Ryan Hirscher
Dr. Oudshoorn
CSC4510 Programming Language Design
9/17/2023

C++ VS Java

Knowing that C++ is the precursor to Java and that decisions were made to make design changes to address problems, the question is: what is the impact of these changes and are they for the better or for the worse?

Through my research and experiments on both languages it was concluded that C++ supports Pass by Value and Pass by Reference while Java supports strictly Pass by Value. However, due to the nature of these languages, we can implement similar features that mimic the behaviors of the unsupported mechanisms with pointers and objects. For example, Pass by Reference and Pass by Value-Result in C++ can be implemented with '*' and '&' respectively. The following represents the prototypes of the functions in separate programs in C++:

```
void swap (int &, int &);
void swap ( int *, int *);
```

Both of the functions apply value changes to both the formal and actual parameters while keeping the same memory locations throughout the runtime as you can see from the output below:

```
Call by Value-Result to swap function
Pre-swap actual X: 10
X memory address: 0x7ffe2b9e80f0
Pre-swap actual Y: 999
Y memory address: 0x7ffe2b9e80f4

In-swap formal X: 999
X memory address: 0x7ffe2b9e80f0
In-swap formal Y: 10
Y memory address: 0x7ffe2b9e80f4

Post-swap actual X: 999
X memory address: 0x7ffe2b9e80f0
Post-swap actual Y: 10
Y memory address: 0x7ffe2b9e80f4
```

```
Pass by reference to swap function
Pre-swap actual X: 10
X memory address: 0x7ffd8201b770
Pre-swap actual Y: 999
Y memory address: 0x7ffd8201b774

In-swap formal X: 999
X memory address: 0x7ffd8201b770
In-swap formal Y: 10
Y memory address: 0x7ffd8201b774

Post-swap actual X: 999
X memory address: 0x7ffd8201b770
Post-swap actual Y: 10
Y memory address: 0x7ffd8201b774
```

As stated earlier, Java only supports Pass by Value. This is because Java does not allow the implicit declaration of pointers. They remove this capability because pointers, if misused, can lead to critical failures that cannot be easily recognized by a compiler and may cause serious issues during runtime. However, pass by reference has not been completely removed and can still be achieved through objects because objects are referenced under the hood with pointers. Pointers are the intrinsic design feature to reference objects in object oriented programming. Below is a declaration statement to create a new instance of type CallByRef named obj which stores 5 and 12 into integers called X and Y:

```
CallByRef obj = new CallByRef(5, 12);
```

Then, we call the method to add 5 to both X and Y by passing the object.

```
obj.addFive(obj);
```

The following output demonstrates that the addition of X and Y of obj was retained in memory outside the scope of the function:

```
Pass by Reference (class object) to addFive function.
Actuals Pre-add: X=5 Y=12
Formals: X=10 Y=17
Actuals Post-add: X=10 Y=17
```

The other techniques like Call by Name, Call by Result, and Call by Value-result are not supported. These mechanisms tend to be the older parameter passing techniques that were used in programming languages like Ada or Algo. In high-level object oriented programming, these mechanisms are considered obsolete and unsupported. In my opinion, I think it should stay that way. I cannot think of any reason that these techniques would be crucial to an application where you are unable to emulate the same features with objects or pointers while being precise. Each of these behaviors can be implemented in similar fashions. Call by Result can be replaced by passing a referenced dummy variable (by object or pointer) with no value and then writing to it at the return. Call by Name works by default for both of the referencing techniques natively. Within the function, the formals can be aliased in many ways to reference the actuals if they are pointers or objects. For Call by Value-Result, it is very similar to pass by reference. In C++ it is also called Call by Reference, when passing '&' it performs like Call by Value-Result and then when passing '*' it performs very similarly to the expected Call by Reference. In Java, there is no such passing mechanism with special characters, there are only objects. Java has succeeded in reducing the number of passing parameter techniques as a whole, but has complicated the process to mimic the behaviors of the individual semantics.

I cannot argue in support of either C++ or Java. Both of them call for different situations and different requirements. This is why we have not come to a consensus on a single programming language. Pointers can be very useful in an application as it gives more functionality and direct access to memory so long as it is not misused by the programmer where there can be critical consequences. Java removes the ability to directly interact with memory and allows the user to implicitly define pointers which limits the users' abilities, but also minimizes the risk of critical failures. This is the tradeoff between the two programming languages. One of the main reasons that Java was created was to offer an even higher-level object oriented programming language that is similar to C++, but also has guardrails to prevent poor memory management, pollution, and leakage. The responsibility to decide on the appropriate language falls on the programmer. If the programmer plans and writes code carelessly then the product is going to be subpar or difficult to troubleshoot. But if the programmer designs a product around knowing how the language operates, then there should not be an issue.