C++ cheat sheet

Daniel Huantes

February 8, 2021

1 Introduction

2 Basics

2.1 Starting

- All C++ files need to be saved in a file ending in .cpp
- All 'header' files, or files containing information including class definitions and function definitions should be stored in .hpp or .h files.
- Header files should include ONE class definition per, with the class sharing the name of the file (minus the .hpp)
- C++ files need a method named 'main' in order to execute. The compiler looks for a method with this name and begins executing from its first line, NOT the first line of the file like in python

2.2 Including

```
//equivalent to importing in python
\#include </*libraryName*/>
\#include </*path/to/file.hpp*/>
\#include "/*local header file*/"

\#include <set>
\#include <map>
\#include <cmath>
\#include <chrono>
\#include <vector>
\#include <iostream>
\#include <libField/Field.hpp>
\#include <libInterpolate/Interpolate.hpp>
\#include "geometry.h"
```

2.3 Common Type Declarations

```
// decimal numbers
double a  = 321.044;
float b = 1.21;
//whole numbers
int c     = 45;
long d = 323e7;
// true/false
bool d = true;
bool d = 3 < 4;
// strings
std::string words = "blah␣blah␣blah";
std::string more_words = "foo␣bar␣BAZ!"
```

2.4 Common Functions

```
// printing
std::cout << "Don't␣forget␣the␣newline!" << "\n";
std::cout << "This␣is␣fine␣as␣well!\n";
std::cout << "Now␣" << "you're␣" << "being␣" << "ridiculous\n";
std::cout << "numbers:␣{" << 1 << ",␣" << 0.243 << "}\n";
// to string
float b = 1.21;
std::string interp = std::to_string(b) + "␣look␣at␣that!\n";
// concatenation
std::string stringAddition = "Run␣" + "strings" + "␣together";
// mathy stuff
// these are non-native and are included in cmath
// pow takes two arguments, x and y and returns x^y
double IV = pow(25, 2);
// the log function is natural log, or ln. If you want log base
// 10 reevaluate your life
double natural = log(0.5);
// the square root or sqrt function will always give the
// positive root
double irrational = sqrt(2);
```

2.5 Common Structure syntax

```
for(/*declaration*/; /*test*/; /*increment/increase*/){
   // do something 10 times
}
for(int i = 0; i < 10; i++){
   // do something N times
}
if(/*conditional*/){
  //do if conditional is true
}
a = 1;
```

```
if(a < 2){
  //do this if conditional is true
} else {
  //do this if conditional ISN'T true
}
while(/*conditional*/){
  // repeat this until conditional is false
}
int n = 0;
while(n < 10){
  // do this over and over
  n++;
}
```

2.6 Compiling (running)

in python compiling/running are interchangable, not in C++ C++ 'compiles' source code (human language) and sends it to 'machine code' (binary) that is stored in an unreadable, but executable file

run the following in the command line to compile (replace FILENAME with your file's name)

```
\$ g++ FILENAME.cpp -o FILENAME
```

this will generate a file called FILENAME that is executable and can be run using

```
\$ ./FILENAME
```

3 Slightly-less-basics

3.1 Writing to file

```
ofstream output;
output.open("./path/to/file.txt");
output << /*Some string or writable thing*/;
output.close();
```

3.2 Plotting

```
ofstream output;
output.open("./path/to/file.txt");
//gnuplot style is 'x y' columns
output << /*Something gnuplot style*/;
output.close();
// -e is the commandline tack, and -p is the
// 'persist' command so it doesn't get destroyed
system("gnuplot -p -e \'plot \"gnuplotFile.txt\" with lines \'");

//example of y = x^2
double dx = 0.2;
```

```
std::string buffer = "";
std::string x, y;
for(int i = 0; i < 100; i++){
  x = std::to_string(i * dx);
  y = std::to_string(pow(i * dx, 2));
  buffer +=  x + "\t" + y + "\n";
}
ofstream output;
output.open("gnuplotFile.txt");
//gnuplot style is 'x y' columns
output << buffer;
output.close();
// -e is the commandline tack, and -p is the
// 'persist' command so it doesn't get destroyed
system("gnuplot -p -e \'plot \"gnuplotFile.txt\" with lines \'");
```

3.3 Function Declaration syntax

```
/*return type*/ /*unique name*/(/*parameters*/){
  // body of function declaration, any math manipulations or
  algorithms go here. Any function with a return type not void
  must have a return statement
}
double sum_squared(double x, double y){
  return (x + y) * (x + y);
}
```

3.4 Class Declaration syntax

```
//best practice is to have one class definition per header file
class /*Name*/{
  public:
    // constructors, public functions, etc
  protected:
    // usable by subclasses
  private:
    // not directly accessible outside of class definition
};
// example class for an app to manage Jimmy John's orders
class Sandwich{
  public:
    // constructors are special 'creation' functions so they
    // share the class name and do not have a return type
    Sandwich(){

    }
    Sandwich(int menuNumber){
      if(menuNumber > 17){
```

```
            menuID = -1;
          } else {
            menuID = menuNumber;
          }
        }
        string printDescription(){
          return descriptions[menuID - 1];
        }
      protected:
        bool mayo = true;
        bool lettuce = true;
        bool tomato = true;
        bool peppers = false;
      private:
        int menuID;
        string descriptions[] = {/*too lazy to actually type this*/};
    };
```

4 Even-more-less basic

4.1 Container Declarations

```
    /*type*/ /*name*/[/*number*/];
    /*type*/ /*name*/[/*number*/] = {/*same \# of elements*/};
    /*type*/ /*name*/[] = {/*any \# of elements*/};
    double x\_values[n];
    double t\_values[100];
    double ages[3] = {34, 23, 17};
    string names[] = {"Emily", "Dan", "Patrick"};

    std::vector</*typename*/> /*name*/;
    std::vector<int> fibbonacci;
    std::vector<int> threes{3, 3, 3};
    std::vector<n, 1> zeros;
    std::vector<n, 0> ones;

    std::map</*give this type*/, /*get that type*/> /*name*/;
    std::map<int, string> MenuDescriptions = {{1, "THE␣PEPE:␣Ham␣and
␣␣␣␣␣␣Provolone..."},
    {2 "Big␣John:␣Roast␣beef,␣lettuce,␣..."},
    {3, "Totally␣Tuna:␣It's␣tuna␣baby"}};
```

4.2 Accessing Container Members

```
    /*array name*/[/*index*/]
    /*array name*/[/*index*/] = /*new value*/;
```

```
std::cout << user\_prompts[4];
x\_values[4] += 3.32;

/*vector name*/.push\_back(/*some new member, added to end*/);
// fake constructor for 'Student' class,  objects added to
// containers don't have to be named
students.push\_back(Student("Matthew", 8));

/*Map Name*/[/*new key*/] = /*new value*/;
MenuDescriptions[13] = "JIMMY␣CUBANO:␣Bacon,␣smoked␣ham...";
```

4.3 Template declarations

```
//add at top of functions to refer to inputs, etc
template <typename /*stand in name*/>;
template <typename /*...*/, typename /*...*/>;
template<typename T, typename S>
T minimum(T v1, S v2){
  if(v1 - v2 < 0){
    return v2;
  } else {
    return v1
  }
}

//add at top of class defs to require a type on declaration
template<class animalType>
class Pasture{
  public:
    Pasture(){

    }
    Pasture(std::vector<animalType> new_herd){
      for(int i = 0; i < new_herd.size(); i++){
        herd.push_back(new_herd[i]);
      }
    }
    Pasture(animalType animal){
      herd.push_back(animal);
    }
  protected:
    int feedingtime = 500;
  private:
    std::vector<animalType> herd;
}
```

4.4 Reading from a file

```
/*I barely know how to do this, updates welcome*/
std::ifstream t;
int length;
t.open("path/to/file.txt");
length = t.tellg()
t.seekg(0, std::ios::end);
buffer = new char[length];
t.read(buffer, length)
t.close();
```

4.5 Reading from a file

4.6 Special functions

```
//a lambda function is basically an unnamed function

// you can 'override' functions based on the parameters you define
// different functions with same name using different parameters, and
// the compiler decides what to use based on arguments
// often less efficient than templating or "optional" args
bool even(int n){
  return (n \% 2 == 0);
}
bool even(double a){
  return (((int) n) \% 2 == 0);
}
bool even(int n, int divisor){
  return (n \% divisor == 0);
}
```

5 Clark Galore

5.1 Setting up a Field

```
Field</*some type*/, /*int for dimensions*/>/*Name*/(/*dimensions*/);
Field<double, 1> T_vs_t(100);
```

5.2 Setting up a Coordinate System

```
Field</*some type*/, /*# of axes*/>/*Name*/(/*dimensions*/);
/*Field Name*/.setCoordinateSystem(/*RangeDiscretizer*/);
// we set the coordinate system using a custom type called a
// RangeDiscretizer which handles the problem of going from
// i -> x and N

Field<double, 1> T_vs_t(100);
T_vs_t.setCoordinateSystem(Uniform<double>(0, 5));
```

5.3 Range Discretizers

```
Field</*some type*/, /*int for dimensions*/>/*Name*/(/*dimensions*/);

Field<double, 1> T_vs_t(100);
```

6 Debugging

6.1 Common fixes

- Have you tried adding a semicolon?
- Did you (re)compile?
- Are you compiling a .cpp file with a main method?
- Did you declare everything you use?
- Are you sure the computer is interpreting your math correctly?
- PEMDAS?
- Print out intermediate variables to make sure they are correct
- Are you including ALL of the libraries you need?
- Google it! No one really ever learns to code, they just get better at using stack overflow

6.2 Less common fixes

- Are you sure you are respecting namespace? Are you correctly
- referring to the variable you intend to? (Did you blow out a value?)
- Is it passing by reference or value?
- Are you including a library one of your included libraries already includes?
- Are race conditions occuring?

6.3 Uncommon fixes

- Null pointer?
- Are you trying to access unallocated memory?
- Is there a namespace conflict?

6.4 Last-Ditch-Effort fixes

- Would garbage collection have occurred?
- Double imprecision?
- How's your prayer life?