The Fuss of C++

Step 1

In his book titled *C++ for Java Programmers*, Mark Allen Weiss describes the different mentality between C++ and Java:

"C++'s main priority is getting correct programs to run as fast as it can; incorrect programs are on their own. Java's main priority is not allowing incorrect programs to run; hopefully correct programs run reasonably fast, and the language makes it easier to generate correct programs by restricting some bad programming constructs."

As Weiss hints at, if you are transitioning from Java to C++, you must beware that C++ is not designed to help you write correct code. The safety checks built into Java slow down the execution of your code, but the lack of these safety checks in C++ means that, even if your code's logic is incorrect, you may still be allowed to execute it which can make debugging tricky.

STOP and Think: Based on the descriptions above, why would we use C++ instead of Java to implement data structures in this course?

Step 2

As mentioned before, since we are learning about data structures and thus want to place an emphasis on speed, everything in this text will be done in C++. If you are already comfortable with C++, you don't need much preparation, if any, language-wise. However, if you are *not* already comfortable with C++, this text assumes that you are at least comfortable with Java. As such, this section highlights the key differences one would need to know to transition from Java to C++. First, just to refresh basic syntax tasks, solve the simple challenge below.

CODE CHALLENGE: Basic Syntax Refresher

Given an integer *n*, return the sum of the integers from 1 to *n* (inclusive).

Hint: Even though your solution will technically be in C++, the syntax for this challenge will be 100% identical to Java.

Sample Input:

```
10
```

Sample Output:

```
55
```

To solve this problem please visit https://stepik.org/lesson/26055/step/2

Step 3

We will first discuss the differences between **data types** in C++ and Java, which are actually quite similar.

Number Representation: Like Java, C++ has the `int` and `double` types. However, with `int`s, the number of bytes an `int` variable takes in C++ depends on the machine, whereas in Java, an `int` variable takes exactly 4 bytes, no matter what hardware is used. For the purposes of this class, however, this difference is negligible, since UCSD's ieng6 servers use 4-byte `int`s.

```
int  a; // this variable can range from -2³¹ to +2³¹ - 1
long b; // this variable can range from -2⁶³ to +2⁶³ - 1
char c; // this variable can range from -2⁷  to +2⁷  - 1
```

Unsigned Data Types: In C++, you can specify unsigned data types, whereas Java does not allow unsigned types. Recall that the first bit of an `int`, `long`, `double`, or even `char` is the "sign bit": it represents if the stored value is positive or negative. This is why, in both languages, a regular `int` (which contains 4 bytes, or 4*8 = 32 bits) ranges from -2^{31} to $+2^{31}-1$ (31 bits to represent the magnitude of the number, and 1 bit to represent the sign). In C++, if you specify a variable to be an `unsigned int` instead of a regular `int`, its value will be able to range from 0 to $+2^{32}-1$.

```
unsigned int  a; // this variable can range from 0 to +2³² - 1
unsigned long b; // this variable can range from 0 to +2⁶⁴ - 1
unsigned char c; // this variable can range from 0 to +2⁸  - 1
```

Booleans: The equivalent of the Java `boolean` data type is simply `bool`. The usage of a C++ `bool` is the same as the usage of a Java `boolean`.

```
bool havingFun = true;
```

Strings: In C++, the `string` type is very similar to Java's `String` class. However, there are a few key differences. First, Java `Strings` are immutable. On the other hand, one can modify C++ `strings`. Next, the substring command in C++ is `substr`, where `s.substring(i,n)` returns the substring of `s` that starts at position `i` and is of length `n`. Also, in Java, `String` objects can be concatenated with any other object (and the other non-`String` object will automatically be converted to a `String`), but in C++, `string` objects can only be concatenated with other `string` objects (the conversion is not automatically done for you).

```
string message = "Hi, Niema!";
string name = message.substring(7,5); // name would have a value of "Niema"
```

Comparing Objects: To compare objects in C++, you simply use the relational operators `== != < <= > >=`. In Java, you only use the relational operators to compare primitives and you use .equals to compare objects. But in C++, you can do something special. You can "overload" the relational operators, meaning you can specify custom methods that get called when the relational operators are applied to objects. So in C++ you use relational operators to compare *everything* (including non-primitives). We'll talk about operator overloading in more detail later.

```
string name1 = "Niema";
string name2 = "Liz";
bool sameName = (name1 == name2); // sameName would have a value of 0, or false
```

Step 4

Next, we will discuss the differences between **variables** in Java and C++. Note that variables in C++ can actually be much more complex than what we cover here, but the more complicated details are out of the scope of this text. This is just to teach you the basics.

Variable Safety: In Java, if you declare a local variable but do not initialize it before you try to use it, the compiler will throw an error. However, in C++, if you declare a local variable but do not initialize it before you try to use it, the compiler will NOT throw an error! It will simply use whatever random "garbage data" happened to be at that location in memory! This can cause a LOT of headache when debugging your code, so always remember to initialize your variables in C++! Also, in Java, if you attempt to "demote" a variable (i.e., store it in a smaller datatype) without explicitly casting it as the lower datatype, the compiler will throw an error. However, in C++, the compiler will not complain! Below is perfectly valid C++ code in which we do both: we declare two variables and use them without initializing them, and we then demote to a smaller datatype without explicitly typecasting.

```
int harry; // dummy variable 1
int lloyd; // dummy variable 2
bool dumbAndDumber = (harry + lloyd); // C++ will allow this!
```

Global Variables: In Java, all variables must be declared either within a class or within a method. In C++, however, variables can be declared outside of functions and classes. These "global variables" can be accessed from any function in a program, which makes them difficult to manage, so try to avoid using them.

```
bool dummy = true;
class DummyClass {
    // some stuff here
};
int main() {
    cout << dummy; // this is valid
}
```

Constant Variables: In Java, a variable can be made so that it cannot be reassigned by using the `final` keyword. In C++, the equivalent keyword is `const`, though there are some subtle differences between Java's `final` and C++'s `const`. In Java `final` simply prevents the variable from being reassigned. If the data itself is mutable, it can still be changed. In C++ for a variable that directly references (mutable) data, `const` will prevent that data from being changed. Actually, the C++ `const` keyword can be a bit tricky, and we'll discuss it in more detail in a few steps.

```
int main() {
    const string DUMMY_NAME = "Harry";  // DUMMY_NAME cannot be reassigned and "Harry" cannot be
modified
    DUMMY_NAME = "Barry";  // This is not allowed!  But if DUMMY_NAME were not const it would be
OK
    DUMMY_NAME[0] = 'L';   // This is not allowed!  But if DUMMY_NAME were not const it would be
OK
}
```

Step 5

Now, we will discuss the differences between **classes** in Java and C++. To help exemplify the differences, we will write a simple Student class in both Java and C++:

```
class Student { // Java
    public static int numStudents = 0;           // declare and define static variable
    private String name;                         // declare instance variable

    public Student(String n) { /* CODE */ }      // declare and define constructor

    public void setName(String n) { /* CODE */ } // declare and define setter method
    public String getName() { /* CODE */ }       // declare and define getter method
}
```

```
class Student { // C++
    public:
        static int numStudents;                    // declare static variable

        Student(string n);                         // declare constructor

        void setName(string n);                    // declare setter method
        string getName() const;                    // declare getter method

    private:
        string name;                               // declare instance variable
};

int Student::numStudents = 0;                      // define static variable

Student::Student(string n) { /* CODE */ }          // define constructor

void Student::setName(string n) { /* CODE */ }   // define setter method
string Student::getName() const { /* CODE */ }   // define getter method
```

- In Java, each individual item must be declared as `public` or `private`, but in C++, we have public and private *sections*, started by the keywords `public` and `private`, respectively.
- In Java, we actually fill out the methods within the class, but in C++, we only declare the methods, and the actual implementations are listed separately outside of the class (prefixed by the class name, with the `::` operator separating the class name and the method name)
- In C++, accessor (or "getter") methods are tagged with the keyword `const`, which prevents the method from modifying instance variables
- In C++, there is a semicolon after the class's closing bracket

Also, note that, in C++, for the sake of convenience, you can initialize non-static variables of the object using a **member initializer list** when you define the constructor. Note that, because static variables cannot be initialized in constructors in C++, you can *only* use the member initializer list to initialize non-static variables of an object. Also note that any instance variables that are declared `const` can be initialized *only* in an initializer list; if you try to set their value in the body of the constructor, the compiler will complain. Below is an example of the syntax for the same `Student` class described above (but with the setter and getter methods omitted for the sake of simplification):

```
class Student { // C++
    public:
        static int numStudents;         // declare static variable
        Student(string n);              // declare constructor

    private:
        string name;                    // declare instance variable
};
int Student::numStudents = 0;           // define static variable

Student::Student(string n) : name(n) { // define constructor using member initializer list for
instance var
    numStudents++;
}
```

To finish off the comparison of classes in C++ versus in Java, we want to introduce the notion of `.cpp` and `.h` files in C++. In Java, you write *all* of your code (declarations *and* definitions) in a `.java` file, which must have a class defined (where the class must have the same name as the filename of the `.java` file, minus the file extension).

In C++, however, you can have your code split between a `.cpp` file (known as a "source file", and sometimes using the extension .cc) and a `.h` file (known as a "header file"). In the `.h` file, you will *declare* your classes and functions, but you will not actually define them. Then, in the `.cpp` file, you will actually fill in the bodies of your functions.

It might seem inconvenient to have a single series of logic split between two files, but the reason behind this is to be able to distribute your header file freely so that people can have a map of how to use your product (i.e., they will have access to the declarations of all of your classes and functions, so they will know *how* they would be able to use your full code) without any fear of anyone stealing your implementation (because all of your actual code is in the source .cpp file). It also makes the compilation process simpler and faster.

Below is an example of how we would split up the previously described `Student` class into a `.h` and `.cpp` file:

Student.h

```
class Student {
    public:
        static int numStudents;         // declare static variable
        Student(string n);              // declare constructor

    private:
        string name;                    // declare instance variable
};
```

Student.cpp

```
int Student::numStudents = 0;          // define static variable

Student::Student(string n) : name(n) { // define constructor using member initializer list for
instance var
    numStudents++;
}
```

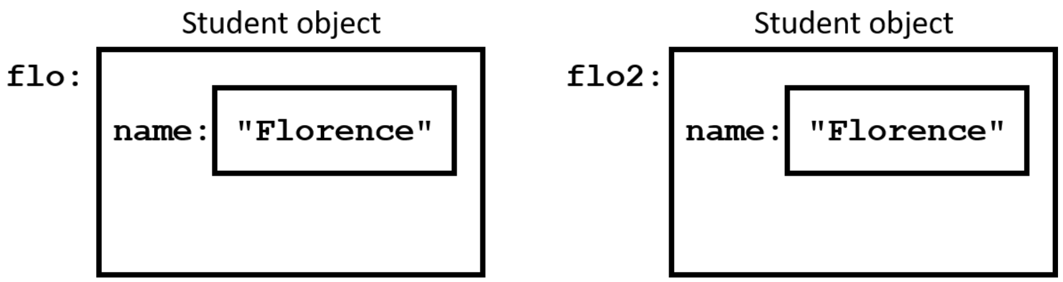
Step 6

We will now discuss the concepts of **references** and **pointers**, and how each concept is used (or not used) in Java and C++.

Objects vs. primitives in Java: In Java, the rules for what is stored in a variable are simple: all object variables store *object references* while primitive type variables store values directly. In Java, assignment is done by value, so when you assign a primitive variable's value to another, the actual value is copied, but when it is an object variable, the reference is copied and you get two references to the same object.

Objects and primitives in C++: In C++ on the other hand, there is no such distinction between object and primitive variables. By default *all* variables, both primitives and objects, actually hold *values*, NOT object references. C++, like Java, does assignment by value. However, this means that in C++ when one object variable is assigned to another, a copy of the entire object is made (like calling `clone` in Java).

```
Student flo("Florence"); // Creates Student object with name "Florence" and stores as variable
'flo'
                         // Note that we do NOT use the keyword 'new' to create the object.
Student flo2 = flo;      // flo2 stores a copy of the Student, with the same name
```



So by default, C++ stores variable values directly, no matter their type, and assignments are done by copying the data. This includes passing parameters to a function. By default the parameter variables get copies of the data in their arguments.

But what if you want to have two ways to access the same data and avoid this copying-the-data behavior? C++ provides two different concepts that give you related but subtly different ways to do this: references and pointers.

References in C++: C++ references are NOT the same as Java references. Although they are related, how they are used and their syntax are pretty different, so it's best if you simply think of them as different concepts.

References in C++ are simply aliases for existing variables. When you define a reference variable in C++, the variable is treated exactly as another name as the variable you set it to. Thus when you modify the reference variable, you modify the original variable as well without needing to do anything special, as shown in the example below.

The syntax for creating a reference variable in C++ is to place a & after the type name in the variable declaration. If `T` is some type, then `T&` is the syntax to declare a reference to a `T` variable. Reference declarations must be combined with assignments except in the case of function parameters (discussed further on the next page).

```
Student lloyd("Lloyd"); // creates Student object with name "Lloyd" and stores as variable 'lloyd'
Student & ref = lloyd;  // creates reference to 'lloyd' called 'ref'
Student harry("Harry"); // creates Student object with name "Harry" and stores as variable 'harry'
```

The picture below shows what the objects and variables look like so far:
[PICTURE OF WHAT THIS LOOKS LIKE SO FAR]
Now we can execute the following lines:

```
ref = harry;            // 'ref' is now a copy of 'harry', so 'lloyd' is ALSO now a copy of
'harry'
cout << lloyd.name;     // this would print "Harry"
```

And the picture below shows the result.

[PICTURE HERE]

There are two main uses for C++ references: parameter passing, and aliasing long variable names. In many cases it's extremely useful not to make copies of objects when they are passed to functions, either because you want the function to be able to modify the data in the object, or because you want to avoid wasting time with the copy. Parameter passing will be discussed in more detail in the next page.

Finally, note that this is an overly simple explanation of references. In particular, C++11 and on has a notion of lvalue references (which are what we showed above) and rvalue references. Don't worry about these subtleties if you don't want to. Throughout this book we'll use only "classic" (lvalue) references like what we described above. Though if you want to learn more, there are certainly plenty of websites that would be happy to explain this in more detail!

Pointers in C++: Pointers in C++ are actually quite similar to references in Java. They are variables that store the memory address of some data, instead of storing the data directly. That is, their *value* is a memory address.

If `T` is some type, then `T*` is the syntax to declare a pointer to a `T` variable. A pointer variable can be initialized with `NULL`, with the value of another pointer variable, with the memory address of another variable, or with a call to `new`. The memory address of a variable can be attained by placing the `&` symbol before the variable name. To access the object to which a pointer points, you must "dereference" the pointer by placing the `*` symbol before the variable name. To access an instance variable of the o

bject to which a pointer points, you can either dereference the pointer and access the instance variable using the `.` symbol as normal, or you can use the `->` operator on the pointer directly (which is the more conventional way).

```
Student lloyd("Lloyd");                // initialize Student object
Student* harry = new Student("Harry"); // initialize Student pointer
Student* ptr1 = &lloyd;                // initialize ptr1 to store the address of 'lloyd'
Student* ptr2 = harry;                 // initialize Student pointer pointing to same object as
'harry'
cout << (*ptr1).name;                  // prints "Lloyd"
cout << ptr2->name;                    // prints "Harry"
```

Beware of the nomenclature used with pointers. The nomencl

ature goes as follows: we can either say that a pointer *points to* an object, or we can say that a pointer *stores the address* of an object. For example, in the code above, look at the line where we initialize `ptr2`. After doing the assignment `ptr2 = harry`, we can either say "`ptr2` *points to* `harry`", or we can say "`ptr2` *stores the address* of `harry`". However, it would be inaccurate to say that "`ptr2` *points to the address of* `harry`". The reason why we're bringing attention to this seemingly meaningless formality is because, in C++, you can actually have a pointer that points to another pointer! For example, the following lines of code are perfectly valid, and the notion of a "pointer to a pointer" can actually be useful in many contexts:

```
Student lloyd("Lloyd");   // initialize Student object
Student* dumb = &lloyd;   // initialize Student pointer to store address of 'lloyd'
Student** dumber = &dumb; // i

nitialize Student pointer pointer to store address of 'dumb'
```

Also, in Java, since all non-primitive data types function in a manner similar to C++ pointers, it was very easy to define a class and have it contain a reference to another object of the same class. However, in C++, because variables hold *objects* by default, a class cannot have member variables of its own type. For example, imagine we are designing a `Human` class, and one of the member variables is `parent`, which represents this `Human` object's parent. If we were to make `parent` be of type `Human`, when we try to create a `Human` object, it will automatically try to create `parent`, which is also of type `Human`. When it tries to create `parent`, the resulting `Human` object will also automatically to create its own `parent`. This cycle of recursion would be infinite, so the code would simply crash. Instead, we need `parent` to be of type `Human*`.

Object creation (and destruction): In Java, to invoke an object's constructor, you must use the new operator. This will create a new object (on the heap) and return a reference to this newly created object. On the other hand in C++, you often do *not* use the new operator to call the object's constructor and create a new instance of the object. You simply supply the construction parameters after the variable name. This is called creating an object using "static" creation, or creating the object "statically". As discussed above, this will create a new object whose value (and *not* a Java-style reference) is stored in the variable to which it is assigned.

In Java, all objects are destroyed automatically by the garbage collector when there is no longer a way to reach them. In C++, objects created statically are automatically destroyed (via a call to their destructor,

```
Turtle harry(0,0);
Turtle lloyd = harry;
harry.move(1,1); // this will only modify harry, NOT lloyd!
```

References: When you define a reference variable in C++, the functionality is NOT the same as in Java. In C++, a reference is treated exactly as another name as the variable you set it to. If `T` is some type, then `T&` is the syntax to declare a reference to a `T` variable. Thus, when you modify the reference variable, you modify the original variable as well.

```
Student lloyd("Lloyd"); // creates Student object with name "Lloyd" and stores as variable 'lloyd'
Student & ref = lloyd;  // creates reference to

'lloyd' called 'ref'
Student harry("Harry"); // creates Student object with name "Harry" and stores as variable 'harry'
ref = harry;            // 'ref' is now a copy of 'harry', so 'lloyd' is ALSO now a copy of
'harry'
cout << lloyd.name;     // this would print "Harry"
```

Pointers: To obtain the functionality seen in Java variables, one needs to use C++ pointers. I

Step 7

EXERCISE BREAK: What would be outputted after the execution of the following lines of code?

```
Student a("Lloyd");
Student & b = a;
Student * c = new Student("Harry");
b = *c;
Student d = a;
cout << d.name;
```

To solve this problem please visit https://stepik.org/lesson/26055/step/7

Step 8

In the previous step, we briefly mentioned the `const` keyword, but what does it actually mean? In general, `const` implies that something will not be allowed to be changed. Of course, this sounds vague, and the reason why we're keeping the general definition vague is because the actual restrictions that come about from the `const` keyword depend on where it is placed with respect to the variable it is modifying.

For variables in general, you can place the `const` keyword before or after the type, and the `const` keyword makes it so that the variable **can never be modified after initialization**. Below is an example using `int` objects:

```
const int a = 5; // 'a' cannot be modified after this: it will always have a value of 5
int const b = 6; // 'b' cannot be modified after this: it will always have a value of 6
```

With **pointers**, the `const` keyword becomes a bit trickier. Below are the different places we can place the `const` keyword and a description of the result:

```
int a = 5;                     // create a regular int
int b = 6;                     // create a regular int
const int * ptr1 = &a;         // can change what ptr1 points to, but can't modify the actual object
itself
int const * ptr2 = &a;         // equivalent to ptr1
int * const ptr3 = &a;         // can modify the object itself, but can't change what ptr3 points to
const int * const ptr4 = &a; // can't change what ptr2 points to AND can't modify the actual
object itself

ptr1 = &b;                     // valid, because I CAN change what ptr1 points to
*pr1 = 7;                      // NOT valid, because I CAN'T modify the object itself

*ptr3 = 7;                     // valid, because I CAN modify the object itself
ptr3 = &b;                     // NOT valid, because I CAN'T change what ptr3 points to

ptr4 = &b;                     // NOT valid, because I CAN'T change what ptr4 points to
*ptr4 = 7;                     // NOT valid, because I can't modify the object itself
```

With **references**, the `const` keyword isn't too complicated. Basically, it prevents modifying the object being referenced via the `const` reference. Below are some examples with explanations:

```
int a = 5;             // create a regular int
const int b = 6;       // create a const int

const int & ref1 = a; // creates a const reference to 'a' (can't modify the object using ref1)
int const & ref2 = a; // equivalent to ref1

ref1 = 7;              // NOT valid, because ref1 can't modify the object

const int & ref3 = b; // valid, because you can have const reference to a const object
int & ref4 = b;        // NOT valid, because you CAN'T have a non-const reference to a const object

int & const ref5 = a; // invalid syntax (const must come before the & symbol)
```

As mentioned previously, **functions** can be `const` as well. By making a function `const`, you are enforcing that the function can *only* modify *local* variables, or variables whose scope only exists *within* the function. You *cannot* modify anything that exists outside of the `const` function.

Step 9

Next, the differences between **functions** in Java and C++.

Global Functions: In Java, every function must either be an instance method or a static function of a class. C++ supports both of these cases, but it also supports functions that are not part of any class, called "global functions". Typically, every functional C++ program starts with the global function `main`:

```
int main() {
    // CODE
}
```

Notice that this `main` method has a return value (`int`), whereas the `main` methods of Java are `void`. By convention, a C++ `main` method returns 0 if it completed successfully, or some non-zero integer otherwise.

Passing Parameters: C++ has two parameter-passing mechanisms: *call by value* (as in Java) and *call by reference*. When an object is passed by value, since C++ objects are not references to objects, the function receives a copy of the actual argument. As a result, the function cannot modify the original. If we want to be able to modify the original object, we must pass the object by reference, which we can do by adding an `&` after the parameter type.

Note that, even if we do not intend to modify the original object, there are still times where we would want to pass by reference. For example, in the world of Bioinformatics, programs often represent genomes as string objects and perform various algorithms on these strings. The human genome is roughly 3.3 billion letters long, so a string object containing the entire human genome would be roughly 3.3 GB large. Even if we have no intention of modifying this object in whatever method we will pass it in, we want to pass by reference so that we do not accidentally create a copy of this absurdly large object.

```
double gcContent(string & genome) {
    // CODE
}
```

In short, in C++, you always use call by reference when a function needs to modify a parameter, and you still might want to use call by reference in other situations as well.

Step 10

Next, we will discuss C++ **vectors**.

Vectors: The C++ `vector` has the best features of the `array` and `ArrayList` in Java. A C++ `vector` has convenient elemental access (using the familiar `[]` operator) and can grow dynamically. If `T` is some type, then `vector<T>` is a dynamic array of elements of type `T`. A `vector` also has the convenient "push" and "pop" functions of a stack (a data type that will be further discussed later in this text), where you can add an element to the back of the `vector` using the `push_back` function, and you can remove the last element of the vector using the `pop_back` function.

```
vector<int> a;      // a vector that is initially empty
vector<int> a(100); // a vector that initially contains 100 elements
a.push_back(0);     // add 0 to the end of a
a.pop_back();       // remove the last element of a
cout << a[10];      // output the element at index 10 of a
```

Vector Indexing: Regarding indexing, the valid indices are between 0 and a.size()-1 (just like in Java). However, unlike in Java, there is no runtime check for legal indices, so accessing an illegal index could cause you to access garbage data without even realizing it.

Memory: Like in a Java `array`, the elements stored in a C++ `vector` are contiguous in memory (i.e., the elements all show up right after one another). Regarding memory storage, like all other C++ objects, a `vector` is a value, meaning the elements of the `vector` are values. If one `vector` is assigned to another, all elements are copied (unlike in Java, where you would simply have another reference to the same `array` object).

```
vector<int> a(100); // a vector that initially contains 100 elements
vector<int> b = a;  // b is now a copy of a, so all of a's elements are copied
```

Step 11

CODE CHALLENGE: Creating a Vector

Given an integer *n*, create a new vector, add all of the numbers from 1 to *n* (inclusive) to the vector, and then return the vector.

Sample Input:

```
10
```

Sample Output:

```
1 2 3 4 5 6 7 8 9 10
```

To solve this problem please visit https://stepik.org/lesson/26055/step/11

Step 12

Lastly, we will discuss **input** and **output** in C++, which is much simpler than in Java.

IO Keywords/Operators/Functions: In C++, the standard output, standard input, and standard error streams are represented by the `cout`, `cin`, and `cerr` objects, respectively. Also, newline characters can be outputted using the `endl` keyword. You use the << operator to write to standard output (or to any `osteam`, for that matter):

```
cout << "Hello, world!" << endl << "My name is Lloyd!" << endl;
```

and you use the >> operator to read from standard input (or any `istream`, for that matter):

```
int n;                      // declare the variable to hold the input
cout << "Please enter n: "; // prompt user
cin >> n;                   // read user input and store in variable n
```

The `getline` method reads an entire line of input (from any `istream`):

```
string userInput;        // declare the variable to hold the input
getline(cin, userInput); // read a single line from cin and store it in userInput
```

End of Input: If the end of input has been reached, or if something could not be read correctly, the stream is set to a failed state, which you can test for using the `fail` method:

```
int n;
cin >> n;
if(cin.fail()) {
    cerr << "Bad input!" << endl;
}
```

Step 13

At this point, you should hopefully be comfortable with C++ and the nuances between it and Java, at least to the point where you will be able to recognize issues with C++ code. If you feel that you are not yet at this point, please re-read this section and search the internet for any concepts you do not fully grasp.

Below are definitions for a BSTNode class in both Java and C++:

```
public class BSTNode { // Java
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode(int d) {
        data = d;
    }
}
```

```
class BSTNode { // C++
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode(const int & d) {
        data = d;
    }
};
```

EXCERCISE BREAK: Which of the following are problems with the C++ implementation above? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/26055/step/13

Step 14

Now that we have reviewed the main basic syntax and variable differences between C++ and Java, we will now discuss how **generic programming** is implemented in C++.

With data structures, recall that we do not always know what data types we will be storing, but whatever data structures we implement should work exactly the same if we store `ints`, `longs`, `floats`, `strings`, etc. In Java, we could use "generics", such as in the following example:

```
class Node<Data> { // Java Node class, with generic type "Data" specified
    public static final Data data;
    public Node(Data d) {
        data = d;
    }
}
```

```
Node<Student> a = new Node<Student>(exampleStudent); // a.data is a variable of type Student
Node<String>  b = new Node<String>(exampleString);   // b.data is a variable of type String
```

In C++, the equivalent of this is the use of **templates**.

```
template<typename Data> // specified generic type "Data"
class Node {            // C++ Node class, which can now use the type "Data"
    public:
        Data const data;
        Node(const Data & d) : data(d) {}
};
```

```
Node<Student> a(exampleStudent); // a.data is a variable of type Student
Node<string>  b(exampleString);  // b.data is a variable of type string
```

Step 15

EXERCISE BREAK: Given the C++ BSTNode class below, how would you create a **pointer** to a BSTNode with integer data?

```
template<typename Data>
class BSTNode {
    public:
        BSTNode<Data>* left;
        BSTNode<Data>* right;
        BSTNode<Data>* parent;
        Data const data;

        BSTNode( const Data & d ) : data(d) {
            left = right = parent = 0;
        }
};
```

To solve this problem please visit https://stepik.org/lesson/26055/step/15

Step 16

In summation, C++ is a very powerful, but very tricky, programming language. There are very few safety checks because the main priority of the language is speed, and it is up to you to ensure that your code is correct and safe.

In Java, if your code throws a runtime error, the Java Runtime Environment gives you very detailed information about where in your code the error occurred and what the error was. In C++, however, there is a very good chance that your runtime error will simply say `SEGMENTATION FAULT`. Many Java programmers who transition to C++ are frustrated with the lack of detail in C++ runtime errors in comparison to Java runtime errors, so if you fall into this boat, do not feel discouraged! Be sure to read up on how to debug your code using `gdb` as it will certainly come in handy when you work on the programming assignments.