```
X's value is: 10
X's address is: 0x7fff4c608d10
Alias's value is: 10
Alias's address is: 0x7fff4c608d14
Incrementing alias
X's value is: 10
X's address is: 0x7fff4c608d10
Alias's value is: 11
Alias's address is: 0x7fff4c608d14
```

One of the most simple concepts we can address in programming is the notion of aliasing, or setting a variable equal to another variable. C++ chooses to handle this concept by creating a clone of the variable and provisioning a new section of memory to hold the value of this clone. To prove this, we created the program **int_alias.cpp** which provisioned a variable **X** in the memory address **0x7fff4c608d10** and set it's value to **10**. Next we created an **alias** variable and set it equal to **X**. The resulting address and value of **alias** is the address **0x7fff4c608d14** with the value of **10**. This means that instead of referring to the same object upon aliasing, C++ actually provisions a clone of the object somewhere else in memory. This is also validated by incrementing the **alias**'s value and this change not being persisted within the **X** variable. A potential weakness of aliasing in C++ is some people may assume that when you set one variable is set to another you will be talking about the same object. This is not the case in C++ and could lead to logical errors that a compiler would not flag upon compiling.

```
Main Function
-------------
Value of number in main is: 1234
Address of number(ptr) in main is: 0x7fff2316191c

parameterized_int_function Function
-------------------------------
The value of number(number) in parameterized_int_function is: 1234
The address of number(&number) in parameterized_int_function is: 0x7fff231618fc
Incrementing the number in parameterized_int_function
The new value of number(number) in parameterized_int_function is: 1235
The address of number(&number) in parameterized_int_function is: 0x7fff231618fc

Main Function
-------------
Value of number in main after parameterized_int_function is: 1234
Address of number(ptr) in main after parameterized_int_function is: 0x7fff2316191c
```

Another fundamental concept of programming is the notion of passing parameters into functions. One of the ways we can pass parameters is by passing the actual object itself through a call by value method. C++ handles this by cloning passed in parameters as a new variable in the scope level of the new function. This means the object being modified within the function is different than the object that was passed into the function. To test this, we created **int_parameter.cpp** which created a variable **number** with the value **1234** and printed out it's address **0x7fff2316191c** in main. This variable was passed into the function **parameterized_int_function** where its value, **1234,** and address, **0x7fff231618fc**, were printed out. Clearly we can see the variable **number** within the **parameterized_int_function** is not the same **number** variable from **main** because the address in memory is different. This proves the **number** variable was cloned into the scope level of **parameterized_int_function** with the same value as the **number** variable from **main**. This was further validated by incrementing the **number** variable within the **parameterized_int_function** and not seeing these changes persisted to the **number** variable within **main**, indicating that all of these changes were limited to the scope level of **parameterized_int_function**. A potential weakness of passing actual objects is people might expect the object from the caller function to be the same object within the callee function and this is not the case in C++. C++ will clone and nest a new variable in the callee function making any modifications to the variable limited to the scope of the callee function. This is not initially clear and could cause logical errors in a similar fashion to aliasing.

Another means of passing parameters is by pointers. C++ can handle the passing of parameters by passing the address in memory of variables. To demonstrate this, we created **pointer_parameter.cpp** which created an integer variable **number** within **main** with the value **1234**. We then created a pointer to this variable's address in memory and passed it as a parameter, which is a call by address method, to the function **parameterized_pointer_function**. In **parameterized_pointer_function**, we are able to print out the value of **number** from **main** via dereferencing the pointer and also print out the memory address of **number**. We can clearly see the memory address of **number** is **0x7fff3c1abcbc** in **main** and **parameterized_pointer_function** which means both functions are referring to exactly the same object. We validate this is the case by incrementing **number** from within **parameterized_pointer_function** and printing out it's value there and in **main** once the function has finished. In both cases the value is **1235**, meaning the changes in **parameterized_pointer_function** persist beyond the scope of **parameterized_pointer_function**. A potential downside to directly accessing memory addresses is there are a ton of opportunities for users to mess up and get a wide variety of errors such as memory leaks, segmentation faults and buffer overflows. You do get a lot more granular control with these mechanisms but with more control means more room for error. <insert java stuff here,if applicable>.

```
Main Function
-------------
Value of number within main is: 1234
Address of number(ptr) within main is: 0x7fff3c1abcbc

parameterized_pointer_function Function
-------------------------------
The value of number(*pointer) within parameterized_pointer_function is: 1234
The address of number(pointer) within parameterized_pointer_function is: 0x7fff3c1abcbc
Attempt to increment the value of number(*pointer) within parameterized_pointer_function
The value of number(*pointer) within parameterized_pointer_function is now: 1235
The address of number(pointer) within parameterized_pointer_function is now: 0x7fff3c1abcbc

Main Function
-------------
Value of number within main is now: 1235
```

A final way we can pass parameters is through a pass by reference means. To demonstrate this, we created **int_reference.cpp** which approached the problem in the exact same way as **pointer_parameter.cpp**. This time, we made the **parameterized_reference_function** which received a reference to the **number** variable from main. By following the same steps as before, we were able to access and modify the same **number** variable in both scopes and we know this because the addresses in memory were identical throughout the experience. Errors with passing by reference still include potential segmentation faults, as we are still fundamentally dealing with the address of the variable in memory.

```
Main Function
-------------
Value of number within main is: 1234
Address of number(ptr) within main is: 0x7ffcfe537ddc

parameterized_reference_function Function
-------------------------------
The value of number(number) within parameterized_reference_function is: 1234
The address of number(&number) within parameterized_reference_function is: 0x7ffcfe537ddc
Attempt to increment the value of number(*pointer) within parameterized_reference_function
The value of number(*pointer) within parameterized_reference_function is now: 1235
The address of number(pointer) within parameterized_reference_function is now: 0x7ffcfe537ddc

Main Function
-------------
Value of number within main is now: 1235
Address of number(ptr) within main is now: 0x7ffcfe537ddc
```

Opposed to C++, Java passes parameters exclusively in the Pass-By-Value format. This makes it so that anything modified inside of a function exists only inside of that function, increasing the importance of scope rules and managing objects. If one wanted to modify object values inside of a function, there needs to be a return type of the corresponding object type in the function. Passing an object in java makes a shallow copy inside the function, a new object with the same values, but different spaces in memory. They are effectively the same object, but do not share the same memory locations. When writing in java, it becomes necessary to move away from the c/cpp paradigm of exit code return types and instead embrace Java's throw/catch exception handling for errors, while returning objects or other worthwhile information. If the information that you want to return in Java is multiple points of data of differing types, it then becomes worthwhile to create a new object with corresponding parameters to act as a catching mechanism. No matter whether in Java you pass primitives or objects, it will always be a pass-by-value language. Objects are passed back and forth not as references to the same memory locations, but as separate, new objects containing only the values relevant to that object.