value:

```
void foo(int x)
{
    //x is copied onto the stack memory for function foo,
    //and changes to x will not persist outside of that function call
}

void foo(int * xPtr)
{
   //here we pass the  address of x to the function
   int x = (*xPtr); //we have to dereference xPtr to get the value
   x += 10; //this will persist outside of the function call
   //since we are adding 10 to the value pointed at by xPtr
}
```

there is a common pitfall working with pointers

```
void foo(int * xPtr)
{
    xPtr += 10;//this just increments the address by 10 using pointer arithmetic;
    //given that c/c++ treat bounds checking as "quaint" this will cause untold chaos
}
```

both of these examples are identical to their C counterparts. A very common pattern in c is to pass an argument as a constant pointer. This allows the function to mutate the data passed in but it cannot re-assing the pointer

```
void foo(const * int xPtr)
{
    //this makes the bug in the previous example a compile time error
    (*xPtr) += 100; //this is, however, legal
}
```

C++ introduces the “pass by reference” semantic

```
void foo(int &x)
{
    //this is identical to const * int
    x += 10; //the main difference is you dont have to dereference references
}
```

If you want to pass by reference but also disallow the function to change the data passed in you can use the const modifier

```
void foo(const int &x)
{
    //x cannot be modified or reassigned
}
```

Below is the c version of the same semantics, the reference implementation is far superior

```
void foo(const * const int x)
{
   //x is a constant pointer to a constant int
}
```