Compiling:

```
$ g++ hello.cpp support.hpp support.cpp -o hello
```

Produces hello.exe. Use the -o modifier to rename the linked/compiled version

Conditionals:

```
if (grade != 60){
//stuff
}
else if (grade < 90){
//more stuff
}
else {
//other stuff
}

switch(number) {
    case 1 :
      std::cout << "Bulbusaur\n";
      break;
    case 2 :
      std::cout << "Ivysaur\n";
      break;
    default :
      std::cout << "Unknown\n";
      break;
}
```

Loops:

```
while (pin != 1234 && tries <= 3) {
    std::cout << "Enter your PIN: ";
    std::cin >> pin;
    tries++;
}
  for (int i = 99; i > 0; i--) {
```

```
    std::cout << i << " bottles of pop on the wall.\n";
    std::cout << "Take one down and pass it around.\n";
    std::cout << i - 1 << " bottles of pop on the wall.\n\n";
  }
```

Vectors:

Name the data type, size, or have them dynamically sized. Can be multidimensional

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

int main() {
  std::vector<int> set = {1, 2, 3, 4, 5};
  int sum = 0; //even
  int product = 1; //odd

  for (int i=0; i < set.size(); i++) {
    if (set[i]%2 != 0) { // if odd
      product = product * set[i];
    }
    else {
      sum = sum + set[i];
    }
  }
  std::cout << "Sum of even numbers is " << sum << "\n";
  std::cout << "Product of odd numbers is " << product << "\n";

}
```

Functions:

Biggest thing I'm not used to is that you have to define the parameter data type when declaring the function.

```
// Define name_x_times() below:
void name_x_times(std::string name, int x){
  while(x>0){
    std::cout << name << "\n";
    x--;
  }
```

```
}

int main() {
  std::string my_name = "Blake!";
  int some_number = 5;
  name_x_times(my_name, some_number);
}
```

```
// Define is_palindrome() here:
bool is_palindrome(std::string text){
  std::string revText = "";
  for(int x = text.length()-1; x >= 0; x--){
    revText.push_back(text[x]);
  }

  if (text.compare(revText) != 0 || text!=revText){
    return false;
  } else if(text.compare(revText) == 0 || text==revText){
      return true;
  }
}

int main() {
  std::cout << is_palindrome("madam") << "\n";
}
```

Scope:

Region of code that can access or view a given element.

- Variables defined in global scope are accessible throughout the program.
- Variables defined in a function have local scope and are only accessible inside the function.

You can declare functions above main() and then define them below main if writing everything in a single file. A cleaner option is to have a main.cpp and then functions.cpp. Still have to declare functions above main() in main.cpp, but can add the definitions in functions.cpp.

Main.cpp:

```
#include <iostream>
#include <cmath>
// Add declarations here:
```

```
double average(double num1, double num2);
int tenth_power(int num);
bool is_palindrome(std::string text);

int main() {
  std::cout << is_palindrome("racecar") << "\n";
  std::cout << tenth_power(3) << "\n";
  std::cout << average(8.0, 19.0) << "\n";
}
```

Functions.cpp:

```
#include <iostream>
#include <cmath>

// Add definitions here:
double average(double num1, double num2) {
  return (num1 + num2) / 2;
}
int tenth_power(int num) {
  return pow(num, 10);
}

bool is_palindrome(std::string text) {
  std::string reversed_text = "";

  for (int i = text.size() - 1; i >= 0; i--) {
    reversed_text += text[i];
  }
  if (reversed_text == text) {
    return true;
  }
  return false;
}
```

Header files

When you wind up with a fuckton of functions, a header file can save the day so that you don't have to scroll through 1 million declarations above your main(). Convention is that the header file has the same name (with .hpp file type) as the .cpp file with function definitions.

Main.cpp:

```
#include <iostream>
```

```cpp
#include "fns.hpp"

int main() {
  std::cout << is_palindrome("noon") << "\n";
  std::cout << tenth_power(4) << "\n";
  std::cout << average(4.0, 7.0) << "\n";
}
```

Functions.hpp:

```cpp
// Move function declarations here:
double average(double num1, double num2);
int tenth_power(int num);
bool is_palindrome(std::string text);
```

Functions.cpp

```cpp
#include <iostream>
#include <cmath>

double average(double num1, double num2) {
  return (num1 + num2) / 2;
}
int tenth_power(int num) {
  return pow(num, 10);
}

bool is_palindrome(std::string text) {
  std::string reversed_text = "";
  for (int i = text.size() - 1; i >= 0; i--) {
    reversed_text += text[i];
  }
  if (reversed_text == text) {
    return true;
  }
  return false;
}
```

Inline functions

Extremely short bodied functions that are added to the header file. They require that you use the “inline” keyword in the header.

Night.hpp:

```
inline
std::string goodnight1(std::string thing1) {
  return "Goodnight, " + thing1 + ".\n";
}
std::string goodnight2(std::string thing1, std::string thing2);
```

Default function arguments

Useful especially when user input is required. It means that you only have to add in some later value when it is absolutely required. All you have to do is put the default argument in the parameter when the function is declared. Here's an example:

Main.cpp:

```
#include <iostream>
#include "coffee.hpp"

int main() {

  // coffee black
  std::cout << make_coffee();
  // coffee with milk
  std::cout << make_coffee(true);
  // coffee with milk and sugar
  std::cout << make_coffee(true, true);
  // coffee with sugar
  std::cout << make_coffee(false, true);
}
```

Coffee.hpp:

```
std::string make_coffee(bool milk = false, bool sugar = false);
//Notice that the default argument is assigned to the parameter
```

Coffee.cpp:

```
#include <string>

std::string make_coffee(bool milk, bool sugar) {
  std::string coffee = "Here's your coffee";
  if (milk and sugar) {
    coffee += " with milk and sugar";
  } else if (milk) {
    coffee += " with milk";
  } else if (sugar) {
```

```
    coffee += " with sugar";
  }
  return coffee + ".\n";
}
```

Function overloading

Useful so that I can do the same operation with different data input types. It requires that the multiple functions has at least one of:

- A different type of parameters.
- A different number of parameters.

They still have to be declared multiple times (with the proper parameter differences) and defined multiple times (with the proper parameter & calculation differences. Example:

Main.cpp:

```
#include <iostream>
#include "num_ops.hpp"

int main() {

  std::cout << fancy_number(12, 3) << "\n";
  std::cout << fancy_number(12, 3, 19) << "\n";
  std::cout << fancy_number(13.5, 3.8) << "\n";
}
```

Num_ops.hpp:

```
int fancy_number(int num1, int num2);
int fancy_number(int num1, int num2, int num3);
int fancy_number(double num1, double num2);
```

Num_ops.cpp:

```
int fancy_number(int num1, int num2) {
  return num1 - num2 + (num1 * num2);
}

int fancy_number(int num1, int num2, int num3) {
  return num1 - num2 - num3 + (num1 * num2 * num3);
}

int fancy_number(double num1, double num2) {
  return num1 - num2 + (num1 * num2);
}
```

Function Templates

There is the option to create function templates which can be a replacement for function overloading. It “slows down compile time but speeds up execution time.” It makes it so that you don’t have to redefine the same function body repeatedly for different data types - note that it does not help with the case where you have different numbers of arguments unless you create a default argument that is discarded in the body of your function definition.

Main.cpp:

```
#include <iostream>
#include "numbers.hpp"

int main() {
  std::cout << get_smallest(100, 60) << "\n";
  std::cout << get_smallest(2543.2, 3254.3) << "\n";
}
```

Numbers.hpp:

```
// Replace repeated declarations with a template
template <typename T>
T get_smallest(T num1,T num2){
    return num2 < num1? num2 : num1;
}
template <typename T>
T average1(T num1,T num2){
    return (num1 + num2) / 2;
}
```

Classes & Objects:

Classes are essentially user-defined data types & serve as a blueprint for objects (for example, *age* can be an instance of *int*). Note that it is possible to have derived classes which inherit all of the objects from the base class. Example of creating a class but doing nothing with it:

Music.cpp:

```
#include <iostream>
#include "song.hpp"

int main() {
  Song elec_relax; //Instantiate object "electric_relaxation" class Song
  elec_relax.add_title("Electric Relaxation"); //add title with method

  std::cout << elec_relax.get_title() << "\n"; //access title with method
```

```
}
```

Song.hpp:

```
#include <string>
// add the Song class here:
class Song {

  std::string title;

public:
  void add_title(std::string new_title);
  std::string get_title();
}; // don't forget this godamn semicolon
```

Song.cpp:

```
#include "song.hpp"
// add Song method definitions here:
void Song::add_title(std::string new_title) {
  title = new_title;
}

std::string Song::get_title() {
  return title;
}
```

Now that there is a class, an object can be created that is of that class. “Instantiate” is the term for creating a new object. Instantiation follows the format of:

```
ClassName object_name;
City atlanta; //example
```

Give attributes with:

```
atlanta.population = 5800000;
```

Can access the attributes/information for an object with predefined methods:

```
std::cout << Atlanta.population(); // outputs 5800000
```

See the music.cpp example above for the instantiation of objects in the Song class.

Public vs Private

By default, everything in a class is private, meaning class members are limited to the scope of the class. This makes it easier to keep data from mistakenly being altered, and abstracts away all the nitty gritty details. The public and private keywords can be used within the class like this:

```
class City {

  int population;

public:
  void add_resident() { //accessible outside of the class
    population++;
  }

private: // this stuff is private (only be referenced within the class?)
  bool is_capital;
};
```

Constructors

Constructors are used to give an object data as soon as it gets created. They have the same name as the class and no return type, because that isn't confusing. For example, if I want to make sure that every City is instantiated with a name and population, the constructor would look something like this:

city.hpp

```
#include "city.hpp"

class City {

  std::string name;
  int population;

public:
  City(std::string new_name, int new_pop);

};
```

city.cpp

```
City::City(std::string new_name, int new_pop)
  // members get initialized to values passed in
  : name(new_name), population(new_pop) {}
```

OR the definition could be this (does the same thing, seems simpler):

```
City::City(std::string new_name, int new_pop) {
  name = new_name;
  population = new_pop;
}
```

Then an instantiation would look like this within main():

```
City atlanta("Atlanta", 5800000);
```

Destructors

Unfortunately you also have to destroy things in C++ because of some bullshit memory management stuff that is taken care of in other languages (Python, Java, C#, Rust?). Destructors have the same name as the class, takes no parameters, and has no return type. It uses the tilda to signify that it is a destructor:

```
City::~City() {
  // any final cleanup
  Std::cout << "Goodbye" << name << "!\n";
}
```

You generally won't need to call a destructor; the destructor will be called automatically in any of the following scenarios:

- The object moves out of scope.
- The object is explicitly deleted.
- When the program ends.

Example of classes and objects dealing in something very near and dear to my heart:
Main.cpp

```
#include <iostream>
#include "yc.hpp"

int main() {

  beer first("AB","Lager",true);
  first.drink();
}
```

Yc.hpp

```
class beer {
  std::string brand;
  std::string type;
  bool full;

public
  Song(std::string new_brand, std::string new_type, bool fullup);
  bool drink();

};
```

Yc.cpp

```cpp
#include "yc.hpp"
#include <iostream>

beer::beer(std::string new_brand, std::string new_type, bool fullup):
brand(new_brand), type(new_type), full(fullup){}

bool drink(){
  full = false;
  std::cout << "You drank the beer.\n";
  return false;
}
```

References (Aliases) & Pointers*:

Dealing with memory like never before.
In C++, a reference variable is an alias for an already existing variable.

- Anything we do to the reference also happens to the original.
- Aliases cannot be changed to alias something else.

```cpp
#include <iostream>

int main() {
  int soda = 99;
  int &pop = soda;
  pop += 1;

  std::cout << soda << "\n"; //returns 100
  std::cout << pop << "\n"; //returns 100
}
```

The biggest benefit of references is that the value can be altered when passed into a function as an argument. Performance wise, it doesn't create a copy of the variable when passed into a function.

```cpp
#include <iostream>

int triple(int &i) {

  i = i * 3;
  return i;
}
```

```
int main() {

  int num = 1;

  std::cout << triple(num) << "\n"; // prints 3
  std::cout << triple(num) << "\n"; // prints 9 because num was modified
}
```

“Const”

The “const” keyword can make sure that a function will NOT change the parameter regardless of whether a reference or pointer is passed to it.

```
#include <iostream>

int square(int const &i) {

  return i * i;
}

int main() {

  int side = 5;
  std::cout << square(side) << "\n"; // prints 25
  std::cout << side << "\n"; // prints 5 - notice that side is unchanged
}
```

Memory Address

The ampersand (&), also known as “address of” operator, can be used to get the memory address:

```
#include <iostream>
int main() {
  int power = 9000;
  std::cout << power << "\n"; // prints 9000
  std::cout << &power << "\n"; // prints 0x7ffef53533404 (hexadecimal)
}
```

A pointer* is a variable that stores the memory address/location of a piece of data. References are newer and exclusive to C++ while pointers are from C. Things I read online recommend

using references as much as possible for a few reasons I can't remember. They are created by placing an asterisk after the data type of the pointer. The asterisk (dereference operator) is also used to obtain the value of the item located at the memory address:

```
#include <iostream>
int main() {

  int power = 9000;
  int* ptr = &power; // Create pointer (ptr) to address (&power) of power
  std::cout << ptr << "\n"; // prints 0x7ffeb05580ec
  std::cout << *ptr << "\n"; // prints 9000
}
```

Nullptr

If a pointer is declared without pointing it to a specific variable, it doesn't contain a specific address -> Danger! If we don't know what address to point to, the *nullptr* is a new feature of C++11 that "provides a typesafe pointer value representing an empty pointer." In old code, *NULL* was used instead of nullptr.

```
int main() {

  int power = 9000;
  int* ptr = nullptr; // Create pointer without specific address
  // Later in the program...
  ptr = &power;
  std::cout << ptr << "\n"; // prints 0x7ffeb05580ec
}
```

A danger with pointers is that you can allocate memory and never deallocate it, causing a memory leak.

Style Guide:

https://github.com/Microsoft/AirSim/blob/master/docs/coding_guidelines.md