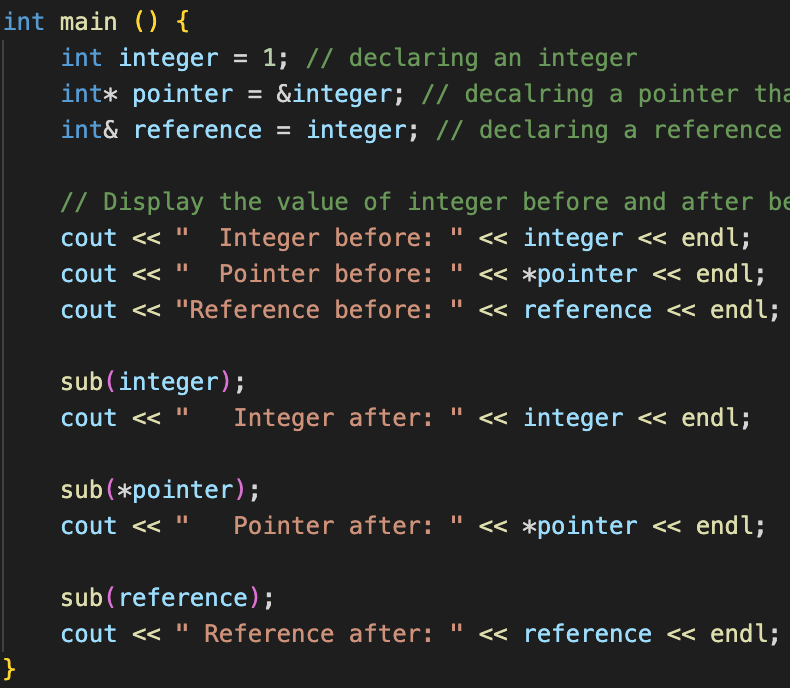
Matt Burton

**Parameter Passing in C++ and Java**

The syntax and semantics of parameter passing in C++ and Java have many similarities and differences, and their respective design choices have a significant influence on memory management and code readability. The most significant difference is that Java prevents developers from explicitly using pointers to reference data. Instead, Java uses references for objects and arrays and keeps memory manipulation under the hood to prevent developers from causing issues. Generally, C++ allows programmers to manipulate memory more directly while Java abstracts the exact details regarding data and memory for the benefit of developers. Many languages have made critical design decisions based on parameter passing, and the differences between Java and C++ illustrate how important these decisions are.

As mentioned in the background information for assignment two, there are five different parameter passing methods for passing parameters in distinct ways.

* **Call-by-value**: Supported by both C++ and Java for primitive data types.
* **Call-by-reference:** Supported for all data types in C++, but only arrays and objects in Java.
* **Call-by-value:** Supported by both C++ and Java.
* **Call-by-name and call-by-result:** Neither C++ nor Java support these parameter-passing methods.Image 1
  

Additionally, parameter passing can become confusing as a result of scope rules, aliasing, and side effects. Regarding scope, both C++ and Java modify the most local instance of any variable of the same name with multiple declarations. For example, in the example to the right, a variable called “integer” is declared twice, once in the global scope and a second time within the main method. Further references to this variable within the main method only access and modify the local variable and ignore the global variable entirely. This is the same behavior in both C++ and Java, and this design decision is straightforward enough that it makes sense why this issue of scope didn’t change between C++ to Java. Furthermore, aliasing and side effects are prevalent and problematic in C++ as a result of pointers. In the example to the left, a reference and pointer were created based on the initial variable declaration, both with the capacity to alter the initial variable’s value. This follows C++’s pattern of allowing developers to directly manipulate memory more than they should, as changing the data via any of these options will alter the data of all three references.Java makes a limiting decision to remove pointers entirely which takes away some of the freedom developers have in C++, but it ultimately saves them from making mistakes that are difficult to track down. Additional experiments revealed further complications that arise from C++’s pointers. Specifically, when declaring a pointer with a reference to memory and then deleting that location, C++ delivers unpredictable results when trying to access that pointer. The known workaround is to assign the value of an undefined pointer to null, but ultimately this is just another area where developers can mess up and reason why Java doesn’t support pointers. 

Generally, I was able to run more varied experiments in C++ than in Java because of its support for pointers. As a result of those uniquely C++-based tests, it is obvious why Java opted to obscure memory referencing behind objects and arrays, as many pointer-based results are confusing and unintuitive, especially in large-scale projects. The change from C++ to Java with regards to parameter passing and memory management was significant and for the better for developers and managers. The intended purpose of Java code is clearer than C++, and there are many fewer issues to carefully avoid as in C++.