**Introducción:**

En este trabajo se propone realizar un análisis comparativo entre los aspectos de programación funcional (FP, por las siglas en inglés) de C++ y de Python. Ambos lenguajes poseen características funcionales, como la mayoría de los lenguajes de programación actuales, pero ¿cuál de los dos es el indicado para la resolución de problemas de manera funcional? ¿Depende del problema o alguno de los dos tiene características definitivamente superiores? Para responder esto, se aplicarán distintas técnicas de FP en uno y otro lenguaje, y se evaluarán puntos como: desempeño; facilidad de escritura, lectura y chequeo de errores; potencial de uso. Luego de cada aspecto, se hará una conclusión parcial de los resultados obtenidos en el experimento. Finalmente, se expresará una conclusión a modo de cierre, en la cual se intentará responder a las preguntas planteadas previamente.

Nota: A lo largo del trabajo, se intentará hacer referencia principalmente a las funciones nativas de cada lenguaje, sin tener en cuenta librerías que puedan suplir las deficiencias del lenguaje. Esto aplica principalmente a C++, que tiene librerías como *Boost* que abarcan un gran número de funciones y algoritmos para trabajar con C++ de forma “pythónica”.

**Conceptos previos:**

Para empezar a responder las preguntas planteadas en la introducción, es necesario primero tener en claro algunos conceptos relacionados a la programación funcional.

En primer lugar, ¿qué es FP? Según la Universidad de Nottingham, FP:

*La programación funcional es un estilo de programación que pone énfasis en la evaluación de expresiones, en lugar de la ejecución de comandos. Las expresiones en estos lenguajes son formadas usando funciones para combinar valores básicos.*[[1]](#footnote-1)

¿Qué quiere decir esto? Esta expresión quiere decir que, a diferencia de la programación imperativa tradicional de C++, se intenta describir a los problemas en términos de funciones, más que en términos de órdenes. Al paradigma FP se lo suele considerar como un paradigma declarativo, en el cual no se le explica a la computadora qué pasos debe seguir para resolver un problema, sino que se definen los parámetros del problema. Así, los programas se estructuran de forma que los pasos que se siguen son evaluaciones de funciones, para las cuales el procesador tiene “una explicación” dada por el programador. Veamos un ejemplo:

Imagen que contiene pantalla, celular, monitor, teléfono

Descripción generada automáticamente

Figura 1: Aplicación en C++ de la función factorial de forma imperativa y recursiva.

En la Figura 1 se ven dos formas alternativas de escribir una función en C++ que reciba un número entero positivo y devuelva su factorial (producto de todos los números enteros entre 1 y sí mismo, inclusive).

Vemos que, en el primer caso, la función se llama a sí misma. No se le dice expresamente a la función cómo se calcula el factorial de un número, sino que simplemente se le dice que evalúe los valores y siga el *step* hasta que, si se lo programó correctamente, el programa llegará al caso base y se efectuarán los sucesivos *return* hasta salir de la función. Es un estilo de programación claro, conciso y declarativo. Sin embargo, trae consigo un agregado de *overhead*, ya que se deben ir apilando las llamadas a funciones.

En el segundo caso, la función se escribe de forma imperativa, con un *loop* explícito en el cual se van multiplicando los números sucesivos y luego sale, cuando la variable *i* deja de cumplir la condición especificada. Como se puede ver, se debe agregar una variable *tot* que mantenga el valor actual de la cuenta, lo cual no es necesario en la versión FP.

Nota: se dejó de lado el chequeo de errores en el input del número a evaluar, con el fin de simplificar el código.

Retomando el tema de FP, en general, los programas que se escriben suelen seguir ciertas reglas generales:

* Utilizar principalmente funciones puras, que son funciones que sólo pueden usar (pero no modificar) los argumentos que se les pasan con el fin de calcular el resultado.[[2]](#footnote-2) Esto quiere decir que una función llamada con los mismos argumentos debería devolver siempre el mismo valor, independientemente del estado del programa que la está llamando.
* Mantener el valor de las variables: también conocido como inmutabilidad. Las funciones no deben modificar variables externas a ella, o que “no le pertenezcan”. Estrictamente, una función no debería tener mutabilidad incluso dentro de sí misma.

Siguiendo en principio estas reglas, se podría comenzar a escribir con el paradigma de FP código cada vez más corto, fácil de leer, manejable y con un mínimo de puntos propensos a errores. Al definir los problemas en términos de sí mismo, o en términos de evaluaciones de funciones que no modificarán su ambiente externo, la posibilidad de errores se minimiza y la detección de los errores que puedan ocurrir es mucho más veloz.

Por último, otro concepto clave que se necesitará conocer es el de funciones de orden superior (HOF, por las siglas en inglés). Éstas son funciones que reciben o devuelven otras funciones. Un ejemplo:

Imagen que contiene negro, sostener, blanco, rojo

Descripción generada automáticamente

Figura 2: Función en C++ que recibe como parámetro una función.

En la Figura 2 se puede observar un *template* de función que recibe un iterable de tipo *T* y una función de tipo *F*, y aplica la función a cada uno de los valores del iterable. Dada la versatilidad de los *templates* y la practicidad de FP, esta combinación resulta muy efectiva a la hora de resolver problemas que necesiten aplicar HOF. Como se observa, el código es extremadamente conciso y fácil de entender.

Entendidos estos conceptos, el siguiente paso será comparar las aplicaciones de FP en los dos lenguajes previamente mencionados: C++ y Python.

**Análisis:**

**Funciones lambda:**

Las funciones lambda son funciones anónimas que se escriben en la parte del código donde se desean usar (Čukić,2018). A fines del trabajo actual, eso es lo más importante que se necesita saber. Están basadas en el cálculo lambda, una piedra angular de la programación funcional. Siguiendo las características de este paradigma de programación, las funciones lambda facilitan la lectura y escritura del código, permitiendo crear funciones de corta duración. Como explica Čukić en su libro, no tendría sentido ir a una parte externa al código y nombrar funciones que serán usadas sólo una vez. Para eso aparecen las funciones lambda.

Nota: en este trabajo se hará uso indistinto de “funciones lambda” y “expresiones lambda”, aunque técnicamente las soportadas por Python y C++ son las últimas.

**C++:**

La sintaxis en C++ consiste en, principalmente, los siguientes aspectos:

* [ ]: indicador de captura. Dice cuáles y de qué forma (valor o referencia) serán utilizadas las variables del entorno.
* ( ): indicador de parámetros. Los parámetros que recibe la función lambda. Sintaxis igual a la de funciones tradicionales.
* { }: indicador de cuerpo. Las acciones que lleva a cabo la función.

Nota: Hay más puntos que puede tener una función lambda, como la mutabilidad de las variables pasadas por valor, un aviso sobre los *throw* que puedan aparecer dentro de la expresión y una explicitación del tipo de dato de retorno en caso de haber múltiples *return*, pero para el análisis actual no son tan relevantes.

Veamos un ejemplo:

Imagen que contiene teléfono, celular, laptop, tabla

Descripción generada automáticamente

Figura 3: Ejemplo de expresiones lambda en C++.

En la Figura 3 se utilizó el *template* apply escrito en la Figura 2, pero aplicado de dos formas distintas a dos vectores distintos. En una primera instancia, se lo aplica con una función tradicional, definida afuera del main. En el segundo caso, se aplicó una expresión lambda, escrita en el momento.

¿Qué ventajas trajo este modo de escritura?

Para empezar, código más legible. Se puede mirar a la función y ver exactamente lo que está haciendo, escrito en el momento.

Por otro lado, código más manejable. Si se quiere modificar lo aplicado en esa llamada a *apply*, basta con ir a dicho punto y modificarlo. De la otra forma, habría que ir afuera del main y modificar la función *add1*, pero teniendo en cuenta a los puntos que potencialmente la puedan haber llamado desde afuera. Alternativamente, se podría definir otra función *f* que pusiera en práctica la nueva funcionalidad requerida, pero en ambos casos la alternativa lambda suena mucho más concisa y al punto.

Sin embargo, es importante notar que ambas formas de escritura son equivalentes. Los distintos paradigmas de programación logran convivir en C++, permitiendo aplicar uno u otro de manera que se considere conveniente en cada caso y para la aplicación específica.

**Python:**

Veamos un ejemplo de expresiones lambda en Python:

Imagen que contiene tabla, pantalla, negro, cuarto

Descripción generada automáticamente

Figura 4: Ejemplo de expresiones lambda en Python.

Nota: la función *map* es equivalente a la función *apply* de la Figura 2, donde se le brinda una función y un iterable. Devuelve un iterable del tipo *map*, construido a partir de la aplicación de la función a cada uno de los miembros del iterable.

Como se observa, la sintaxis es incluso más concisa que en C++. Consta simplemente de la palabra ‘lambda’ seguida por la lista de parámetros, dos puntos y el *return*. Las funciones lambda de Python también permiten aplicar discriminaciones *if/else* en su cuerpo, variando el *return* (o no) en cada caso.

Nuevamente, las ventajas son fácilmente apreciables entre las dos formas de escribir el mismo programa. Mayor facilidad de lectura, mayor versatilidad y menos posibilidad de errores.

**Comparación:**

Se vio que tanto C++ como Python tienen expresiones lambda, cada lenguaje con su sintaxis propia. Pero ¿qué diferencias hay entre ellas? Por un lado, veamos el siguiente ejemplo en Python:

Captura de pantalla de computadora

Descripción generada automáticamente

Figura 5: Ejemplo de valores por referencia en lambda en Python.

Como se observa en la Figura 5, los valores pasados a las funciones lambda de Python son siempre por referencia. Esto, en muchas ocasiones, puede resultar beneficioso: no hay que llamar a los constructores innecesariamente, por ejemplo. Sin embargo, en otras ocasiones puede resultar perjudicial, como se ve en el ejemplo. Uno esperaría que cada una de las funciones lambda guardadas en *v\_lambdas* o *v\_lambdas\_2* devolvieran el argumento más el valor de la posición. Pero como *i* fue pasado por referencia, todas hacen referencia al mismo *i*, por lo que cualquier cambio en el valor de éste afecta a todas por igual. Es por esto por lo que todas terminan devolviendo, en primera instancia, el argumento más el último valor que tuvo *i* al salir del loop. Por otro lado, se ve que al configurar *i = 0,* todas pasan a devolver simplemente el argumento.

Imagen que contiene captura de pantalla, tabla, laptop

Descripción generada automáticamenteEn cambio, en C++:

Figura 6: Ejemplo de valores por valor y referencia en lambda en C++.

Se ve en la Figura 6 que, haciendo uso de las clausuras ([ ]) se pueden configurar distintas formas de capturar los valores del entorno, haciendo que la función lambda sea mucho más versátil. En un primer caso, se toma *i* por valor, por lo que todas las expresiones lambda devuelven el argumento más la posición. El segundo ejemplo es equivalente a lo que en Python ocurre por defecto, que *i* está pasada por referencia, por lo que todas las funciones devuelven lo mismo. Comentado hay también un tercer caso, en el que no se especifica captura de ninguna variable del entorno, por lo que la expresión escrita no compilará, al no haber sido definida *i*.

Por otro lado, las funciones lambda de Python son “más funcionales”, es decir, adhieren más al paradigma funcional declarativo. Las funciones lambda deben ser una declaración simple, especificando sus parámetros y sus valores de retorno. No se pueden poner múltiples acciones dentro de una expresión lambda (como hacer un *print* y luego devolver un valor). Para ello, se deben usar las funciones tradicionales, definidas con *def*.

Captura de pantalla con letras y números

Descripción generada automáticamenteSin embargo, en C++ se puede hacer algo así:

Figura 7: Ejemplo de programación imperativa en lambda en C++.

Vemos en la Figura 7 cómo las expresiones lambda de C++ permiten realizar cualquier acción que se podría realizar dentro de una función normal, con la clara ventaja de que los parámetros pueden ser del tipo *auto*, es decir, las funciones lambda se convierten en un *template*.

Por último, cabe destacar que no sólo las funciones lambda, sino todas las funciones en Python actúan como *template*, en el sentido de que no es necesario especificar el tipo de dato, por lo que una misma función podría recibir como argumento distintos tipos de dato y funcionar correctamente. Ejemplo:

Imagen que contiene pantalla, monitor, teléfono, celular

Descripción generada automáticamente

Figura 8: Ejemplo de lambda aplicada a varios tipos de dato en Python.

Se observa claramente en la Figura 8 cómo una misma función lambda funciona correctamente para enteros, listas, diccionarios, rangos, *strings, floats* y hasta para otra función lambda. Esta versatilidad es de gran utilidad en la programación funcional, porque permite reutilizar aún más las mismas funciones y reducir aún más la proporción de errores que se puedan llegar a cometer.

Nota: No se ahondará en claridad y legibilidad del código, tiempo de ejecución ni ocupación de memoria, porque son características propias de cada lenguaje y no tanto de la forma que tenga el lenguaje de expresar las funciones lambda.

**Comprensión:**

Una herramienta muy usada en FP es la comprensión. En este trabajo, se ejemplificará esencialmente con listas, pero se puede extender a la mayoría de las colecciones. La comprensión es un método por el cual se crea una colección (a partir de ahora, repito, se hablará exclusivamente de listas) a partir de otra, en una sola línea, de manera concisa.

¿Qué ventajas trae esto? Como la gran mayoría de lo relacionado FP, trae consigo claridad en el código, reducción de la probabilidad de errores y una escalabilidad y maleabilidad mucho mayores.

**C++:**

La comprensión no es una característica nativa de C++. Si bien hay librerías *open source* que intentan recrearla, principalmente a partir del uso de la librería *Boost::Phoenix[[3]](#footnote-3) o range\_v3* (que luego dio lugar a C++20 *Ranges*)[[4]](#footnote-4) y algoritmos como *transform* o *for\_each*, no se tocará el tema por irse del alcance de la monografía. Pero resulta relevante mencionar al menos su existencia, la cual no se está pasando por alto.

**Python:**

En Python, por otro lado, la comprensión es moneda de uso corriente. Al ser un lenguaje orientado principalmente a *Data Science*, Python está preparado para trabajar con grandes cantidades de datos. En ese proceso, generalmente se busca convertir esos datos de un formato a otro, generalmente aplicándoles en medio alguna función, y guardarlo rápidamente en otra lista. Para evitar crear funciones extra, listas vacías e iteraciones innecesarias, se utiliza la comprensión de listas. Ejemplo de ambas formas:

Imagen que contiene monitor, tabla, teléfono, pantalla

Descripción generada automáticamente

Figura 9: Comprensión de listas en Python.

En la Figura 9 se observa la facilidad de escritura de la comprensión de listas en Python, tanto así que el código en sí suele ser más explicativo que los comentarios al respecto. En una línea se pudo escribir lo que de otra forma se habría tenido que escribir en tres, y además se lo hizo de forma mucho más expresiva. Cabe destacar que las comprensiones de listas tienen la misma sintaxis que una función lambda, por lo que podrían aplicarle lógicas condicionales dentro de la comprensión.

**Comparación:**

En este caso, no se puede llevar a cabo una comparación, porque en Python las comprensiones son tan poderosas como fáciles de usar, mientras que en C++ implican inclusión de librerías y contenido que todavía está en vías de desarrollo. En este sentido, Python tiene superioridad en cuanto al *approach* FP, permitiendo usar esta herramienta que es extremadamente común en la resolución de problemas de manera funcional.

**Functores:**

Los functores son el punto en el que FP se encuentra con la programación orientada a objetos (OOP, por sus siglas en inglés). C++ es un lenguaje principalmente del estilo OOP, por lo cual ese aspecto está ampliamente desarrollado. Python, por su parte, también tiene un manejo de objetos que es comparable con el nivel de C++, con muchas similitudes y algunas marcadas diferencias. Los functores son simplemente clases que tienen sobrecargado el operador ( ). Esto permite que estos objetos sean “como funciones”, en el sentido de que pueden ser llamados con un valor, y se ejecutará una serie de pasos (o evaluaciones en funciones, si se plantea de manera funcional), para luego devolver un valor final (o no, dependiendo del caso).

¿Qué ventajas tienen los functores? Dado que las implementaciones tanto en Python como en C++ son similares, veamos su uso con un ejemplo.[[5]](#footnote-5)

**C++:**

Problema: Partiendo de la función *apply* de la Figura 2, si se quisiera hacer una función que sume uno, una forma de hacerlo sería la siguiente:

Imagen que contiene pantalla, televisión

Descripción generada automáticamente

Figura 10: Planteo del problema que solucionan los functores en C++.

Sin embargo, si luego se quisiera incrementar los valores en cinco, se debería hacer lo siguiente:

Imagen que contiene pantalla, monitor, televisión, sostener

Descripción generada automáticamente

Figura 10.1: Parte 2 del problema que solucionan los functores en C++.

Imagen que contiene captura de pantalla, monitor, pantalla, televisión

Descripción generada automáticamenteEsto puede resultar tedioso, ya que justamente el paradigma de FP trata sobre la reducción de excesos y la no implementación de funciones innecesarias. Con functores, la solución sería así:

Figura 11: Solución del problema con functores en C++.

Así, se soluciona el problema, lográndolo generalizar a través de objetos que sirven como funciones. Además, esto posee otra ventaja: los objetos pueden guardar más información que sólo el constructor y la sobrecarga del operador ( ). Potencialmente, esta propiedad tiene múltiples usos, ya que se pueden mantener registros de los procedimientos que se van haciendo, o se pueden realizar procedimientos en paralelo con los datos obtenidos.

**Python:**

Equivalentemente, la situación en Python sería la siguiente:

Problema:

Imagen que contiene interior, foto, pantalla, monitor

Descripción generada automáticamente

Figura 12: Planteo del problema que solucionan los functores en Python.

Solución:

Imagen que contiene tabla, pantalla, monitor, teléfono

Descripción generada automáticamente

Figura 13: Solución del problema con functores en Python.

**Comparación:**

Como se mencionó antes, vemos claramente que la solución del problema es muy similar en ambos lenguajes, limitándose a la creación de una clase *Adder* con un constructor que recibe un número y lo guarda en una variable propia, y la sobrecarga del operador ( ), con la cual devuelve la suma entre el número pasado como argumento y su propio número.

En cuanto a ventajas de Python, nuevamente vuelve a aparecer la generalidad que se obtiene al no tener que especificar tipos de datos. En C++, para implementar el mismo caso con *float*, se debería sobrecargar nuevamente el operador ( ), o escribir un *template* para la clase *Adder*, donde se haga de manera más genérica. Ambas soluciones son más tediosas que en Python, donde nativamente no se especifica tipo de dato. Aunque, cabe destacar, en el caso particular de estudio, el operador suma que se aplica en la sobrecarga al operador ( ) ya es en parte restrictivo, dado que su correcto funcionamiento dependerá de con qué tipo de dato se instanció la clase.

Sin embargo, una clara desventaja de Python radica en su manejo de los métodos y datos miembro de las clases. En las clases de Python, y en particular en los functores, no existen restricciones sobre quién puede acceder a los datos miembro y a sus métodos, y más aún, cualquiera puede crear nuevos datos miembro y nuevos métodos desde afuera de la clase. Si bien esto puede resultar ventajoso en muchas ocasiones, en este caso resulta más que nada un impedimento para el funcionamiento “hermético”, puro, que se busca en la programación funcional. Si se espera que cada vez que se llame a una función, el resultado sea el mismo, entonces es lógico pensar que no se deberían poder modificar los pasos que sigue dicha función.

Nota: Si bien existen formas de hacer que en apariencia haya métodos o datos miembro que sean inaccesibles desde afuera de una clase, estas formas son fácilmente *bypasseables*, es decir, existen maneras de salteárselas.[[6]](#footnote-6) Por este motivo, a fines de esta monografía, se considera que no existen tales maneras.

Ejemplos:

Captura de pantalla con letras blancas

Descripción generada automáticamente

Figura 14: Sobreescritura de método *\_\_call\_\_* en Python.

Se puede ver en la Figura 14 un ejemplo de un problema leve que podría ocasionar la falta de seguridad en las clases de Python. En este caso, la clase *Adder* que se había definido, pasó en una sola línea a tener una funcionalidad opuesta a la que tenía originalmente. Problemas de este estilo son los que C++ intenta evitar con las declaraciones *private, public, protected* y sus variaciones.

Otro ejemplos:Captura de pantalla con letras y números

Descripción generada automáticamente

Figura 14.1: Otra sobreescritura de método *\_\_call\_\_* en Python.

Imagen que contiene monitor, pantalla, televisión, negro

Descripción generada automáticamente

Figura 14.2: Sobreescritura de método *\_\_init\_\_* en Python.

En los casos de la Figura 14.1 y 14.2, los problemas que puede ocasionar esta falta de seguridad son mucho más severos que en la figura 14. En estos casos, el programa directamente se romperá.

En la figura 14.1, el programa se romperá porque se está intentando llamar a la sobrecarga *\_\_call\_\_* con dos parámetros (*self* y el número), pero la sobreescritura hizo que ahora *\_\_call\_\_* tome sólo un parámetro. Por lo tanto, el programa correrá con errores.

En la figura 14.2, el programa se romperá porque se sobreescribió el método *\_\_init\_\_*, haciendo que nunca se cree la variable *self*.*num* (o *add3.num*, en el caso particular). Por lo tanto, al querer utilizarla en el método *\_\_call\_\_*, habrá un error.

1. Graham Hutton, “Frequently Asked Questions for comp.lang.functional”, University of Nottingham, Nottingham, Inglaterra, 2002. [↑](#footnote-ref-1)
2. Ivan Čukić, “Functional Programming in C++”, Manning Publications, 2018. [↑](#footnote-ref-2)
3. Joel de Guzmán, Dan Marsden, Thomas Heller, John Fletcher, “Phoenix 3.2.0”, Boost Software, 2015. [↑](#footnote-ref-3)
4. Autor desconocido, Ranges Library, CPP Reference, <https://en.cppreference.com/w/cpp/ranges>, 2020. [↑](#footnote-ref-4)
5. Ejemplo tomado de Autor desconocido, Functors in C++, GeeksforGeeks, <https://www.geeksforgeeks.org/functors-in-cpp/>, sin fecha especificada. [↑](#footnote-ref-5)
6. nikhilaggarwal3, Private Methods in Python, GeeksforGeeks, <https://www.geeksforgeeks.org/private-methods-in-python/>, sin fecha especificada. [↑](#footnote-ref-6)