1. **Intro to C/C++:**

   C and C++ are programming languages, like Python, but with several major differences aimed towards maximizing speed and minimizing memory usage. This guide is for both languages, since C++ is the same as C but with some extra features, so most of the distinctions between the two languages are omitted here.

   One important difference between C++ and Python is that C++ is a **compiled** language.

   In Python, you can directly run your file from VSCode using the `ctrl-b` command, or open a shell and type Python code directly into it. This is because Python uses an interpreter, so code can be directly run, or entered into a shell.

   For C++, you instead use a command to compile your code into a file called an executable. This executable then can be run in the terminal with another command. These extra steps allow the computer to run our code faster than Python could ever dream of (see the appendix for a brief explanation of why).

   As an example, suppose I have some C++ file called `foo.cpp`. In the terminal, once I navigate to the correct folder, I would compile foo.cpp by typing the following command:

```
g++ -std=c++17 foo.cpp -o foo
```

   The `g++` command is the name of the compiler on my computer. If you have a different C++ compiler, you may need to use a different command. Also, it is possible that you may not have a C++ compiler on your computer at all, in which case you may need to download one from the internet. More information on that in the link below:

   https://gcc.gnu.org/

   The `-std=c++17` specified which version of C++ the compiler should use.

   The `-o foo` is an optional flag (there are several of these that you could provide). This one tells the terminal that once it compiles `foo.cpp`, it should name the executable `foo`. If this is omitted, then the executable will be named `a.out` instead. To run the executable, I would type the following into my terminal:

```
./foo
```

   This would then run the file (just like when you hit command-B or ctrl-B in VSCode for running a Python file), except that once it is done, there is no shell where you can type in more code.

2. **Basic Syntax and Static Typing:**

   The most important syntax differences from Python are as follows:

   Every statement must end with a semicolon.

```
// This is a one-line comment
x += 42;
```

   Whitespace is meaningless. Control blocks start/end with curly braces.

```
/*
    This is a multi-line comment
*/
if (a < b) {
    a = b - a; b = b * 3;
}
```

   All variables, function parameters, and function outputs have **static typing**. This means that their types MUST be declared, and cannot be changed. **C++ strictly enforces these types**; if it detects that you are trying to do something potentially illegal (like writing x + y when x is an int but y is a string) it will crash at compile-time.

```
// This function takes in 2 integers and returns an integer
int minimum(int a, int b) {
    int lower = a;
    if (a > b) {lower = b;}
    return lower;
}
```

   Besides stuff like declaring functions, **no top-level code is allowed**. Everything that "does something" (variable definitions, loops, if statements, etc.) must be inside of a function. When a C++ file is run, it will look for a function called main, which is the only code that will be executed.

   Syntax, type checking, and top level code (amongst other things) are all checked when you run the compiler. If any of these fail, the compilation will fail.

3. **File 1: Hello**

Below is the first C++ file used in the seminar: `hello.cpp`. The components of this file are as follows:

- The `#include` statement works similarly to imports in Python. In this case, `#include <iostream>` allows us to use the printing utilities we'll get to soon...
- The `swapDigits` function takes in an integer and returns the last 2 digits, but swapped. It also makes it positive using the standard function `std::abs`. The function also contains an if statement. Note that the parenthesis around the conditional are **NOT** optional (in Python they are).
- The `main` function is what will be executed when the executable is run in the terminal. After declaring a local variable i, it uses several `std::cout` and `std::cin` statements to get user input and print to the console.
  (a) `std::cout <<` will print whatever follows to the console.
  (b) `std::cin >>` will get user input from the console and store it inside the variable that follows.
  (c) The first `std::cout` prints the string "Please enter a number :"
  (d) The `std::cin` reads a number from the console and stores it in the local variable i (converting it to an int in the process).
  (e) The next two `std::cout` statements print out "Hello world! My favorite number is ", followed by whatever number was inputted (with its digits swapped), then followed by an exclamation point and a newline character.

```cpp
#include <iostream>

int swapDigits(int x) {
    if (x < 0) {
        x = std::abs(x);
    }
    int ones = x % 10;
    int tens = (x / 10) % 10;
    return 10 * ones + tens;
}

int main() {
    int i;

    std::cout << "Please enter a number: ";
    std::cin >> i;
    std::cout << "Hello world! My favorite number is ";
    std::cout << swapDigits(i) << "!\n";
}
```

An example of how to compile and run the file:

```
g++ -std=c++17 hello.cpp -o hello
./hello
-24
```

And here is what that example would output:

```
Hello world! My favorite number is 42!
```

4. **Loops:**

   While loops in C++ work the exact same as Python, except that parenthesis MUST be included around the loop guard. The example below loops through numbers 0 to 9 and prints each of them on a new line).

```
int i = 0;
while (i < 10) {
    std::cout << i << "\n";
    i++; // Shorthand for i += 1
}
```

   For loops in C++ are very different than in Python. They are basically just fancy while loops (there are exceptions to this, but we'll get to that much later).

   The for loop will have the following syntax: `for (initialize; condition; step) {code}`. When the for loop begins, the initialize block will be run (this is often where you declare the variable that you loop over). The loop will continue running so long as the condition block is true. Between each loop iteration, the step code is executed (this is often where you take actions like incrementing i).

   The example below is the same as the while loop example from above, but written with a for loop.

```
for (int i = 0; i < 10; i++)
    std::cout i;
}
```

5. **Command Line Arguments:**

   You can take in arguments from the command line (i.e. when you run the executable). These arguments are passed in as two parameters to the main function. The first parameter is an integer `argc`, the number of command line arguments. The second argument is `argv`, an array of strings, where each string is one of the command line arguments. For now, you can just treat argv like you would a list of strings in Python.

   Consider the snippet of code below:

```
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Number of arguments: " << argc;
    for (int i = 0; i < argc; i++) {
        std::cout << "Argument #" << i << ": " << argv[i];
    }
}
```

   Suppose I compiled the code above, then executed it as follows:

```
./a.out 15 foo
```

   Then it would print out the following:

```
Number of arguments: 3
Argument #1: ./a.out
Argument #2: 15
Argument #3: foo
```

6. **File 2: Digits**

Below is the second C++ file used in the seminar: `digits.cpp`. The components of this file are as follows:

- A `digitCount` function that works the same as in Python
- A `exponentiate` function that calculate `a ** b` (there is no exponent operator in C++)
- A `getKthDigit` function that works the same as in Python
- `main` takes in a number via command line arguments, gets its digit count, and then prints out all of its digits left-to-right with a comma in between

```cpp
#include <iostream>

int digitCount(int n) {
    n = std::abs(n);
    if (n == 0) {return 1;}
    int count = 0;
    while (n > 0) {
        count++;
        n /= 10;
    }
    return count;
}

int exponentiate(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {result *= base;}
    return result;
}

int getKthDigit(int n, int k) {
    return (std::abs(n) / exponentiate(10, k)) % 10;
}


int main(int argc, char **argv) {
    // Atoi converts a string to an integer (i.e. "15" -> 15)
    int n = atoi(argv[1]);
    int count = digitCount(n);

    std::cout << "n: " << n << "\n";
    std::cout << "digitCount(" << n << "): " << count << "\n";
    std::cout << "digits of " << n << ": ";

    for (int k = count-1; k >= 0; k--) {
        std::cout << "" << getKthDigit(n, k);
        if (k > 0) {std::cout << ", ";
        } else {std::cout << "\n";}
    }
}
```

An example of how to compile and run the file:

```
g++ -std=c++17 digits.cpp -o digits
./digits 15112
```

And here is what that example would output:

```
n: 15112
digitCount(15112): 5
digits of 15112: 1, 5, 1, 1, 2
```

7. **Arrays:**

Arrays in C++ are much more difficult to use than lists in Python for several reasons:

- All of the elements must be of the same type
- The length must be set when the array is created, and it cannot be changed later
- There is no way to obtain the length of an array, so the programmer needs to keep track of it
- Returning an array is difficult
- Aliasing works the same as in Python
- 2D arrays are even worse to work with. The number of rows and columns must be known, each of the inner elements must be an array of the same size, and passing them in as arguments to a function is hellish.

The function below is an example of how to work with arrays. It takes in an array of integers A, as well as its length n, and returns its sum.

```
int listSum(int n, int A[]) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum;
}
```

The annoying part is that if we create an array as a local variable in a function, we can't return that array. This is because local variables are stored in a region in memory known as **the stack**, and when a function call returns, the part of the stack corresponding to that function call is consumed, so any arrays that were being stored there cannot be later used.

An array can be created in a couple ways, demonstrated below:

```
// Declares an array of 5 integers
int A[5];

// Declares an array of 5 integers, and sets the first 3
int B[5] = {1, 2, 3};

// Declares an array, lets the compiler infer that the length is 3
int C[] = {4, 5, 6};
```

8. **File 3: Arrays**

Below is the third C++ file used in the seminar: `arrays.cpp`. The components of this file are as follows:

- A `dotProduct` function that takes in 2 int array, and their length, and returns the dot product of their elements.
- A `median` function takes in an array, and its length, and finds the median. It does this by first making a local array B and copying the elements of A into it, and using the destructive `std::sort` function to sort B. The two arguments represent the two memory locations at the beginning and end of the array that are to be sorted (if X is a memory location, then X+1 is the next memory location).
- A `printIntArray` function that takes in an int array, its length, and a prefix/suffix string (as optional arguments), then prints the array like in Python (with the prefix in front and the suffix at the end, if provided).
- `main` creates an array A of size 11 then prints it, creates an array B of 5 elemetns and prints it but with different values of n, creates 2 more arrays C and D of 3 elements and shows their dot product, then creates another array E and finds its median.

```
#include <iostream>

int dotProduct(int A[], int B[], int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        result += A[i] * B[i];
    }
    return result;
}

int median(int A[], int n) {
    int B[n];
    for (int i = 0; i < n; i++) {
        B[i] = A[i];
    }
    // Destructively sorts B from index 0 to index n-1
    std::sort(B, B+n-1);
    if (n % 2 == 0) {
        return (B[n/2] + B[n/2-1]) / 2;
    } else {
        return B[n/2];
    }
}

void printIntArray(int A[], int n,
// These are default arguments (they work the same way in Python)
std::string prefix="", std::string suffix="\n") {
    std::cout << prefix << "[";
    for (int i = 0; i < n; i++) {
        std::cout << A[i];
        if (i < n - 1) {
            std::cout << ", ";
        }
    }
    std::cout << "]" << suffix;
}
```

(Continued on the next page)

```
int main() {

    int A[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
    printIntArray(A, 11);

    int B[5];
    B[0] = 1; B[1] = 5; B[2] = 1; B[3] = 1; B[4] = 2;

    printIntArray(B, 2, "n < len(B): ");
    printIntArray(B, 5, "n = len(B): ");
    printIntArray(B, 7, "n > len(B): ", " (what!?!)\n");


    int C[] = {1, 2, 3};
    int D[] = {4, 5, 6};
    printIntArray(C, 3, "Dot Product: ", " · ");
    printIntArray(D, 3, "", " = ");
    std::cout << dotProduct(C, D, 3) << "\n";

    int E[] = {55, -56, 81, -2, -11, 65, 6, -38, -27, -29};
    printIntArray(E, 10);
    std::cout << "Median: " << median(E, 10) << "\n";
}
```

An example of how to compile and run the file:

```
g++ -std=c++17 arrays.cpp -o arrays
./arrays
```

And here is what that example would output:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
n < len(B): [1, 5]
n = len(B): [1, 5, 1, 1, 2]
n > len(B): [1, 5, 1, 1, 2, 0, 1] (what!?!)
Dot Product: [1, 2, 3] · [4, 5, 6] = 32
[55, -56, 81, -2, -11, 65, 6, -38, -27, -29]
Median: 2
```

9. **Pointers:**

There is an alternative to using arrays: pointers. Pointers are memory addresses that refer to a specific location which you can allocate and de-allocate. This means that you can store data in their (like arrays), but unlike arrays you can return the pointer from a function.

The basic idea is that if a variable x has type `int *` (pronounced "int star"), then it is a pointer to an integer (or multiple integers). Below is an example of a function that takes in a pointer to an integer, accesses the value within, and destructively adds one to it. The return type is `void` because it returns nothing.

```
void incrementPointer(int *p) {
   // "dereferences" the pointer, accessing the value inside
   int x = *p;

   // Sets the value stored inside of p to x+1
   *p = x+1;
}
```

There are (typically) two ways to create a pointer.

The first is if you have a local variable, you can obtain a pointer to it using the address of operator `&`. In the example below, a pointer y to a local variable x is obtained. If incrementPointer(y) is called, then the local variable x is changed.

```
int main() {
    int x = 10;
    int *y = &x;
    incrementPointer(x);
    std::cout << x << "\n"; // Will print 11
}
```

The other way is to allocate a pointer using the `new` keyword. This takes in a type and allocates a pointer of that type. It can also take in a number of elements, in which case it will allocate a pointer to contain each of those elements. That pointer will then work like an array. Importantly, this memory will not be consumed until you de-allocate it (or the program ends), meaning that **you are responsible for deleting it** using the `delete` keyword when you are done with the pointer.

Below is an example that takes in a pointer (and its length) and returns an identical one with all of the elements squared.

```
int *squareElems(int *A, int n) {
    int *result = new int[n];
    for (int i = 0; i < n; i++) {result[i] = A[i] * A[i];}
    return result;
}
int main() {
    int *A = new int[3];
    A[0] = -15; A[1] = 5; A[2] = 8;
    int *B = squareElems(A, 3);

    // Prints -15,5,8
    std::cout << A[0] << ", " << A[1] << ", " << A[2] << "\n";
    // Prints 225,25,64
    std::cout << B[0] << ", " << B[1] << ", " << B[2] << "\n";

    delete[] A;
    delete[] B;
}
```

10. File 4: Pointers

Below is the fourth C++ file used in the seminar: `pointers.cpp`. The components of this file are as follows:

- A `extrema` function that takes in an array (as a pointer), its length, and 2 pointers, then stores the min and max of the array inside of those two pointers.
- An `oddRange` function that takes in a number n, a pointer m, then returns a pointer containing all the odd numbers from 1 to n, with the length stored inside of m.
- A `printIntPointer` function that works the same as before.
- `main` takes in a bunch of numbers via command line arguments, stores them in a pointer, prints it and its min max, then obtains and prints the odd numbers from 1 to 15, before deleting all allocated memory.

```
#include <iostream>

void extrema(int *A, int n, int *min, int *max) {
    *min = A[0];
    *max = A[0];
    for (int i = 1; i < n; i++) {
        if (A[i] < *min) {
            *min = A[i];
        }
        if (A[i] > *max) {
            *max = A[i];
        }
    }
}

int *oddRange(int n, int *m) {
    *m = (n + 1) / 2;
    int *A = new int[*m];
    for (int i = 1; i < n + 1; i+= 2) {
        A[i/2] = i;
    }
    return A;
}

void printIntPointer(int *A, int n,
std::string prefix="", std::string suffix="\n") {
    std::cout << prefix << "[";
    for (int i = 0; i < n; i++) {
        std::cout << A[i];
        if (i < n - 1) {
            std::cout << ", ";
        }
    }
    std::cout << "]" << suffix;
}
```

(Continued on the next page)

```
int main(int argc, char **argv) {
    int n = argc-1;
    int *A = new int[n];
    for (int i = 1; i < argc; i++) {
        A[i-1] = atoi(argv[i]);
    }

    printIntPointer(A, n);

    int min;
    int *max = new int;
    extrema(A, n, &min, max);
    std::cout << "Min(A) = " << min << "\n";
    std::cout << "Max(A) = " << *max << "\n";

    int m;
    int *B = oddRange(15, &m);
    printIntPointer(B, m, "Odds from 1 to 15: ");

    delete[] A;
    delete[] B;
    delete max;
}
```

An example of how to compile and run the file:

```
g++ -std=c++17 pointers.cpp -o pointers
./pointers 55 -56 81 -2 -11 65 6 -38 -27 -29
```

And here is what that example would output:

```
[55, -56, 81, -2, -11, 65, 6, -38, -27, -29]
Min(A) = -56
Max(A) = 81
Odds from 1 to 15: [1, 3, 5, 7, 9, 11, 13, 15]
```

11. **Structs:**
A struct is a user-defined type, similar to a tuple but where each of the elements has a specific name and type. Consider the definition below:

```
struct Point {
    int x;
    int y;
};
```

This defines a new type `struct Point` which contains two elements, known as **fields**. The first is an integer x, the second is an integer y. Because writing out `struct Point` (or longer struct names) may be a pain, you can use a `typedef` statement to give that type a nickname, of sorts. In the example below, `struct Point` is given the nickname `point_t`.

```
typedef struct Point point_t;
```

There are several ways to create and access structs. You can create the struct and initialize its values the same way as a list, or create it on one line then individually set its fields using the dot operator. Both aproaches are shown below.

```
point_t p1 = {15, 112};
point_t p2;
p2.x = 4;
p2.y = 2;
```

But there are more ways to create and handle them, especially if you are dealing with pointers to structs. In the example below, a pointer to a struct is created, its fields are set using arrow syntax instead of dot syntax, then it is deleted.

```
point_t *p3 = new point_t;
// Same as (*p3).x = 42 and (*p3).y = 16
p3->x = 42;
p3->y = 16;
delete p3;
```

12. **Enums:**
Enumerations (enums for short) are another kind of user-defined type. An Enum type has a fixed number of values. A simple example is booleans, which have two values (true or false). Below is an example of a Cent enum which has four possible values. The four values in the enum can now be passed around as values just like true and false. In the example below, they are stored in an array.

```
enum Cent {Penny, Nickel, Dime, Quarter};
Cent wallet[5] = {Penny, Quarter, Penny, Penny, Dime};
```

13. **Overloading:**
In C++, a function can be defined multiple times if each definition has different input types. When the function is called, C++ will figure out which version to call based on the inputs. Consider the min functions below: one finds the min of 2 numbers, the other of an array.

```
int min(int x, int y) {
    if (x < y) {return x;}
    else {return y;}
}

int min(int A[], int n) {
    int res = A[0];
    for (int i = 1; i < n; i++) {
        if (A[i] < res) {res = A[i];}
    }
    return res;
}
```

If `min(x, y)` is called and both x and y are ints, then the first implementation is used. If x is an int array and y is an int, then the second implementation is used.

14. **Vectors:**

Vectors are one of many datastructures that C++ provides as libraries (these are some of the biggest improvements between C and C++). Vectors are much closer to Python lists than normal arrays, but they still require every element to be of the same type. They offer several key conveniences:

- Like arrays, arbitrary elements can still be accessed/modified with indexing.
- Their length can be found with the `.size()` method
- Elements can be appended/popped to/from the end with `.push_back()` and `.pop_back()`
- A vector can be looped over with a for loop, similarly to lists in Python.

Vectors can do even more than this, and there are many other C++ datastructures, including stacks, queues, priority queues, maps (aka dictionaries) and sets.

Examples of some of these are shown below:

```cpp
// Imports the vector library
#include <vector>

// The type of a vector of ints is std::vector<int>
// But for convenience it is nicknamed intVector
typedef std::vector<int> intVector;

int main() {
    // Creates a new int vector (initially empty)
    intVector V;

    // Appends 1, 2, 4, 8, 16, 32, 64 to the vector
    for (int i = 1; i < 100; i *= 2) {
        V.push_back(i);
    }

    // Prints out the length of V
    std::cout << V.size();

    // Loops over each element of V and prints it
    for (int elem : V) {
        std::cout << elem << "\n";
    }

    // Wipes out the elements of the vector
    V.clear();
}
```

15. **File 5: Fruits**

Below is the final C++ file used in the seminar: `fruits.cpp`. The components of this file are as follows:

- Defines a `Color` enum with 6 possible values
- Defines a `Fruit` struct with three fields: a name (which is a `char *`, aka a string), a color (which is a `Color`), and a cost (which is a `float`)
- Uses typedef to create a shorter nickname for `struct Fruit *` and for vectors of `fruit_t`
- `newFruit` takes in a name, color, cost, and creates a new pointer to a struct that will store the three.
- `basketCost` adds up the total cost of all fruits that are in a vector of fruit struct pointers.
- 2 overloaded definitions of `printer`: one prints a single `friut_t`, the other prints a vector of fruits.
- `main` does several things, described in the comments on the next page

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

enum Color {Red, Orange, Yellow, Green, Blue, Purple};

struct Fruit {
    const char *name;
    Color color;
    float cost;};

typedef struct Fruit * fruit_t;
typedef std::vector<fruit_t> fruityVector;

fruit_t newFruit(const char *name, Color color, float cost) {
    fruit_t F = new struct Fruit;
    F->name = name;
    F->color = color;
    F->cost = cost;
    return F;
}

float basketCost(fruityVector B) {
    float result = 0;
    for (fruit_t F : B) {result += F->cost;}
    return result;
}

void printer(fruit_t F) {
    std::cout << "(" << F->name << " $" << F->cost << ")";
}

void printer(fruityVector B) {
    std::cout << "< ";
    for (fruit_t F: B) {
        printer(F);
        std::cout << " ";
    }
    std::cout << ">\n";
}
```

(Continued on the next page)

```
int main(int argc, char **argv) {
    // Creates 5 fruit struct pointers
    fruit_t A = newFruit("apple", Red, 1.32);
    fruit_t O = newFruit("orange", Red, 1.45);
    fruit_t L = newFruit("lemon", Yellow, 0.62);
    fruit_t K = newFruit("kiwi", Green, 2.19);
    fruit_t P = newFruit("pomegranate", Red, 5.92);

    // Creates a vector and appends the 5 structs above to it
    fruityVector catalog;
    catalog.push_back(A);
    catalog.push_back(O);
    catalog.push_back(L);
    catalog.push_back(K);
    catalog.push_back(P);

    // Creates a vector and loops through each string in the command
    // line arguments, appending the corresponding fruit to the vector
    fruityVector basket;
    for (int i = 1; i < argc; i++) {
        for (fruit_t F : catalog) {
            // Returns True if F->name and argv[i] are identical strings
            if (strcmp(F->name, argv[i]) == 0) {basket.push_back(F);}
        }
    }

    // Counts the number of red fruits in the basket vector
    int redCount = 0;
    for (fruit_t F : basket) {
        if (F->color == Red) {
            redCount++;
        }
    }

    // Prints out the entire basket, the # of red fruits, # of total
    // fruits, and the total cost of all the fruits in the basket
    std::cout << "Basket: ";
    printer(basket);
    std::cout << "Red Fruits: " << redCount << "\n";
    std::cout << "Size: " << basket.size() << "\n";
    std::cout << "Cost: $" << basketCost(basket) << "\n";

    // Deletes the struct pointers and cleans up the vectors
    for (fruit_t F : catalog) {delete F;}
    catalog.clear();
    basket.clear();
}
```

An example of how to compile and run the file:

```
g++ -std=c++17 fruits.cpp -o fruits
./fruits apple apple kiwi apple lemon pomegranate apple lemon kiwi
```

And here is what that example would output:

```
Basket: < (apple $1.32) (apple $1.32) (kiwi $2.19) (apple $1.32) (lemon $
    0.62) (pomegranate $5.92) (apple $1.32) (lemon $0.62) (kiwi $2.19) >
Red Fruits: 5
Size: 9
Cost:
```