La importancia de la enseñanza de C++ en la carrera de Ingeniería en Computación en Argentina

Ing. Gabriel Valenzuela, Ing. Luis Orlando Ventre, Dr. Ing. Orlando Micolini, and Ing. Mauricio Ludemann

Laboratorio de Arquitectura de Computadoras, FCEFyN-Universidad Nacional de Córdoba Av. Velez Sarfield 1601, CP-5000, Córdoba, Argentina
{gabriel.valenzuela, luis.ventre, orlando.micolini, mauri.ludemann}@unc.edu.ar

Resumen En las carreras de Ingeniería en Computación, la elección del lenguaje de programación inicial es un debate vigente. Este trabajo defiende la tesis de que C++ constituye un lenguaje clave en la formación del ingeniero en computación en Argentina, debido a que ofrece una comprensión integral desde el *top level* hasta el *bottom level* y tiene aplicaciones que abarcan desde sistemas embebidos hasta sistemas complejos de alta performance. Se analiza el contexto educativo argentino, comparando C++ con otros lenguajes populares (C, Java, Python) a partir de literatura académica reciente, incluyendo las novedades de estándar 20 ,**std20**, en programación concurrente. Usando el modelo de Toulmin para estructurar el argumento, se aportan evidencias inductivas locales (ejemplos de la UBA, que si bien no es propiamente Ingeniería en Computación se destaca su forma de enseñanza, UTN, UNLP y otras instituciones argentinas) y deducciones generales sobre pedagogía en computación. Los resultados sugieren que la enseñanza temprana de C++ fortalece al futuro ingeniero incluso si luego trabaja con otros lenguajes, impactando positivamente materias avanzadas como Sistemas Operativos, Sistemas Embebidos, Sistemas Ciberfísicos y Programación Concurrente.

Keywords: Educación en ICOMP, C++, Lenguajes de Programación, Programación Concurrente, Argentina

1. Introducción

La formación de ingenieros en computación enfrenta el desafío de elegir el lenguaje de programación adecuado para iniciar a los estudiantes. En años recientes, se observa una tendencia global a adoptar lenguajes de *alto nivel* como Python en los cursos introductorios, debido a su sintaxis simple y curva de aprendizaje más amigable. De hecho, encuestas internacionales ubican a Python como el lenguaje más popular en entornos educativos, seguido de Java y C, mientras que C++ se mantiene firmemente dentro de los cinco principales [1, pp. 86–91] (ver también ranking IEEE Spectrum 2022 [6]). En Argentina, muchas carreras

de Ingeniería en Computación y afines, han mantenido un énfasis tradicional en lenguajes de nivel bajo/medio como C y C++ en sus primeros años de estudio. Por ejemplo, en la Universidad Tecnológica Nacional (UTN) – Facultad Regional Buenos Aires, la materia "Algoritmos y Estructuras de Datos" utiliza C++ a lo largo de todo el curso (en algunos planes se emplea C básico) [7]. Este contexto educativo local contrasta con la creciente adopción de Python en otras instituciones, incluso dentro del país: por ejemplo, UTN Córdoba introdujo Python en cursos iniciales [8]. Esto plantea el debate sobre cuál enfoque produce mejores resultados en la formación del ingeniero. La tesis que se defiende en este trabajo es que el uso de C++ representa un lenguaje *clave* en la educación del ingeniero en computación. C++ permite una comprensión integral desde el "top level" (abstracciones de alto nivel) hasta el "bottom level" (cercano al lenguaje ensamblador), y sus aplicaciones abarcan un espectro amplio, desde sistemas embebidos de bajo nivel hasta sistemas de alta performance como el *high-frequency trading* (trading algorítmico de alta frecuencia). En soporte de esta tesis, se empleará el modelo argumentativo de Toulmin, proporcionando la *afirmación central*, *fundamentos empíricos*, *garantía lógica que conecta evidencias con la tesis*, *respaldo adicional*, *calificadores que delimitan la afirmación* y *refutaciones potenciales* para brindar un análisis completo. A continuación, se presenta primero el contexto educativo actual y luego una comparación de C++ con otros lenguajes dominantes (C, Java, Python), incluyendo un análisis de las mejoras introducidas en **C++20** para la enseñanza de programación concurrente. Finalmente, se discuten las implicancias futuras en el desarrollo profesional de los ingenieros en computación, utilicen o no C++ en su ámbito laboral.

2. Desarrollo

2.1. Contexto educativo y fundamentos para la enseñanza de C++

Una de las razones fundamentales para enseñar C++ en carreras de computación es su dualidad pedagógica: es un lenguaje **multiparadigma** que expone al estudiante a detalles de bajo nivel (manejo manual de memoria, punteros, representación de datos en memoria) a la vez que ofrece abstracciones de alto nivel (programación orientada a objetos, programación funcional, metaprogramación). Este rango de alcance facilita cumplir el objetivo formativo de muchos planes de estudio de ingeniería en computación, que buscan que el egresado comprenda tanto el hardware como el software a profundidad. C++ actúa como un puente entre ambos extremos: por un lado, permite implementar estructuras de datos y algoritmos eficientes por el control que ofrece sobre el hardware; por otro lado, fomenta buenas prácticas de diseño modular y reutilización de código propias de lenguajes de alto nivel. Podemos identificar la *garantía lógica* de nuestra tesis en la idea de que aprender a programar con una herramienta que ofrece control total sobre el hardware resulta en una comprensión más sólida de cómo funcionan los sistemas de computación. La conexión entre teoría y práctica se refuerza cuando, por ejemplo, un estudiante implementa una estructura de datos en C++ y puede analizar su comportamiento en memoria, e incluso inspeccionar

el ensamblador generado por el compilador. Este respaldo se alinea con estándares académicos: la carrera de Ingeniería en Computación en Argentina, según las pautas de acreditación nacionales, exige conocimientos tanto de alto nivel (algoritmos, paradigmas) como de bajo nivel (arquitectura de computadoras, sistemas operativos) [9]. C++ contribuye en ambos aspectos formativos. Por supuesto, existen *calificadores que delimitan la afirmación* a esta afirmación: n**o se sostiene que C++ sea el único lenguaje valioso ni que deba ser el único enseñado**. Más bien, se argumenta que es **central** o esencial en la formación porque aporta algo que los lenguajes más abstractos no logran por sí solos. Un graduado con dominio de C++ podrá aprender con relativa facilidad lenguajes como Java o Python posteriormente (de hecho, se ha observado anecdóticamente que la transición de C++ a Python/Java es más sencilla que el camino inverso) [**?**]. El proceso inverso (pasar de solo saber Python a entender C++ y conceptos de memoria) suele resultar más desafiante y requerir re-educar fundamentos.

2.2. Comparación de C++ con C: Abstracción vs. simplicidad de bajo nivel

El lenguaje C es un antecesor directo de C++ y goza de amplio uso en áreas de sistemas embebidos, firmware y desarrollo de sistemas operativos. En muchas universidades, C se enseña primero debido a su menor complejidad sintáctica y semántica en comparación con C++. C carece de características de alto nivel como clases u objetos, lo cual paradójicamente puede ser una ventaja pedagógica inicial al centrar la enseñanza en conceptos básicos de control de flujo, tipos de datos primitivos y organización de memoria sin otras distracciones. De hecho, algunos educadores han considerado a C más apropiado que C++ para principiantes absolutos por su relativa simplicidad en el modelo de ejecución. Sin embargo, estudios contemporáneos empiezan a cuestionar esta suposición tradicional. Por ejemplo, un trabajo de 2021 reporta que, al implementar algoritmos introductorios, estudiantes que aprendieron con C++ no mostraron mayor dificultad que aquellos que usaron lenguajes supuestamente "más simples" [3]. Este estudio, apoyado en métricas de software (Halstead), halló que Java resultó incluso más difícil que C++ para novatos en ciertos problemas, requiriendo más esfuerzo, tiempo y produciendo más errores en promedio [3]. La afirmación central aquí, es que C++ ofrece lo mejor de ambos mundos: retiene el rendimiento y control de C (esencial en los sistemas ciberfísicos e industria 5.0) añadiendo mecanismos que permiten manejar la complejidad de grandes proyectos. Un *fundamento empírico* que ilustra esto es la prevalencia de C++ en la industria de software de sistemas críticos. C++ es utilizado en desarrollo de *drivers*, motores de bases de datos, sistemas de telecomunicaciones, etc., donde antes se usaba únicamente C, precisamente porque ofrece abstracciones (ej. diferentes paradigmas de programación) *sin sacrificar eficiencia*. Este no es un detalle menor, ya que, las abstracciones en C++ suelen tener costo-cero en tiempo de ejecución, es decir, el programador puede escribir código a alto nivel y confiar en que el compilador lo traducirá a ejecutables tan eficientes como si hubieran sido escritos en C o ensamblador manualmente. En sistemas embebidos modernos comienza a

verse una transición de C a C++, aprovechando características como RAII (inicialización y liberación automática de recursos) que reducen errores de gestión de memoria manteniendo un uso mínimo de recursos. Reportes industriales señalan que C++ es ampliamente usado en IoT y desarrollo embebido precisamente por estas ventajas, proporcionando fiabilidad y flexibilidad junto con acceso al manejo de memoria de bajo nivel cuando es necesario [12].

La *garantía lógica* que liga estas evidencias con la importancia de C++ en educación sería: si un lenguaje supuestamente más sencillo no garantiza mejores resultados académicos, y además es menos pertinente para desarrollar software de alto rendimiento, entonces no hay justificación sólida para desplazar a C++ del plan de estudios en favor de ese otro lenguaje. En cambio, incluir C++ empodera al estudiante para luego aprender otros lenguajes con mayor facilidad (por ejemplo, la sintaxis de Java deriva en gran parte de C++), a la vez que comprende qué sucede en el hardware. Por ejemplo, un estudiante que ha lidiado con punteros y gestión manual de memoria en C++ tendrá mucho más claro qué hace el *garbage collector* de Java y por qué ciertas estructuras de datos tienen ciertos costos.

2.3. Comparación de C++ con Java: Manejando la complejidad y el rendimiento

Java ha sido durante décadas un lenguaje prominente tanto en la enseñanza de la programación como en el desarrollo de software empresarial. Muchas universidades migraron sus cursos introductorios a Java a inicios de los 2000, incentivadas por la adopción de Java en exámenes estandarizados (ej. el AP Computer Science) y por la promesa de un lenguaje orientado a objetos puro, con gestión automática de memoria (garbage collector) y una rica biblioteca estándar. En Argentina, varias facultades incorporaron Java en cursos de segundo año (como Programación Orientada a Objetos), usualmente después de una base en C o C++. Por ejemplo, en la UNC se ve C o C++ en primer año, y más adelante Java en materias de POO y concurrente. Al comparar Java con C++, es importante analizar ventajas y desventajas concretas basadas en estudios. Del lado de Java, su sintaxis es más sencilla (no hay aritmética de punteros ni necesidad de manejo manual de memoria) y su modelo de objetos unificado puede ser más fácil de asimilar para ciertos estudiantes. No obstante, Java sacrifica la cercanía al hardware y la eficiencia en tiempo de ejecución. Un programa Java corre sobre una máquina virtual (JVM), lo que **introduce una capa de abstracción adicional**; además, la gestión de la memoria por parte del garbage collector, si bien libera al programador de gestionarla, puede provocar pausas no deterministas en la ejecución (lo cual es problemático en sistemas de tiempo real). La evidencia empírica reciente arroja resultados interesantes: el estudio de 2021 antes mencionado que midió la implementación de algoritmos clásicos en C++ vs Java encontró que C++ resultó en menos *bugs* entregados y tiempos de desarrollo menores en promedio que Java para esas tareas [3]. En otros términos, a pesar de la percepción común de que "Java es más fácil", los datos sugieren que

para ciertos problemas básicos los estudiantes consiguieron soluciones más rápidamente y con menos errores en C++ que en Java. Los autores concluyen que "C++ es más adecuado que Java" para implementar algoritmos introductorios en cursos iniciales de programación [3].

Este resultado sirve de *fundamento empírico* para nuestro argumento: indica que la supuesta simplicidad de Java no siempre se traduce en mejores resultados de aprendizaje o de productividad en la etapa formativa. Desde el punto de vista de desempeño, históricamente Java ha sido más lento que C++ en ejecución cruda, aunque la brecha se ha acortado gracias a optimizaciones *just-in-time* de la JVM. Aún así, en aplicaciones donde cada ciclo cuenta (simulaciones numéricas intensivas, motores de juegos, cálculos científicos), C++ mantiene una ventaja. Un reporte señala que si bien lenguajes como Python dominan el panorama general, el conjunto de lenguajes *estilo C* (C, C++, C#) continúa siendo sumamente relevante y combinado supera en popularidad a cualquier lenguaje individual [6]. Esto refleja también la preferencia industrial: para una gran categoría de problemas de ingeniería, la familia de C,C++ es la opción natural por rendimiento y control. Una *garantia* que conecta estas evidencias con la importancia de C++ en educación sería: si un lenguaje "más sencillo" no garantiza mejores resultados académicos, y además es menos adecuado para software de alto desempeño, entonces no existe motivo pedagógico firme para reemplazar C++ por Java en la currícula básica. Incluir C++ en la formación, en cambio, empodera al estudiante para luego aprender Java con facilidad (dado que gran parte de la sintaxis y conceptos de Java derivan de C++), a la vez que le permite entender qué ocurre tras bambalinas. Por ejemplo, un estudiante que haya lidiado con punteros en C++ comprenderá mejor qué hace el recolector de basura de Java y por qué ciertas estructuras en Java tienen el costo que tienen.

2.4. Comparación de C++ con Python: facilidad inicial vs. profundidad conceptual

Python se ha posicionado como el lenguaje de moda tanto en la industria (particularmente en ciencia de datos, inteligencia artificial, automatización) como en la educación pre-universitaria debido a su sintaxis cercana al lenguaje natural y a la rapidez con que se pueden escribir programas simples, así como la gran cantidad de librerías que posee. Es comprensible entonces que muchos programas académicos consideren a Python como candidato para primer lenguaje. Sin embargo, la adopción de Python en ingeniería en computación debe evaluarse cuidadosamente, ya que existen indicadores empíricos de que una introducción exclusivamente con Python puede generar lagunas conceptuales.

Un estudio amplio que analizó el desempeño de estudiantes en ejercicios de programación básicos en cursos introductorios (comparando 10 cursos que usaban Python vs. 11 que usaban C++ en 20 universidades) tuvo un hallazgo sorprendente: los estudiantes de Python presentaron tasas de dificultad significativamente más altas que los estudiantes de C++ al resolver los mismos problemas [1]. En concreto, se midió que el 26 % de los estudiantes iniciados en Python necesitaron intentos excesivos o tiempo muy prolongado para resolver ciertas tareas,

mientras que en C++ ese porcentaje fue del 13 % [1]. Esta estadística es contundente, ya que contradice la creencia popular de que Python facilita el aprendizaje inicial. Incluso controlando por factores como si el curso tenía prerrequisitos o si los estudiantes provenían de carreras de computación vs. otras disciplinas, el efecto persistió [1]. Este dato actúa como *ground* basado en evidencia: sugiere que la "facilidad" de Python puede ser engañosa. Quizás su sintaxis no impone tantas trabas al comienzo, pero puede que los estudiantes se confíen o no desarrollen un pensamiento algorítmico riguroso, resultando en más dificultades cuando enfrentan problemas que requieren cuidado en los detalles. Otro estudio, de Johnson et al. (2020), analizó las ideas erróneas (*misconceptions*) que adquieren los estudiantes cuando Python es usado como primer lenguaje. Concluyó que Python por sí solo puede ser inapropiado como lenguaje introductorio en cursos de ciencias de la computación, debido a esas concepciones incorrectas que los estudiantes incorporan [10]. Por ejemplo, estudiantes podrían no entender cómo funciona la gestión de memoria (al no tener que pensar en ello en Python) o tener nociones imprecisas sobre tipos de datos (dado que Python es dinámico y no tipado estáticamente como C++). Este *respaldo* refuerza la idea de que, si se usa Python tempranamente, conviene complementarlo luego con lenguajes como C++ para cubrir esos vacíos. De hecho, en la mayoría de los planes de estudio locales que han incorporado Python, éste no ha desplazado a C,C++ sino que se ofrece como complemento en materias específicas (por ejemplo, para computación científica o machine learning en años superiores). En resumen, la defensa de C++ no implica descartar Python; por el contrario, un ingeniero en computación idealmente debería ser políglota en tecnología.

El punto es que Python cumple muy bien el rol de lenguaje de scripting y de alto nivel, pero no provee por sí solo la formación en pensamiento eficiente y cercano al sistema que C++ sí brinda. Una analogía útil es considerar que aprender solo Python sería análogo a aprender a conducir un automóvil automático sin conocer los fundamentos del motor, mientras que aprender C++ es más parecido a aprender a conducir un vehículo manual entendiendo qué ocurre con el motor. Un ingeniero formado con C++ podrá más adelante aprovechar Python de forma más efectiva, consciente de lo que ocurre tras bambalinas (por ejemplo, sabrá que operaciones intensivas de cómputo en Python conviene delegarlas a librerías internas escritas en C/C++ debido a las limitaciones de rendimiento de Python).

Hablando de rendimiento, es bien sabido que Python es mucho más lento que C++ en ejecución. Diversos *benchmarks* lo cuantifican: Python puede ser decenas de veces más lento que C++ en tareas intensivas de CPU. Un estudio sobre eficiencia energética de lenguajes mostró que Python consume hasta 45 veces más energía que C++ para ejecutar ciertas cargas de trabajo, correlacionado con ser decenas de veces más lento [5]. En la Figura 1 se ilustran resultados de un ensayo de eficiencia donde se comparan los 4 lenguajes más rápidos (C, Rust, C++ y Java) frente a 3 de los más lentos (Perl, Python, Ruby) en términos de consumo de energía y tiempo de ejecución para un mismo conjunto de programas de prueba. Es evidente la brecha de rendimiento: C++ ejecuta las tareas en

Top 4 Most Energy Efficient Programming Languages

Programming Language	Energy Consumption (J)	Speed of Execution (ms)
C	57	2,019
Rust	59	2,103
C++	77	3,155
Java	114	3,821

Figura 1: Comparativa de tiempo de ejecución y consumo energético entre lenguajes compilados (C, C++, Java, etc.) e interpretados (Python, Ruby, Perl) en un conjunto equivalente de tareas. Los lenguajes compilados muestran tiempos y consumos muy inferiores.

∼3.1 segundos consumiendo 77 joules, mientras Python requiere ∼145 segundos y 4390 joules [5]. Estas diferencias son críticas en contextos de ingeniería donde se procesan grandes volúmenes de datos o se requiere tiempo real. Si bien en un curso introductorio el rendimiento no es una preocupación inmediata, aquí el *warrant* es que enseñar C++ inculca desde temprano la noción de eficiencia algorítmica y de código, algo que puede quedar oculto en un entorno interpretado como Python donde casi cualquier solución "funciona" sin retroalimentación de performance hasta etapas muy posteriores.

En el ámbito local argentino, las universidades han empezado a incorporar Python principalmente en materias optativas o complementarias, no como base de la carrera. Por ejemplo, algunas facultades dictan cursos de Python para análisis de datos o introducción a la inteligencia artificial, como el FAMAF. Esto es positivo y refleja la realidad profesional donde Python es útil. Sin embargo, dichas universidades no han eliminado C++ de la currícula obligatoria; típicamente, el estudiante aprende C,C++ en primeros años y luego puede aprovechar Python en contextos de más alto nivel. Esta coexistencia parece ser la fórmula adecuada: C++ brinda los cimientos sólidos y Python permite construir herramientas rápidas para ciertas aplicaciones. A la inversa, sería riesgoso formar ingenieros que solo sepan Python esperando que "luego aprendan C++ si lo necesitan", ya que, como indicamos, se corre el riesgo de graduar profesionales sin entendimiento profundo de los costos computacionales o de cómo funcionan realmente las cosas a bajo nivel.

2.5. C++ en programación concurrente: Impacto del estándar C++20

Un aspecto crucial donde C++ aporta valor es en la enseñanza de la programación concurrente. Históricamente, materias como Sistemas Operativos o

Programación Concurrente utilizan C (con la biblioteca POSIX `pthreads`) o Java (con hilos) para introducir los conceptos de multihilo, sincronización y concurrencia. C++ heredó de C la capacidad de manejo explícito de hilos a bajo nivel, pero durante mucho tiempo careció de un soporte estándar cómodo, lo que hacía su uso pedagógico más complejo. Sin embargo, a partir de C++11 se incorporó en la biblioteca estándar un modelo de *threads* y *mutex* (std::thread, std::mutex, etc.), y más recientemente C++20 trajo mejoras significativas, incluyendo `std::jthread` (hilos con auto-*join* y cancelación cooperativa) y primitivas de sincronización de más alto nivel como semáforos (`std::counting_semaphore`), barreras y *latches*. Gracias a estas incorporaciones, hoy es posible enseñar concurrencia en C++ de forma más segura y sencilla. Por ejemplo, con `std::jthread` se evita el clásico error de olvidar unir (*join*) un hilo al terminar, ya que el destructor de `jthread` se encarga de ello automáticamente. Esto, combinado con las facilidades de alto nivel, permite que los estudiantes experimenten con hilos, exclusión mutua y comunicación entre hilos en C++ sin requerir tanto código *boilerplate* ni riesgo de errores sutiles como en el pasado. En una materia de Sistemas Operativos, poder usar C++20 implica que los estudiantes aplican en laboratorio los mismos conceptos vistos teóricamente (condiciones de carrera, secciones críticas, semáforos) con código relativamente conciso y cercano a aplicaciones reales. Esto crea un puente entre la teoría y la práctica profesional, ***reflexión***, dado que muchas herramientas industriales (ej., motores de juegos, software de redes, sistemas distribuidos) emplean C++ para sus componentes concurrentes por eficiencia. Asimismo, utilizar C++ en concurrencia complementa la formación recibida en estructuras de datos y algoritmos: al programar estructuras concurrentes (como colas multitarea, *thread pools*, etc.), los estudiantes deben pensar en consideraciones de bajo nivel (consistencia de memoria, atomicidad) a la par de diseñar soluciones algorítmicas, desarrollando un criterio de ingeniería integral. Un lenguaje que permite ejercitar concurrencia con control fino (como C++) asegura que el egresado comprenda los detalles del paralelismo a un nivel difícil de alcanzar con lenguajes totalmente administrados. Un egresado que implementó sincronización manual en C++ entenderá profundamente qué hace, por ejemplo, el *scheduler* de hilos de un sistema operativo o el recolector de basura paralelo de una JVM, proporcionándole un *know-how* valioso independientemente del lenguaje que use luego. Por supuesto, la complejidad de la concurrencia en C++ debe gestionarse pedagógicamente: se recomienda introducir estos tópicos una vez sentadas las bases de programación básica. Sin embargo, la disponibilidad de bibliotecas estándar modernas actúa a favor del docente, ya que se pueden crear ejemplos claros (ej. productor/consumidor, monitor) sin requerir bibliotecas externas ni entornos especiales, aprovechando el mismo lenguaje ya conocido por el estudiante.

Se podría argumentar, que otros lenguajes más nuevos (como Go o Rust) ofrecen modelos de concurrencia más seguros o sencillos. Si bien es cierto que Rust, por ejemplo, introduce verificaciones en tiempo de compilación para evitar condiciones de carrera, su adopción curricular aún es incipiente. C++ sigue siendo ampliamente usado con la capacidad de hacer dichas verificaciones a través

de flags del compilador y dominado en la industria, y las mejoras de C++20 reducen significativamente la brecha en facilidad de uso. Además, enseñar concurrencia en C++ no impide exponer a los estudiantes a modelos alternativos; al contrario, un ingeniero que entiende concurrencia en C++ podrá luego aprender modelos como el de Go (basado en *goroutines* o *green-threads*) o el de Rust con menor dificultad, pues cuenta con un *mental model* sólido de bajo nivel.

3. Conclusiones

A modo de síntesis, la enseñanza de C++ en la carrera de Ingeniería en Computación se justifica no por tradición, sino por evidencia pedagógica y necesidad formativa. Hemos afirmado que C++ es un lenguaje clave en la educación del ingeniero en computación, y los argumentos presentados – articulados mediante el modelo de Toulmin – apoyan sólidamente esta tesis. Los *fundamentos* incluyeron estudios académicos recientes que muestran mejor desempeño de estudiantes con C++ frente a lenguajes supuestamente más fáciles como Python [1], y análisis comparativos donde C++ demostró ser igual o más efectivo que Java en cursos introductorios de algoritmos [3]. Conectamos estos datos con la idea central (*warrant*) de que una formación que abarca desde bajo nivel hasta alto nivel empodera al futuro ingeniero: C++ obliga a pensar con precisión y eficiencia, cualidades que luego se transfieren al uso de cualquier otro lenguaje o herramienta. También se han considerado *repaldos* y refuerzos al argumento, tales como la demanda industrial de habilidades en C++ en campos diversos. Desde el desarrollo de videojuegos (donde motores como Unreal Engine emplean C++ ampliamente) [2] hasta la programación de sistemas financieros de trading algorítmico de alta frecuencia, o sistemas ciberfísicos complejos de la industria 5.0 el dominio de C++ abre puertas a oportunidades que difícilmente serían accesibles con solo conocimientos de lenguajes de alto nivel [11]. Incluso en aplicaciones científicas y de datos masivos, lenguajes como Python dependen críticamente de librerías internas escritas en C++ para obtener rendimiento (ej. `NumPy`, bibliotecas de aprendizaje automático), lo que convierte al ingeniero que sabe C++ en alguien capaz de contribuir a optimizar y desarrollar esas piezas de software fundamentales. Al mismo tiempo, reconocimos *qualifiers*: la importancia de C++ en la formación no implica descuidar otros lenguajes o paradigmas. Un currículo equilibrado puede (y debe) incluir exposición a lenguajes gestionados como Java, a lenguajes interpretados/dinámicos como Python, e incluso a lenguajes especializados según la rama (por ejemplo, Lingua Franca para el modelado y simulado de sistemas ciberfísicos, etc.). Lo crucial es el orden y el énfasis: ***no es lo mismo introducir primero a un estudiante en un lenguaje muy abstracto y luego enseñarle conceptos de sistemas, que hacer el camino inverso***. La evidencia sugiere que formar primero en C o C++ crea una base sólida sobre la cual añadir otros lenguajes es más sencillo [**?**]; por el contrario, comenzar con algo como Python podría requerir luego remediar ciertas carencias cuando se aborden temas de sistemas operativos, arquitectura o estructuras de datos avanzadas. En cuanto a posibles *refutaciones*, ninguna de

las examinadas debilita la conclusión de fondo. La mayor complejidad de C++ puede gestionarse didácticamente; las ventajas de otros lenguajes (simplicidad, rapidez de prototipado) no sustituyen el entendimiento que brinda C++, sino que lo complementan; y las tendencias actuales en la industria siguen valorando el perfil de ingeniero "completo" que entiende desde la electrónica hasta el software de aplicación. Es previsible que en el futuro surjan nuevos lenguajes y herramientas (por ejemplo, mayor adopción de Rust u otros lenguajes de nicho); sin embargo, un egresado que haya pasado por el rigor de programar en C++ estará mejor preparado para dominar dichas novedades rápidamente, a diferencia de alguien formado solo en ambientes muy abstractos.

En conclusión, la inclusión de C++ en la malla curricular de Ingeniería en Computación no es arbitraria sino estratégica. Los ingenieros formados con C++ desarrollan una mentalidad orientada a la optimización y al entendimiento pleno del funcionamiento de los sistemas computacionales. Aun si en su vida profesional cotidiana terminan usando mayormente otros lenguajes, la huella formativa de C++ se traduce en criterio técnico, capacidad de solucionar problemas complejos y versatilidad para adaptarse a distintas tecnologías. Como dijo alguna vez un renombrado educador en computación: "Enseñamos ciertos lenguajes no solo para que el estudiante los use, sino para que aprenda conceptos que trascienden el lenguaje". C++ enseña conceptos de computación de una manera que pocos lenguajes logran. Por ende, sigue y seguirá siendo un pilar en la educación del ingeniero en computación en Argentina y en el mundo, cimentando las bases para profesionales integrales capaces de enfrentar desafíos desde sistemas embebidos hasta sistemas de alto rendimiento.

Referencias

1. N. Alzahrani, S. Romero, and C. Rich, "Python versus C++: An analysis of student struggle on small coding exercises in introductory programming courses," in *Proc. 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*, 2018, pp. 86–91.
2. S. Hooper, S. Hunt, J. Elsworth, and G. McKeown, "Teaching game programming in an upper-level computing course through the development of a C++ framework and middleware," in *Eurographics 2024 – Education Papers*, 2024, pp. 45–52.
3. M. S. Naveed, "Comparison of C++ and Java in implementing introductory programming algorithms," *QUEST Research Journal*, vol. 19, no. 1, pp. 95–103, 2021.
4. X. He, Y. Huang, and M. Li, "Research on teaching reform of C programming language course for deep integration of professional fields," *Open Journal of Social Sciences*, vol. 10, pp. 267–275, 2022.
5. Eco-Programming Alliance, "The most efficient and environmentally friendly programming languages in 2024," Softjourn Inc., Feb. 2023. [Online]. Available: `https://softjourn.com/insights/environmentally-friendly-programming-languages` (Accedido: 01 de Abril, 2025).
6. IEEE Spectrum, "Top programming languages 2022," *IEEE Spectrum*, Aug. 2022. [Online]. Available: `https://spectrum.ieee.org/top-programming-languages` (Accedido: 01 de Abril, 2025).

7. UTN Buenos Aires, "Programa Algoritmos y Estructuras de Datos," [Online]. Available: https://www.frba.utn.edu.ar/wp-content/uploads/2022/02/algoritmosyestructuradedatos08web.pdf (Accedido: 01 de Abril, 2025).
8. UTN Córdoba, "Programa Algoritmos y Estructuras de Datos," [Online]. Available: https://www.institucional.frc.utn.edu.ar/sistemas/noticias/ACA/Modalidades/2024/E38_planificacion_aed_2024_v1.0.pdf (Accedido: 01 de Abril, 2025).
9. Comisión de Carrera Ing. en Computación UNC, "Plan de desarrollo de la carrera de Ingeniería en Computación, UNC-FCEFyN," Córdoba, Argentina, 2011. [Online]. Available: https://fcefyn.unc.edu.ar/documents/2285/Plan_Desarrollo_Ing_Computacion_Cordoba_2011_v9.pdf (Accedido: 01 de Abril, 2025).
10. F. Johnson, M. Patra, and S. Ho, "Analysis of student misconceptions using Python as an introductory programming language," in *Proc. 4th Conf. on Computing Education Practice (CEP'20)*, 2020, pp. 1–4.
11. L. Trejo, "High-frequency trading secret: Code with C++," *Medium.com*, Feb. 2023. [Online]. Available: https://medium.com/@luiggitrejo/high-frequency-trading-secret-code-with-c-d63e0983373a (Accedido: 01 de Abril, 2025).
12. M. Roberts, "C++ in embedded systems," *Quantum Zeitgeist*, Jan. 2021. [Online]. Available: https://quantumzeitgeist.com/c-in-embedded-systems/ (Accedido: 01 de Abril, 2025).