Programación Básica para Ingeniería en Sistemas de Información:
De los Sistemas de Información al Código

Gleiston Guerrero Ulloa

Índice general

Prefacio

Parte I

Fundamentos Computacionales para Sistemas de Información

Capítulo 1

Sistemas de Información, Tecnología y Programación Aplicada

La formación en Ingeniería en Sistemas de Información exige una comprensión articulada entre estructuras organizacionales y mecanismos computacionales. La programación constituye el medio formal mediante el cual las necesidades operativas se traducen en soluciones ejecutables, integrando procesos, datos y reglas de negocio dentro de arquitecturas tecnológicas coherentes. La literatura especializada sostiene que el desarrollo de aplicaciones fortalece la capacidad de análisis estructurado y modelización de problemas empresariales [1], [2]. El dominio de fundamentos computacionales permite diseñar sistemas alineados con dinámicas administrativas reales y con entornos tecnológicos sujetos a evolución constante.

La disciplina integra personas, procesos y tecnología bajo un marco que combina análisis organizacional con implementación técnica. El software opera como mecanismo de materialización de políticas institucionales y controles operativos [3]. Desde la ingeniería de software se establecen criterios de trazabilidad, consistencia y verificabilidad que inciden directamente en la calidad del sistema construido [4]. El dominio de la programación habilita la construcción de aplicaciones capaces de gestionar inventarios, nóminas, transacciones financieras o registros académicos, ampliando el horizonte profesional del ingeniero y fortaleciendo su capacidad de intervención técnica en organizaciones de diversa escala.

La actividad del desarrollador trasciende la mera codificación; involucra modelado estructural, definición de reglas formales y validación sistemática de resultados. La automatización administrativa demanda control preciso de estructuras condicionales, ciclos y manejo de datos [5]. Los estándares técnicos sobre prácticas seguras de programación proporcionan lineamientos que reducen defectos y aumentan la confiabilidad del software [6].

La práctica sostenida consolida disciplina analítica, rigor metodológico y responsabilidad profesional.

Los sistemas de información transforman datos en insumos útiles para la gestión. La captura, almacenamiento y procesamiento algorítmico de registros digitales constituye la base operativa de dichos sistemas [7]. La experiencia directa con estructuras de control y funciones permite observar cómo los datos adquieren significado operativo cuando son sometidos a transformaciones formales. Estudios sobre ecosistemas informacionales muestran que la organización estructurada de datos incrementa la capacidad analítica institucional [8]. La programación habilita participación activa en la construcción de estos mecanismos informacionales.

La infraestructura computacional sostiene la ejecución de aplicaciones mediante la interacción coordinada entre procesadores, memoria y sistemas operativos [9]. La gestión de procesos y recursos, descrita en la teoría de sistemas operativos, determina el comportamiento observable del software en ejecución [10]. Conocer estos fundamentos permite escribir código que utilice racionalmente los recursos disponibles y que opere de manera estable en plataformas heterogéneas. La formación técnica proporciona criterio para evaluar rendimiento, eficiencia y adecuación arquitectónica.

La representación digital introduce al estudiante en la estructura formal de los datos en memoria. El estándar IEEE 754 define la organización de números en punto flotante mediante campos de signo, exponente y fracción [11]. El análisis clásico de errores de redondeo evidencia cómo pequeñas variaciones en la representación binaria influyen en cálculos posteriores [12]. Este conocimiento favorece decisiones técnicas informadas al diseñar modelos financieros, estadísticos o administrativos que dependen de precisión numérica.

La formulación rigurosa del problema computacional constituye la base del diseño de soluciones informacionales. La identificación sistemática de entradas, salidas y restricciones permite estructurar modelos verificables [13]. El diseño algorítmico proporciona criterios formales de finitud, precisión y corrección, garantizando coherencia interna en la solución implementada. La práctica continua en este ámbito fortalece el razonamiento estructurado y la capacidad de afrontar desafíos organizacionales mediante procedimientos definidos.

El modelo curricular IS2020 plantea que el profesional en Sistemas de Información debe integrar análisis organizacional con implementación técnica en plataformas reales [14], [15]. La programación actúa como puente entre teoría administrativa y ejecución tecnológica, permitiendo materializar procesos empresariales en sistemas operativos concretos. El dominio de estas competencias amplía oportunidades laborales y consolida una formación

integral orientada a la transformación digital organizacional.

1.1. Ingeniería en Sistemas de Información y programación aplicada

La Ingeniería en Sistemas de Información (IS) se orienta a comprender, diseñar e implementar soluciones socio-técnicas en las que personas, procesos, datos y reglas organizacionales se integran mediante sistemas de software. En este ámbito, el software no se concibe como un producto aislado, sino como un mecanismo de coordinación y control que operacionaliza políticas, procedimientos y restricciones bajo condiciones reales de uso (tiempos, recursos, variabilidad y excepciones). Por ello, la programación adquiere relevancia formativa y profesional: permite convertir descripciones organizacionales en comportamientos ejecutables, verificables y mantenibles sobre plataformas tecnológicas concretas [1], [2], [16].

En un curso de programación básica para IS, el objetivo no consiste únicamente en "aprender un lenguaje", sino en adquirir una capacidad transferible: representar problemas con precisión, diseñar soluciones paso a paso y expresar dichas soluciones como algoritmos que un computador pueda ejecutar sin ambigüedades. Esta capacidad se relaciona con el pensamiento computacional, que enfatiza abstracción, descomposición y formulación sistemática de procedimientos, habilidades útiles para contextos organizacionales donde los problemas suelen estar incompletos o descritos informalmente [17], [18], [19].

Desde una perspectiva curricular basada en competencias, la programación se valida por su utilidad para intervenir en tareas reales de análisis, construcción e implementación de soluciones que atienden necesidades de actores y unidades organizacionales. En consecuencia, aprender a programar en IS implica articular conceptos (algoritmos, datos, control) con prácticas elementales de ingeniería (especificar, probar, documentar y mejorar), de modo que el estudiante avance desde ejercicios controlados hacia la resolución de problemas con significado organizacional [2], [14], [15].

En este capítulo, la programación se presenta como una herramienta que permite: (i) formalizar procesos organizacionales mediante reglas explícitas; (ii) automatizar operaciones repetitivas con criterios de calidad; y (iii) producir resultados consistentes que sostienen decisiones. El desarrollo se mantiene deliberadamente en un nivel conceptual y de aprendizaje, reservando para secciones posteriores el tratamiento detallado de transformación digital, responsabilidades profesionales y ciclo de vida del dato, con el fin de evitar redundancias y preservar la progresión pedagógica [1], [2], [16].

1.1.1. La programación como competencia profesional en Sistemas de Información

En IS, la programación se reconoce como una competencia transversal porque conecta el análisis del entorno organizacional con la implementación concreta de mecanismos de información, control y coordinación. Esta conexión no es abstracta: en la práctica profesional, decisiones operativas y administrativas quedan codificadas en reglas de validación, cálculos, flujos de trabajo y restricciones que el software ejecuta de manera consistente para múltiples usuarios y situaciones [1], [2], [16].

Para el estudiante que aprende a programar, esta transversalidad se traduce en una idea guía: un programa es una especificación ejecutable de un procedimiento. En términos formativos, ello exige dominar nociones fundamentales (variables, tipos, expresiones, estructuras de control, modularidad) y, al mismo tiempo, aprender a justificar la elección de una estrategia algorítmica frente a un requerimiento. Dicho aprendizaje se apoya en la disciplina algorítmica: claridad, finitud, corrección y trazabilidad entre el problema y la solución [5], [7], [13].

Además, la programación fortalece habilidades de abstracción y descomposición aplicables a dominios no exclusivamente informáticos. La capacidad de separar un problema en subproblemas, definir entradas y salidas, y construir soluciones incrementales constituye un fundamento metodológico para diseñar soluciones organizacionales robustas, incluso cuando los requisitos iniciales son ambiguos o cambian con el contexto [17], [18], [19].

Desde la perspectiva de IS2020, el dominio técnico adquiere valor cuando se integra con comprensión organizacional, comunicación y pensamiento crítico para producir soluciones utilizables. En consecuencia, aprender a programar debe orientarse a producir artefactos comprensibles, verificables y evolucionables, no únicamente a “hacer que el código funcione” [14], [15], [16].

Programación como medio de formalización de procesos organizacionales

Los procesos organizacionales suelen describirse mediante lenguaje natural, documentos operativos o acuerdos tácitos. Sin embargo, para ser automatizables y auditables requieren una traducción a reglas explícitas, condiciones y secuencias de acciones que un sistema pueda ejecutar de forma consistente. La programación cumple aquí una función de formalización: convierte decisiones, excepciones, validaciones y dependencias en estructuras algorítmicas precisas que reducen ambigüedades [5], [7], [13].

En clave pedagógica, formalizar significa aprender a “cerrar” lo que el lenguaje natural deja abierto. Por ejemplo, una frase organizacional como “registrar la solicitud si está completa” obliga a precisar qué se entiende por completitud (campos obligatorios, formatos válidos, rangos aceptables) y qué ocurre si la condición falla (rechazo, corrección, notificación). Estos aspectos se expresan mediante condicionales, validaciones de entrada y manejo de casos límite, elementos básicos que estructuran el razonamiento del programador principiante [5], [6], [7].

La formalización también exige representar con precisión los estados del proceso y sus transiciones. Aunque el modelamiento de procesos se tratará de manera más extensa en otras secciones, aquí basta destacar que la programación permite materializar estados como valores y condiciones (por ejemplo, “pendiente”, “aprobado”, “rechazado”), y transiciones como reglas que dependen de eventos y verificaciones. Esta correspondencia entre proceso y control de flujo introduce al estudiante en el diseño de soluciones que no son lineales, sino gobernadas por decisiones y excepciones [1], [13], [16].

Finalmente, la formalización mediante programación aporta un lenguaje común entre lo organizacional y lo técnico. Modelos de análisis y diseño (por ejemplo, orientados a objetos o basados en UML) describen estructuras y comportamientos; la programación convierte dichas descripciones en componentes ejecutables en los que cada operación implementada representa una regla o decisión del proceso. La trazabilidad conceptual entre modelo e implementación se vuelve, así, una disciplina temprana que el estudiante debe aprender a cultivar desde ejercicios básicos [4], [16], [20].

Programación como instrumento de automatización operativa

La automatización operativa delega en procedimientos computacionales aquellas actividades repetitivas, intensivas en registro o propensas a error humano, con el fin de lograr ejecución uniforme y medible. En IS, esta automatización se expresa mediante programas que capturan datos, validan transacciones, aplican reglas y producen salidas operativas (cálculos, estados, comprobaciones, reportes), integrándose con los sistemas de información que sostienen la operación organizacional [1], [2], [3].

Desde el aprendizaje de programación, automatizar no equivale a “mecanizar” un procedimiento sin comprenderlo. Automatizar exige identificar entradas, decidir el orden de los pasos, definir condiciones de aceptación y establecer salidas verificables. En términos de construcción algorítmica, ello se traduce en seleccionar estructuras de control adecuadas (secuencia, selección, iteración), diseñar funciones para dividir responsabilidades y aplicar

pruebas simples para confirmar que el programa reproduce el procedimiento esperado [5], [7], [13].

La automatización también introduce un criterio didáctico esencial: la corrección debe sostenerse en la diversidad de casos, no solo en el "caso típico". Por ello, el estudiante debe aprender a anticipar excepciones (datos incompletos, formatos inválidos, valores fuera de rango) y a diseñar validaciones y mensajes de error que hagan explícito el contrato del programa. Estos aspectos se relacionan con prácticas de programación responsable y con lineamientos que buscan reducir fallas recurrentes asociadas a suposiciones inseguras o entradas no controladas [4], [6], [7].

Finalmente, automatizar con impacto organizacional requiere considerar calidad y mantenibilidad. La ingeniería de software destaca propiedades como corrección, claridad, modularidad y adecuación al propósito, que se vuelven críticas cuando un proceso depende del software para operar. En consecuencia, aprender a programar debe incluir hábitos tempranos de estructura y legibilidad (nombres, modularización, comentarios mínimos justificados), ya que estos hábitos condicionan la sostenibilidad del sistema más allá del primer funcionamiento [1], [4], [21].

Programación como soporte informacional para la toma de decisiones

La toma de decisiones organizacionales depende de transformar datos en información contextualizada y, posteriormente, en conocimiento accionable. Los sistemas de información sostienen esa transformación mediante mecanismos de captura, almacenamiento, procesamiento y presentación, donde los programas implementan operaciones que filtran, validan, agregan y derivan resultados que funcionan como evidencia para decidir [2], [8], [22].

En términos formativos, este rol informacional permite introducir al estudiante en tareas de programación con sentido: calcular totales, promedios, frecuencias, máximos y mínimos; clasificar registros; detectar inconsistencias; o generar resúmenes que hagan visible el estado de una operación. Estas tareas, aunque básicas, enseñan a pensar en términos de datos, reglas y resultados verificables, y facilitan comprender por qué la precisión en tipos, operaciones y validaciones afecta directamente la confiabilidad de la información producida [5], [7], [22].

El soporte informacional exige consistencia: decisiones distintas no deben originarse en cálculos inconsistentes o reglas implícitas. Por ello, la programación debe explicitar supuestos (por ejemplo, unidades, rangos, criterios de inclusión), y establecer controles que permitan reproducibilidad del procesamiento. Esta exigencia se conecta con la disciplina

algorítmica (precisión y finitud) y con criterios de calidad del software (comportamiento estable y verificable), aspectos que deben introducirse desde el nivel básico para evitar que el estudiante normalice resultados "aproximados" sin justificación [4], [13], [21].

Además, la confiabilidad informacional se asocia a la correcta representación de valores numéricos, especialmente cuando se realizan cálculos acumulativos o comparaciones. En programación, comprender límites de representación y el manejo apropiado de tipos numéricos fortalece la integridad de los resultados y reduce errores sutiles que afectan reportes e indicadores. Aunque los detalles de representación digital se abordarán posteriormente, aquí se subraya su impacto directo en la calidad del soporte informacional [7], [11], [12].

1.1.2. Relación entre procesos organizacionales y software

Un sistema de información puede entenderse como la materialización computacional de un conjunto de procesos organizacionales que gestionan recursos, eventos, registros y comunicaciones. En consecuencia, la calidad del software depende del entendimiento del proceso que modela y de la coherencia entre el procedimiento operativo y la lógica implementada, ya que el programa se convierte en un "ejecutor" de reglas organizacionales en escenarios reales [1], [2], [16].

Esta relación es bidireccional. Por un lado, los procesos condicionan requisitos y reglas del software; por otro, el software introduce capacidades de trazabilidad, validación sistemática y generación de evidencia que reconfiguran la forma en que el proceso se ejecuta y se controla. Para quien aprende a programar, esta bidireccionalidad se traduce en comprender que cada decisión de diseño (estructuras de datos, validaciones, orden de pasos) produce consecuencias operativas: habilita o restringe acciones, reduce o incrementa errores, y determina la calidad de la información disponible [2], [3], [4].

En el plano pedagógico, esta subsección cumple un propósito específico: ayudar a que el estudiante no programe "en el vacío". Incluso en ejercicios introductorios, se busca que identifique qué parte del proceso representa el programa (qué se recibe, qué se decide, qué se produce) y que evalúe la solución por criterios de corrección y claridad, no solo por ejecución. Con ello, la programación se integra como una práctica de análisis aplicado, coherente con marcos de análisis y diseño de sistemas [1], [5], [16].

Proceso, procedimiento y sistema en entornos organizacionales

En IS resulta útil distinguir: (i) *proceso* como conjunto de actividades coordinadas orientadas a un resultado organizacional; (ii) *procedimiento* como la forma prescrita o normativa

de ejecutar parte del proceso; y (iii) *sistema* como el conjunto socio-técnico que integra personas, reglas y tecnología para ejecutar y controlar dichos procesos. Estas distinciones son relevantes porque el software no implementa "la organización" en abstracto, sino comportamientos concretos asociados a procedimientos y reglas dentro de un proceso delimitado [1], [2], [3].

Para aprender a programar, estas distinciones ofrecen una guía práctica: un programa suele representar un procedimiento (una forma específica de hacer) dentro de un proceso mayor. Por ello, antes de escribir código, el estudiante debe aprender a identificar el alcance: qué parte del proceso será automatizada, qué decisiones quedan dentro del programa y cuáles permanecen bajo control humano. Esta delimitación reduce errores de diseño típicos en principiantes, como querer resolver "todo el proceso" con una sola rutina o ignorar casos excepcionales relevantes [1], [5], [7].

La precisión conceptual evita errores de alcance en proyectos de software. Cuando no se distingue proceso de procedimiento, se tiende a sobredimensionar funciones del sistema o a automatizar actividades que requieren juicio humano o validación externa. Por ello, marcos de análisis y diseño recomiendan delimitar responsabilidades, entradas y salidas por función, y especificar la interacción de usuarios y componentes con el proceso organizacional [1], [16], [20].

En consecuencia, una práctica formativa adecuada consiste en traducir un procedimiento a una especificación mínima: entradas, reglas y salidas. Esta especificación se convierte en una estructura de programa con control de flujo explícito y validaciones claras, lo que refuerza la disciplina algorítmica y disminuye suposiciones implícitas sobre la dinámica organizacional [4], [6], [13].

Modelamiento de procesos y necesidades informacionales

El modelamiento constituye el puente entre la descripción de un proceso y su instrumentación computacional. En IS, modelar significa representar actividades, decisiones, actores, datos y reglas, con el fin de identificar qué información debe capturarse, en qué momento, bajo qué validaciones y con qué propósito operativo o gerencial. Esta representación no es un formalismo ornamental: orienta el diseño del programa, reduce ambigüedades y establece criterios para evaluar la solución [1], [2], [16].

En este sentido, la necesidad informacional no se reduce a "guardar datos", sino a definir semántica y utilidad: qué atributos representan un evento, qué consistencia deben mantener, y cómo habilitan operaciones de control o seguimiento. Para el aprendizaje de

programación, ello implica entrenar al estudiante en la elección de estructuras de datos coherentes con el significado de la información y en la implementación de validaciones que preserven la integridad del registro [2], [8], [22].

El modelamiento también orienta decisiones de diseño: modularidad, responsabilidades y operaciones. En enfoques orientados a objetos, por ejemplo, el modelo facilita la identificación de entidades, relaciones y comportamientos que luego se traducen a clases y métodos; incluso cuando se programa de manera introductoria, esta perspectiva ayuda a organizar el código en partes comprensibles y reutilizables, evitando soluciones monolíticas difíciles de mantener [4], [16], [20].

Como consecuencia, el estudiante aprende que "programar" incluye un trabajo previo de clarificación: construir una representación mínima del proceso y de sus datos, a partir de la cual se derivan reglas y casos de prueba elementales. Esta práctica refuerza el vínculo entre lo que el proceso requiere y lo que el programa ejecuta, con criterios verificables de corrección [1], [5], [7].

De la necesidad organizacional a la solución computacional

La transición desde una necesidad organizacional hacia una solución computacional exige transformar objetivos (reducir errores, mejorar control, acelerar registro, producir reportes) en especificaciones operables: funciones, reglas y restricciones que puedan implementarse y verificarse. En análisis y diseño de sistemas, esta transición se estructura mediante identificación de requerimientos, definición de alcance y elaboración de modelos que conectan actores, procesos y datos con funcionalidades del software [1], [2], [16].

Desde la perspectiva del aprendizaje, esta transición se traduce en una secuencia disciplinada: (i) comprender el enunciado y sus restricciones; (ii) definir entradas y salidas con precisión; (iii) descomponer en subproblemas; (iv) diseñar un algoritmo; y (v) implementar de forma incremental con verificación continua. Este enfoque evita que el estudiante confunda la escritura de código con la solución del problema, y promueve que la implementación sea una consecuencia de un diseño algorítmico explícito [5], [7], [13].

La programación cierra el ciclo al convertir dichas especificaciones en algoritmos y estructuras de datos ejecutables, asegurando que el comportamiento real del sistema corresponda al comportamiento requerido. En términos formativos, esto implica habituarse a la verificación: comprobar resultados, probar escenarios alternativos y revisar supuestos. De este modo, el estudiante desarrolla un criterio profesional temprano: el programa es correcto cuando satisface el problema bajo condiciones relevantes, no cuando solo produce

una salida “razonable” [4], [13], [21].

Conviene enfatizar que, en esta sección, la relación se plantea a nivel conceptual y de trazabilidad: necesidad → modelo → implementación. Los métodos detallados para formular entradas, salidas, restricciones y criterios de aceptación, así como su articulación con otras dimensiones organizacionales, se desarrollan en secciones posteriores del capítulo. Esta decisión editorial evita redundancias y sostiene una progresión pedagógica que acompaña el aprendizaje de programación desde fundamentos hacia aplicaciones más integrales [1], [15], [16].

1.2. Rol profesional del desarrollador en Sistemas de Información

El rol profesional del desarrollador en Sistemas de Información (IS) se define por la capacidad de traducir necesidades organizacionales en soluciones de software que operan bajo restricciones reales: tiempo, presupuesto, normativas, interacción humana y evolución de procesos. Esta labor exige convertir descripciones funcionales en comportamientos ejecutables, con reglas explícitas, datos bien definidos y resultados verificables, de modo que el sistema sostenga operaciones y decisiones sin degradar la coherencia del proceso organizacional [1], [2], [16].

Dentro de este panorama, aprender a programar constituye una decisión formativa orientada a la empleabilidad y al desempeño profesional. La programación habilita al estudiante a pasar de la intención a la implementación: formular un algoritmo, elegir estructuras de datos, controlar la ejecución y validar resultados mediante pruebas. Esta progresión convierte el aprendizaje en una práctica acumulativa, donde cada concepto (secuencia, selección, iteración, modularidad) aporta un recurso concreto para construir soluciones aplicables en distintos contextos organizacionales [5], [7], [13].

La práctica profesional no se limita a producir código funcional; incluye asegurar calidad, mantenibilidad y seguridad desde etapas tempranas. Por esa razón, el desarrollo de competencias técnicas incorpora hábitos de diseño, lectura de requisitos, documentación mínima útil y revisión sistemática de defectos frecuentes. Este marco garantista se vincula con la ingeniería de software, donde la corrección técnica se evalúa junto con claridad, evolución y adecuación al propósito [4], [23], [24].

1.2.1. Automatización, digitalización y transformación organizacional

La automatización y la digitalización representan dos líneas de intervención que suelen confluir en iniciativas de transformación digital. En IS, el desarrollador participa en estas iniciativas implementando mecanismos que capturan información, normalizan registros, ejecutan reglas y coordinan actividades, buscando que el sistema reduzca fricción operativa y produzca evidencia útil para control y mejora [2], [16], [25].

El aprendizaje de programación contribuye directamente a este tipo de intervención. Un estudiante que domina estructuras de control y modelado básico de datos puede implementar validaciones, automatizar cálculos, generar reportes y construir prototipos que hagan visibles fallas del proceso o inconsistencias del registro. Esa experiencia temprana favorece un enfoque riguroso: cada decisión de diseño se expresa como una regla ejecutable y cada regla se verifica con casos de prueba y entradas variadas [5], [6], [7].

Paralelamente, la transformación organizacional demanda sensibilidad socio-técnica. El software modifica rutinas, tiempos y responsabilidades; por ello, el desarrollador debe anticipar efectos sobre usuarios y sobre la interacción con el sistema. Aun en programación básica, resulta pertinente introducir ejercicios que consideren uso real: mensajes de error informativos, flujos de entrada consistentes y salidas interpretables por usuarios no técnicos [1], [2], [26].

Digitalización de información y procesos

La digitalización consiste en representar información y actividades de manera que puedan almacenarse, consultarse y procesarse mediante sistemas computacionales. En organizaciones, este paso implica identificar qué datos describen un evento, qué atributos garantizan trazabilidad y qué reglas preservan integridad, evitando registros incompletos o ambiguos que comprometan la operación [2], [22], [27].

Para quien aprende a programar, la digitalización se materializa en decisiones concretas: definir tipos de datos, estructurar registros, validar entradas y diseñar formatos de salida. Un formulario, un archivo de datos o una estructura en memoria no son detalles secundarios; constituyen la base del comportamiento posterior del sistema. Por ello, el entrenamiento debe incluir prácticas de validación (rangos, formatos, obligatoriedad) y de coherencia semántica (nombres consistentes, unidades, estados), destacando que una buena representación reduce errores aguas abajo [5], [6], [7].

De manera complementaria, la digitalización se conecta con principios de gestión de

datos en sistemas de información. Modelos de datos y conceptos de bases de datos ayudan a evitar duplicidad, inconsistencias y pérdidas de significado al pasar de registros aislados a estructuras persistentes. Sin profundizar en administración de bases (tema desarrollado en otras unidades), conviene que el estudiante perciba la relación entre "cómo se programa" y "qué calidad tienen los datos" [27], [28], [29].

Automatización de operaciones y flujos de trabajo

La automatización se orienta a ejecutar tareas repetitivas con uniformidad, control y medición, integrando reglas operativas en procedimientos computacionales. En IS, esta automatización se implementa mediante programas que registran transacciones, calculan resultados, aplican restricciones y coordinan secuencias de trabajo. En organizaciones con alta carga operativa, pequeños incrementos de consistencia y velocidad producen mejoras acumulativas en calidad del servicio y trazabilidad [1], [2], [3].

El aprendizaje de programación ofrece un camino pedagógico natural para esta automatización. Primero se ejercita la especificación de pasos; luego se introduce la selección condicional para manejar decisiones; posteriormente se incorporan bucles para procesar conjuntos de registros; después se emplea modularidad para separar responsabilidades y facilitar pruebas. Esta ruta promueve un estilo de pensamiento donde el estudiante aprende a transformar un procedimiento narrado en un algoritmo que el computador ejecuta sin interpretaciones implícitas [5], [7], [13].

No obstante, la automatización requiere atención a fallos frecuentes: entradas inválidas, supuestos no verificados y manejo insuficiente de excepciones. La calidad operativa depende de validar datos, registrar errores de forma útil y conservar invariantes del proceso. Catálogos y lineamientos de codificación ayudan a identificar riesgos habituales y favorecen prácticas tempranas de programación responsable, incluso en ejercicios introductorios [4], [6], [24].

En distintos contextos organizacionales, la automatización también se implementa como orquestación de tareas y flujos. Sistemas de gestión de procesos (BPMS) y enfoques afines muestran que las reglas de negocio y los pasos de un flujo pueden representarse, monitorearse y refinarse. Para el estudiante, estas ideas se traducen en diseñar programas que hagan explícitas transiciones, estados y condiciones, preparando el terreno para trabajos más avanzados sin anticipar contenidos de unidades posteriores [1], [16], [30].

Transformación digital orientada al valor organizacional

La transformación digital orientada al valor se vincula con cambios en la forma de operar y decidir, apoyados por software, datos y rediseño de procesos. Su propósito no se reduce a informatizar tareas existentes; introduce nuevas capacidades de coordinación, medición y mejora continua, alineadas con objetivos de desempeño, servicio y sostenibilidad organizacional [25], [31], [32].

Desde la formación en programación, la conexión con el valor se enseña mediante problemas bien planteados: definir qué variable representa un indicador, qué regla determina una condición de alerta, qué estructura permite consolidar registros, y cómo se verifica que el resultado refleje la intención del proceso. La finalidad de ilustrar esta relación consiste en que el lector perciba que cada estructura de control y cada diseño de datos tienen impacto directo sobre la utilidad del sistema [2], [5], [7].

En contraste con una visión centrada en herramientas, el aprendizaje orientado al valor prioriza principios: consistencia, verificabilidad, trazabilidad y mantenibilidad. Un programa con resultados correctos pero difícil de leer o modificar impone costos de cambio; un programa legible y modular facilita adaptación ante cambios de política o de proceso. Esta relación entre diseño y evolución se encuentra documentada en fundamentos de ingeniería de software, con énfasis en calidad y ciclo de vida [4], [23], [24].

1.2.2. Responsabilidades y competencias técnicas en IS

El desarrollador en IS asume responsabilidades técnicas que exceden la escritura de código: interpretar requisitos, estructurar soluciones, verificar corrección y comunicar decisiones de diseño. Estas responsabilidades se ejercen con mayor solidez cuando el aprendizaje de programación se acompaña de hábitos de razonamiento: formular supuestos, delimitar alcance, definir contratos de entrada/salida y construir pruebas que respalden el resultado [1], [4], [16].

El modelo de competencias IS2020 destaca la integración entre capacidades técnicas, pensamiento analítico y comunicación profesional. Por ello, el aprendizaje de programación debe presentarse como una competencia de articulación: convierte una necesidad en un algoritmo, un algoritmo en una implementación, y una implementación en un componente operable dentro de un sistema socio-técnico [2], [14], [15].

Asimismo, la responsabilidad profesional incorpora prácticas de calidad que deben iniciarse desde programación básica. La estructura del código, la claridad de nombres, el uso de funciones y la revisión de errores comunes son prácticas que evitan deuda técnica

temprana y preparan al estudiante para proyectos colaborativos. La ingeniería de software provee criterios y técnicas para sostener estas prácticas en el tiempo [23], [24], [33].

Pensamiento lógico y abstracción aplicada

El pensamiento lógico aplicado se expresa en la habilidad de construir una solución paso a paso y justificar cada transición del algoritmo. En programación básica, esta habilidad se cultiva al formular procedimientos precisos, identificar condiciones y diseñar estructuras de control que representen decisiones del problema. La disciplina algorítmica proporciona un marco para evaluar si un procedimiento es correcto, finito y ejecutable [5], [7], [13].

La abstracción aplicada consiste en seleccionar lo relevante del problema y representarlo con estructuras mínimas: variables, colecciones, funciones y tipos. En IS, esta selección tiene una dimensión pragmática: se abstrae con el objetivo de operar sobre datos y reglas organizacionales sin perder significado. La formación debe favorecer ejercicios que obliguen a decidir qué información se guarda, qué se calcula, qué se valida y qué se omite por carecer de utilidad operacional [2], [18], [22].

El aprendizaje mejora cuando se incorpora una práctica incremental: comenzar con una versión simple que funcione, ampliar para cubrir casos adicionales y refactorizar para conservar claridad. Este patrón enseña al lector a programar con control del crecimiento del código, evitando soluciones extensas y frágiles. En ingeniería de software, esta idea se relaciona con modularidad y evolución de sistemas, aportando criterios para decidir cuándo dividir, renombrar o reorganizar [4], [24], [33].

Calidad, mantenibilidad y responsabilidad profesional

La calidad del software se manifiesta en corrección, legibilidad, seguridad y adaptabilidad. En IS, estas propiedades impactan operaciones y decisiones, ya que el sistema ejecuta reglas de negocio, valida transacciones y produce información para seguimiento. La responsabilidad profesional del desarrollador incluye prevenir defectos previsibles, documentar supuestos y asegurar que el sistema responda de manera consistente ante entradas diversas [4], [6], [23].

La mantenibilidad depende de estructuras comprensibles por terceros y por el propio autor en el tiempo. Para el lector que aprende a programar, esta idea se traduce en hábitos observables: nombres informativos, funciones con una responsabilidad definida, manejo explícito de errores y reducción de duplicación. Buenas prácticas de diseño y limpieza de código favorecen el aprendizaje porque convierten el programa en un objeto legible que puede discutirse, corregirse y mejorar con criterios explícitos [7], [24], [33].

La responsabilidad también incluye atención a fallas vinculadas a seguridad y robustez, incluso en programas pequeños. Validar entradas, controlar rangos y manejar condiciones inesperadas previene comportamientos indeseados y errores silenciosos que contaminan datos o producen resultados equivocados. Lineamientos de codificación y estándares de ingeniería proporcionan referencias para evitar patrones inseguros y para estructurar verificaciones de forma sistemática [4], [6], [23].

En síntesis, aprender a programar con orientación a IS implica adoptar una ética técnica: producir código que funcione, que pueda evolucionar y que proteja el significado de los datos y reglas que implementa. Esta ética se consolida cuando el estudiante aprende a justificar decisiones de diseño y a evaluar consecuencias, articulando práctica algorítmica con calidad de ingeniería [4], [13], [24].

Colaboración interdisciplinaria y documentación técnica

Los proyectos de IS requieren colaboración con actores de negocio, usuarios finales, analistas, administradores de datos y responsables de infraestructura. El desarrollador debe comunicar decisiones, restricciones y supuestos en un lenguaje accesible para perfiles no técnicos, manteniendo precisión suficiente para evitar interpretaciones erróneas. Por ello, la documentación técnica se consolida como un componente profesional del desarrollo, asociado a trazabilidad y a coordinación del trabajo [1], [2], [16].

Para quien aprende a programar, la colaboración se entrena mediante prácticas concretas: describir el algoritmo antes del código, comentar decisiones no obvias, registrar precondiciones y postcondiciones de funciones, y producir ejemplos de entrada/salida que sirvan como contrato. Estas prácticas fortalecen la capacidad de revisión y facilitan que otras personas validen el programa, reproduzcan resultados y detecten defectos de forma temprana [5], [7], [23].

Destacando la dimensión socio-técnica, la interacción con usuarios exige considerar usabilidad básica: mensajes, consistencia de opciones, prevención de errores y retroalimentación en tiempo de ejecución. En ejercicios introductorios, esta orientación puede incorporarse mediante requisitos simples de interacción y validación, promoviendo que el estudiante produzca programas que apoyen al usuario y reduzcan fricción operativa [1], [2], [26].

De manera complementaria, la documentación se apoya en representaciones estructuradas que conectan análisis y construcción. Modelos orientados a objetos y notaciones de diseño ofrecen un vocabulario compartido para describir entidades, responsabilidades

y relaciones, ayudando a que el código conserve coherencia con el diseño. Esta práctica prepara al estudiante para proyectos de mayor envergadura, donde la coordinación depende de artefactos que complementan el código [4], [16], [20].

1.3. Datos, información y conocimiento en Sistemas de Información

En Sistemas de Información (IS), la distinción entre datos, información y conocimiento estructura la forma en que las organizaciones operan, controlan procesos y orientan decisiones. Los sistemas computacionales gestionan registros, ejecutan transformaciones y producen resultados interpretables; sin embargo, esta secuencia solo adquiere sentido cuando el desarrollador comprende qué representa cada nivel y cómo se articulan entre sí [2], [16], [22]. Para quien aprende a programar, esta tríada constituye una guía conceptual que conecta estructuras de datos, algoritmos y salidas con impactos organizacionales concretos.

Dentro de este panorama, programar implica asumir responsabilidad sobre la calidad de los datos que ingresan, la coherencia de los procesos que los transforman y la utilidad de los resultados que se generan. Cada variable declarada, cada validación implementada y cada cálculo realizado influyen en la transición de dato a información y de información a conocimiento operativo. Esta relación se ha consolidado como fundamento en IS, donde la arquitectura de sistemas y el modelado de procesos se articulan con gestión de datos y análisis organizacional [1], [2], [27].

Aprender programación desde esta perspectiva transforma ejercicios técnicos en prácticas con sentido. El estudiante deja de manipular números o cadenas aisladas y comienza a interpretar registros, estados y eventos que describen actividades organizacionales. Esta orientación promueve un enfoque riguroso: cada decisión de diseño debe justificarse por su aporte a la coherencia informacional y a la trazabilidad del proceso [5], [7], [13].

1.3.1. Fundamentos conceptuales

La diferenciación conceptual entre dato, información y conocimiento organiza la reflexión sobre cómo operan los sistemas de información en distintos contextos organizacionales. Un sistema no agrega valor por almacenar registros de manera masiva; lo hace cuando transforma registros en resultados utilizables y alineados con decisiones y acciones. Esta transición ha sido abordada por la literatura de IS y de gestión de datos como una secuencia

que exige precisión técnica y claridad semántica [2], [8], [22].

Para entender esta secuencia en clave formativa, el estudiante debe identificar qué representa cada nivel en términos programáticos. El dato se vincula con estructuras elementales; la información surge del procesamiento controlado; el conocimiento emerge cuando los resultados se interpretan bajo reglas organizacionales. Programar, por tanto, no consiste en producir salidas arbitrarias, sino en diseñar transformaciones que respeten significado y contexto [1], [5], [27].

Asimismo, estos fundamentos orientan el aprendizaje hacia la precisión. Cuando un algoritmo procesa registros sin validación o sin definición clara de atributos, se degrada la calidad informacional. Este marco garantista exige que el estudiante adopte prácticas de validación y modelado coherentes con principios de ingeniería de software y administración de datos [4], [24], [28].

Dato como registro estructurado

El dato puede definirse como un registro estructurado que representa un hecho, evento o atributo dentro de un dominio organizacional. En IS, los datos adquieren significado cuando se asocian a entidades, relaciones y restricciones, lo cual exige definir tipos, formatos y dominios válidos para cada atributo [2], [27], [28]. Esta caracterización evita ambigüedades y facilita que el sistema procese registros de forma consistente.

Desde la perspectiva del aprendizaje de programación, el dato se materializa en variables, arreglos, registros o clases. Declarar correctamente un tipo, seleccionar una estructura adecuada y validar valores de entrada son decisiones que determinan la integridad del sistema. Por esa razón, los ejercicios iniciales deben insistir en rangos válidos, coherencia de formatos y prevención de valores fuera de dominio [5], [6], [7].

Paralelamente, la representación estructurada se relaciona con principios de normalización y modelado conceptual. Aunque la teoría detallada de bases de datos se abordará en otras unidades, resulta indispensable introducir la noción de que cada atributo responde a una definición precisa y a restricciones explícitas. Esta práctica favorece una visión holística donde el programador entiende que los datos constituyen la materia prima del sistema [27], [28], [29].

En síntesis, aprender a programar implica asumir que cada dato posee significado organizacional. No se trata de valores abstractos, sino de representaciones de hechos que deben mantenerse coherentes, completos y verificables a lo largo del ciclo de vida del sistema [2], [4], [22].

Información como procesamiento contextualizado

La información surge cuando los datos son procesados bajo reglas que incorporan contexto, relaciones y criterios de interpretación. En IS, esta transformación requiere operaciones de cálculo, agregación, filtrado y clasificación que dotan a los registros de significado operativo [2], [8], [22]. La diferencia entre un dato aislado y una información útil radica en el marco que orienta su procesamiento.

Para quien aprende a programar, esta transición se traduce en diseñar algoritmos que transformen entradas en resultados verificables. Calcular totales, identificar tendencias simples o validar condiciones son ejemplos de procesamiento contextualizado que enseñan a relacionar datos con reglas explícitas. Esta práctica promueve disciplina algorítmica y precisión en la definición de operaciones [5], [7], [13].

En contraste con el manejo superficial de registros, el procesamiento contextualizado exige que el estudiante justifique cada operación. ¿Por qué se filtra un conjunto? ¿Qué criterio define una categoría? Estas preguntas fortalecen el razonamiento estructurado y conectan programación con análisis organizacional [1], [2], [16].

De manera complementaria, la calidad de la información depende de la correcta implementación de validaciones y de la coherencia en los cálculos. Errores en tipos numéricos o en comparaciones pueden alterar indicadores y decisiones posteriores. Por ello, la enseñanza de programación debe integrar pruebas sistemáticas y revisión de resultados para garantizar consistencia [4], [11], [12].

Asimismo, la producción de información útil requiere claridad en la presentación. El formato de salida, la organización de resultados y la legibilidad influyen en la interpretación por parte de usuarios. Introducir estos criterios en ejercicios básicos fomenta sensibilidad hacia el impacto organizacional del código [1], [2], [26].

Conocimiento como soporte para decisiones organizacionales

El conocimiento organizacional emerge cuando la información se interpreta bajo objetivos estratégicos y reglas de negocio. En IS, los sistemas apoyan esta etapa proporcionando reportes, indicadores y alertas que orientan acciones y ajustes de procesos [2], [8], [22]. La generación de conocimiento exige coherencia entre cálculo, contexto y finalidad organizacional.

Desde el aprendizaje de programación, el paso hacia conocimiento se refleja en la capacidad de diseñar funciones que traduzcan información en criterios de decisión. Por ejemplo, un programa puede evaluar umbrales, clasificar estados o sugerir acciones en

función de reglas predefinidas. Esta práctica fortalece la habilidad de vincular estructuras lógicas con consecuencias operativas [5], [7], [13].

Destacando la dimensión estratégica, el conocimiento computacional requiere trazabilidad. Cada resultado debe poder vincularse con datos y reglas que lo originaron, evitando decisiones basadas en cálculos opacos. La ingeniería de software respalda esta necesidad mediante principios de diseño claro y documentación coherente [1], [4], [24].

Dentro de este panorama, aprender a programar con orientación a IS significa asumir que cada algoritmo puede influir en decisiones organizacionales. Esta responsabilidad impulsa al estudiante a diseñar soluciones robustas, verificables y alineadas con criterios explícitos [2], [14], [15].

1.3.2. Ciclo de vida del dato en entornos organizacionales

El ciclo de vida del dato describe la secuencia de etapas que atraviesa un registro desde su captura hasta su utilización estratégica. En IS, esta secuencia integra procesos técnicos y organizacionales que determinan calidad, accesibilidad y utilidad del dato a lo largo del tiempo [2], [28], [29].

Para el estudiante de programación, visualizar este ciclo permite ubicar su intervención en cada fase. Programar no se limita a procesar información existente; también implica diseñar mecanismos de captura, validación, almacenamiento y presentación coherentes con reglas y objetivos. Esta visión dinámica conecta fundamentos técnicos con impacto organizacional [1], [7], [16].

En este marco, adoptar prácticas responsables desde programación básica favorece sistemas más robustos en etapas posteriores. Validar entradas, estructurar datos con claridad y producir salidas interpretables constituyen hábitos que acompañan al desarrollador en proyectos de mayor envergadura [4], [6], [24].

Captura y validación de datos

La captura de datos inicia el ciclo informacional. En organizaciones, esta etapa puede involucrar formularios digitales, sensores o registros manuales digitalizados. La calidad del sistema depende de que los datos ingresen con coherencia y consistencia desde el primer momento [2], [27], [28].

Para aprender programación, la captura se traduce en leer entradas, verificar formatos y establecer condiciones mínimas de aceptación. Implementar validaciones evita que el

sistema procese información defectuosa y enseña disciplina en el diseño de algoritmos [5], [6], [7].

Paralelamente, la validación debe contemplar rangos, tipos y relaciones entre atributos. Un registro puede ser válido en forma individual, aunque inconsistente respecto a otros valores. Introducir estas verificaciones en ejercicios básicos fortalece el razonamiento lógico y la atención a detalles estructurales [4], [13], [24].

Asimismo, la captura responsable promueve trazabilidad. Registrar fecha, origen o responsable de un dato facilita auditoría y control posterior, aspectos relevantes en distintos contextos organizacionales [1], [2], [28].

Almacenamiento y procesamiento informacional

El almacenamiento organiza datos para su recuperación y análisis posterior. Sistemas de bases de datos proporcionan mecanismos de persistencia, integridad y consulta que permiten mantener coherencia a lo largo del tiempo [27], [28], [29]. La programación interactúa con estas estructuras mediante consultas, inserciones y actualizaciones controladas.

Para el estudiante, esta etapa se vincula con estructuras de datos persistentes y con operaciones que transforman conjuntos de registros. Diseñar algoritmos que recorran colecciones, filtren elementos y produzcan resultados agregados constituye un paso formativo hacia sistemas informacionales más complejos [5], [7], [13].

Destacando la importancia de integridad, el almacenamiento debe preservar restricciones definidas en el modelo conceptual. Claves, relaciones y dominios protegen coherencia del sistema y evitan inconsistencias acumulativas [4], [27], [28].

En síntesis, aprender a programar implica reconocer que el procesamiento no ocurre en el vacío: se apoya en estructuras persistentes que deben diseñarse con rigor y mantenerse coherentes a lo largo del ciclo de vida [2], [16], [24].

Presentación, análisis y uso estratégico

La etapa final del ciclo informacional consiste en presentar resultados de forma que apoyen análisis y acción. Reportes, visualizaciones y alertas traducen cálculos en representaciones interpretables por actores organizacionales [2], [8], [22]. La forma de presentación influye en la interpretación y en la calidad de decisiones derivadas.

Para quien aprende programación, esta fase introduce diseño de salidas legibles y estructuradas. Organizar información, etiquetar resultados y seleccionar formatos adecuados

forman parte del proceso algorítmico. Esta práctica fortalece sensibilidad hacia usuarios y hacia el impacto organizacional del código [1], [7], [26].

Asimismo, el análisis estratégico puede incorporar reglas adicionales que identifiquen patrones o condiciones de alerta. Implementar comparaciones, umbrales y clasificaciones enseña a vincular información con criterios decisionales [2], [5], [13].

Dentro de este panorama, aprender a programar desde la perspectiva de IS significa participar en todo el ciclo de vida del dato. Cada línea de código contribuye a capturar, transformar y presentar información que orienta acciones organizacionales. Esta visión integradora motiva al lector a desarrollar competencias técnicas con conciencia del impacto y con compromiso hacia la calidad del sistema [4], [14], [15].

1.4. Tecnología computacional como soporte del Sistema de Información

La tecnología computacional constituye el entramado material y lógico que posibilita la operación de un Sistema de Información (SI). En organizaciones contemporáneas, hardware, software y componentes de control interactúan para capturar eventos, ejecutar reglas y producir resultados coherentes con objetivos operativos y estratégicos [1], [2], [16]. Desde la perspectiva formativa, este entramado adquiere sentido cuando el estudiante reconoce que cada programa escrito se ejecuta sobre una arquitectura concreta y que sus decisiones de diseño dialogan con restricciones técnicas reales.

Aprender a programar implica asumir que el código no existe en abstracto: se ejecuta en procesadores con capacidades definidas, interactúa con sistemas operativos que gestionan recursos y puede integrarse con dispositivos físicos que generan datos. Esta conciencia tecnológica orienta al lector hacia una práctica responsable, donde el rendimiento, la consistencia y la interacción con el entorno se consideran desde etapas iniciales del aprendizaje [5], [7], [10].

Dentro de este panorama, la comprensión del soporte tecnológico fortalece la calidad del desarrollo. Cuando el estudiante identifica cómo fluyen los datos desde un dispositivo hasta una aplicación y cómo se gestionan memoria y almacenamiento, adquiere una visión holística del sistema. Esta visión favorece decisiones algorítmicas coherentes con la infraestructura y evita supuestos infundados sobre disponibilidad de recursos [2], [4], [34].

Desde un enfoque riguroso, la tecnología computacional se convierte en aliada del aprendizaje. El conocimiento de sus componentes no persigue erudición técnica aislada;

proporciona criterios para diseñar programas robustos, prever limitaciones y articular soluciones alineadas con la realidad organizacional [1], [16], [23].

1.4.1. Componentes tecnológicos

Los componentes tecnológicos de un SI pueden agruparse en infraestructura de hardware, software de aplicación y sistemas operativos, así como firmware encargado del control directo de dispositivos. Cada componente cumple funciones específicas que, integradas, permiten capturar, procesar y distribuir información de manera coherente [2], [10], [34].

Para entender la relevancia de estos elementos en el aprendizaje de programación, es preciso reconocer que cada línea de código interactúa con ellos. Un programa gestiona memoria proporcionada por el sistema operativo, utiliza capacidades de procesamiento del hardware y puede depender de controladores que median la comunicación con dispositivos externos. Esta interdependencia sugiere que la formación del desarrollador en IS debe incorporar nociones básicas sobre arquitectura y gestión de recursos [4], [7], [10].

Asimismo, la articulación entre componentes tecnológicos respalda la confiabilidad del sistema. Un algoritmo correcto puede degradar su desempeño cuando ignora limitaciones de hardware o comportamientos del sistema operativo. Por ello, la programación con orientación a IS exige considerar latencia, almacenamiento y concurrencia como variables que influyen en la ejecución [23], [24], [34].

Infraestructura de hardware

La infraestructura de hardware comprende procesadores, memoria principal, almacenamiento secundario y dispositivos periféricos que permiten entrada y salida de información. Estos elementos determinan capacidad de cómputo, velocidad de respuesta y volumen de datos que el sistema puede manejar [2], [34]. En organizaciones con operaciones intensivas, la selección y configuración del hardware impactan directamente en la eficiencia del SI.

Desde la perspectiva del aprendizaje, comprender nociones básicas de arquitectura favorece decisiones algorítmicas coherentes. La elección de estructuras de datos, el uso de ciclos anidados o la gestión de colecciones influyen en consumo de memoria y tiempo de ejecución. Esta relación introduce al estudiante en el análisis de eficiencia y en la importancia de evaluar costos computacionales [5], [7], [13].

De manera complementaria, el hardware interactúa con dispositivos que generan datos organizacionales: lectores, sensores o terminales de usuario. Programar implica considerar formatos de entrada y latencias asociadas a estos dispositivos, preparando al lector para

entornos donde la tecnología trasciende la pantalla y se integra con procesos físicos [1], [2], [34].

Software de aplicación y sistemas operativos

El software de aplicación implementa reglas de negocio y funcionalidades específicas del SI, mientras que el sistema operativo administra recursos, procesos y comunicación entre componentes. Esta distinción organiza la arquitectura lógica del entorno donde se ejecuta el código [2], [10], [24].

Para quien aprende a programar, entender esta separación favorece claridad conceptual. El código desarrollado en un lenguaje de alto nivel depende de servicios del sistema operativo para acceder a archivos, gestionar memoria o interactuar con red. Esta dependencia exige respetar convenciones, manejar excepciones y estructurar programas conforme a estándares del entorno [5], [7], [10].

Paralelamente, el software de aplicación refleja decisiones organizacionales. Cada módulo implementa funciones alineadas con procesos y requisitos definidos. Diseñar programas coherentes con estos requisitos fortalece la competencia del desarrollador y lo prepara para integrar su trabajo en sistemas de mayor envergadura [1], [16], [23].

Asimismo, la interacción entre aplicaciones y sistema operativo introduce al estudiante en conceptos como concurrencia y gestión de recursos compartidos. Aunque su estudio detallado corresponde a etapas posteriores, una noción inicial sobre cómo el entorno gestiona procesos enriquece la práctica de programación básica [10], [24], [34].

Firmware y control de dispositivos

El firmware constituye el nivel intermedio entre hardware y software de aplicación. Se trata de programas almacenados en dispositivos que controlan operaciones básicas y permiten comunicación con sistemas superiores [2], [34]. En entornos organizacionales con dispositivos inteligentes, el firmware coordina captura y transmisión de datos hacia aplicaciones centrales.

Aprender programación con conciencia de este nivel amplía la visión del estudiante. La interacción con dispositivos exige considerar protocolos, formatos y validaciones adicionales. Aunque el desarrollo de firmware puede exceder la programación inicial, reconocer su existencia fortalece la comprensión de la cadena completa de procesamiento [4], [16], [24].

En síntesis, el firmware ilustra que la programación no se limita a aplicaciones visibles; integra capas que permiten funcionamiento coordinado del sistema. Esta comprensión

prepara al lector para escenarios donde software y hardware interactúan de manera dinámica y organizada [1], [2], [34].

1.4.2. Modelo funcional de entrada–proceso–salida en SI

El modelo funcional de entrada–proceso–salida (E–P–S) describe la estructura básica de operación de un SI. Datos ingresan al sistema, son transformados mediante reglas y algoritmos, y producen resultados destinados a usuarios o a otros sistemas [1], [2]. Este esquema sintetiza la lógica fundamental que el estudiante reproduce en cada ejercicio de programación.

Desde una perspectiva pedagógica, el modelo E–P–S actúa como guía para estructurar programas. Identificar claramente entradas, diseñar procesamiento coherente y definir salidas verificables ordena el razonamiento y facilita pruebas sistemáticas. Esta estructura fomenta disciplina y claridad en el diseño algorítmico [5], [7], [13].

Asimismo, el modelo evidencia que cada componente tecnológico participa en la secuencia. El hardware recibe datos, el software procesa información y los dispositivos presentan resultados. Esta integración subraya la relevancia de comprender cómo interactúan los niveles tecnológicos descritos anteriormente [2], [10], [34].

Entrada de datos organizacionales

La entrada de datos constituye el punto inicial del modelo E–P–S. Formularios, interfaces digitales y dispositivos de captura registran eventos y atributos relevantes para la operación organizacional [1], [2]. La calidad de esta etapa condiciona el procesamiento posterior.

Para quien aprende a programar, diseñar entradas implica validar tipos, rangos y formatos antes de ejecutar cálculos. Implementar estas verificaciones refuerza disciplina y reduce propagación de errores [5], [6], [7].

Asimismo, la entrada puede provenir de sistemas externos o de dispositivos automáticos. Integrar estas fuentes exige comprender protocolos básicos y estructuras de datos compatibles [2], [10], [34].

Destacando la dimensión organizacional, cada dato capturado representa un evento con significado. Reconocer esta conexión motiva al estudiante a programar con responsabilidad y coherencia [1], [4], [15].

Procesamiento de información

El procesamiento transforma entradas en resultados mediante algoritmos estructurados. Operaciones aritméticas, comparaciones, iteraciones y funciones constituyen la base de esta transformación [5], [7], [13]. En IS, este procesamiento implementa reglas de negocio y políticas operativas.

Aprender a diseñar algoritmos eficientes implica analizar pasos, evaluar complejidad y verificar consistencia de resultados. Esta práctica fortalece pensamiento lógico y prepara al lector para problemas de mayor escala [4], [23], [24].

De manera complementaria, el procesamiento puede involucrar interacción con almacenamiento persistente o con sistemas externos. Introducir estas consideraciones en ejercicios graduales permite visualizar la amplitud del entorno tecnológico [16], [28], [29].

Salida para usuarios y sistemas

La salida representa el resultado visible del modelo E–P–S. Puede consistir en mensajes en pantalla, reportes estructurados o datos transmitidos a otros sistemas [1], [2]. Su claridad y coherencia determinan utilidad organizacional.

Para el estudiante, diseñar salidas legibles y consistentes forma parte integral del algoritmo. Organizar resultados y explicar estados refuerza la conexión entre código y usuario [5], [7], [26].

Asimismo, la salida puede alimentar procesos posteriores. Programar con esta visión holística fomenta responsabilidad técnica y coherencia sistémica [4], [16], [24].

1.5. Representación digital de la información

La representación digital define la forma en que un Sistema de Información (SI) convierte hechos organizacionales en símbolos manipulables por software. Esta conversión condiciona almacenamiento, transmisión y procesamiento, dado que toda operación computacional se ejecuta sobre codificaciones concretas que deben preservar significado y consistencia [1], [2], [34].

Aprender a programar con orientación a IS se fortalece cuando el lector domina las decisiones elementales de representación. Un error de codificación de texto, un desbordamiento en enteros o una interpretación incorrecta de números reales puede degradar reportes, alterar reglas de negocio y producir decisiones apoyadas en resultados inválidos [7], [11], [12].

Dentro de este panorama, la representación digital aporta un marco formativo que conecta teoría y práctica. Comprender bases numéricas, codificaciones y formatos conduce a programas más robustos, con validaciones mejor justificadas y salidas que preservan semántica organizacional [4], [5], [13].

Esta sección organiza la representación digital en tres perspectivas complementarias: fundamentos binarios, codificación de tipos de contenido y riesgos habituales de representación. La finalidad de ilustrar esta organización consiste en que el lector encuentre motivos técnicos y profesionales para programar con rigor, desde ejercicios básicos hasta escenarios organizacionales de mayor envergadura [2], [16], [24].

1.5.1. Fundamentos de representación binaria

La representación binaria constituye el lenguaje operativo del hardware. El procesador y la memoria trabajan con estados discretos, y el software se apoya en esos estados para representar números, texto y estructuras de datos que describen eventos organizacionales [2], [10], [34].

Aprender estos fundamentos permite que la programación deje de ser una actividad "mágica" y se convierta en una práctica justificable. Cuando el lector relaciona variables y estructuras con bytes reales en memoria, adquiere criterio para elegir tipos, anticipar límites y explicar fallas asociadas a rangos y conversiones [5], [7], [35].

La perspectiva binaria también facilita evaluar eficiencia. Representar información con estructuras apropiadas reduce consumo de memoria y costo de procesamiento, lo cual se refleja en tiempos de respuesta y escalabilidad del SI, especialmente cuando se procesan grandes volúmenes de registros [13], [34], [36].

No obstante, la utilidad formativa más directa se manifiesta en la depuración. Reconocer cómo se codifica un dato habilita rastrear errores de interpretación, fallos de lectura/escritura y resultados inesperados en operaciones aritméticas o lógicas, fortaleciendo hábitos de verificación [4], [7], [23].

Paralelamente, el manejo de bits introduce destrezas prácticas frecuentes en IS: validación mediante máscaras, compresión simple, control de permisos y codificación de estados. Estas destrezas ofrecen ejercicios motivadores para aprender a programar con problemas cercanos a escenarios reales [5], [13], [37].

Bits, bytes y almacenamiento

Un bit representa la unidad mínima de información en sistemas digitales, y un byte agrupa bits para formar una unidad práctica de almacenamiento. Esta organización responde a cómo hardware y sistema operativo direccionan memoria y gestionan transferencia de datos, lo cual incide en tamaño de variables, estructuras y archivos [2], [10], [34].

En programación, la noción de byte se conecta con el tipo de dato elegido. Un entero, un carácter o un valor booleano ocupan un número determinado de bytes; esa ocupación define rangos posibles y condiciona operaciones de lectura, escritura y serialización. Esta relación orienta al estudiante a seleccionar tipos con criterio y a justificar la representación en función del dominio organizacional [5], [7], [35].

El almacenamiento también se expresa en estructuras: arreglos, cadenas y registros ocupan memoria de manera distinta. Reconocer esta diferencia favorece decisiones sobre cómo modelar colecciones de datos organizacionales, evitando estructuras infladas o diseños que dificulten procesamiento por recorridos repetidos [13], [36], [37].

En escenarios de IS, el almacenamiento se extiende a archivos y bases de datos. Aunque el estudio detallado de persistencia se desarrolla en otras unidades, resulta pertinente observar que un archivo codifica bytes y que una mala interpretación de esa codificación produce datos corruptos o inconsistentes [27], [28], [29].

De manera complementaria, el almacenamiento se relaciona con integridad y seguridad. Tamaños mal controlados, conversiones inseguras o supuestos erróneos sobre longitudes introducen fallas previsibles, por lo que se recomienda entrenar desde temprano validaciones y manejo seguro de entradas [4], [6], [23].

Sistemas de numeración

Los sistemas de numeración constituyen el fundamento matemático que permite representar cantidades mediante símbolos y reglas formales. En términos generales, un sistema de numeración define un conjunto de dígitos permitidos y un mecanismo para combinar dichos dígitos con el fin de expresar valores. Desde la perspectiva computacional, esta formalización es determinante porque toda información procesada por un Sistema de Información (SI) se expresa finalmente como secuencias numéricas codificadas en binario [13], [34], [37].

Existen sistemas posicionales y no posicionales. En los sistemas no posicionales, el valor de un símbolo no depende de su posición; ejemplos históricos como la numeración romana ilustran esta característica. En contraste, los sistemas posicionales asignan significado a cada posición en función de una potencia de la base. Esta distinción resulta indispensable

para quien aprende a programar, ya que los sistemas digitales utilizan exclusivamente esquemas posicionales para almacenar y transformar datos [5], [7], [34].

En un sistema posicional de base b, los dígitos permitidos pertenecen al conjunto $\{0, 1, \ldots, b-1\}$. El valor mínimo representable con un solo dígito es 0, y el valor máximo es $b-1$. Cuando se combinan n dígitos, el valor total se obtiene mediante la suma ponderada de cada dígito multiplicado por la potencia correspondiente de la base:

$$N = \sum_{i=0}^{n-1} d_i \cdot b^i,$$

donde d_i representa el dígito en la posición i. Esta formulación no constituye un formalismo abstracto; proporciona al estudiante un modelo exacto para diseñar algoritmos de conversión y para justificar resultados obtenidos por el programa [5], [13], [37].

En computación, base 2 se emplea para operación interna, base 16 se utiliza para representación compacta y lectura técnica, y base 10 se mantiene como referencia organizacional para usuarios y reportes [2], [7], [34]. Dominar la noción de base permite al lector interpretar resultados de depuración, comprender límites de tipos numéricos y diseñar validaciones coherentes con el dominio del problema.

Representación en base decimal (10). El sistema decimal utiliza base 10, con dígitos entre 0 y 9. Su valor máximo por posición es 9, y cada posición representa una potencia de 10. Por ejemplo:

$$372_{10} = 3 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0.$$

En programación, esta representación coincide con la forma habitual en que los usuarios ingresan datos numéricos, lo cual exige validaciones de rango y control de formato [5], [7].

Ejercicios propuestos:

1. Representar el número 5042_{10} como suma de potencias de 10.

2. Determinar el valor máximo representable con tres dígitos decimales.

3. Codificar en decimal el número ASCII del carácter ‘A‘ (65) y verificar su interpretación como carácter.

Representación en base binaria (2). La base 2 emplea dígitos $\{0, 1\}$. El valor máximo por posición es 1. Cada posición representa una potencia de 2. Por ejemplo:

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}.$$

Este sistema se ha consolidado como fundamento del hardware digital [7], [13], [34].

Ejercicios propuestos:

1. Convertir 110101_2 a decimal.
2. Determinar el valor máximo representable con 8 bits sin signo.
3. Representar en binario el código ASCII de la letra 'B' (66).

Representación en base octal (8). El sistema octal utiliza base 8, con dígitos entre 0 y 7. El valor máximo por posición es 7. Su uso en informática histórica y en permisos de sistemas operativos lo convierte en un recurso práctico para quien aprende a programar [10], [34], [37]. Un número octal como:

$$157_8 = 1 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0$$

puede convertirse a decimal mediante el algoritmo posicional descrito anteriormente.

Ejercicios propuestos:

1. Convertir 345_8 a base decimal.
2. Determinar el mayor número representable con cuatro dígitos octales.
3. Representar en octal el código decimal 65 y relacionarlo con su carácter ASCII correspondiente.

Representación en base hexadecimal (16). La base 16 utiliza dígitos $\{0, \ldots, 9, A, \ldots, F\}$, donde $A = 10$ hasta $F = 15$. El valor máximo por posición es 15. Esta base facilita lectura compacta de direcciones de memoria y datos binarios [7], [34], [35]. Por ejemplo:

$$2F_{16} = 2 \cdot 16^1 + 15 \cdot 16^0 = 47_{10}.$$

Ejercicios propuestos:

1. Convertir $3A7_{16}$ a decimal.
2. Determinar el máximo valor representable con dos dígitos hexadecimales.
3. Representar en hexadecimal el código ASCII de la palabra "SI" (S: 83, I: 73).

Aplicación formativa. El estudio de los sistemas de numeración fortalece pensamiento algorítmico, ya que cada conversión puede implementarse mediante procedimientos iterativos y verificables [5], [7], [13]. La conexión con datos alfanuméricos se evidencia en la codificación ASCII o Unicode, donde cada carácter se representa como número entero que puede expresarse en distintas bases.

Desde la perspectiva de IS, dominar estas representaciones habilita interpretar identificadores, permisos, códigos de estado y estructuras de almacenamiento. Programar con criterio numérico reduce errores de conversión, facilita depuración y aporta fundamento técnico a decisiones de diseño que impactan directamente en la calidad del Sistema de Información [2], [24], [34].

Conversión de valores enteros entre distintos sistemas de numeración

La conversión entre bases se apoya en descomposición posicional. Este procedimiento constituye un ejemplo didáctico de algoritmo finito, con pasos verificables y resultados contrastables, por lo que se ha posicionado como práctica formativa adecuada para principiantes [5], [7], [13].

En programación, implementar conversiones refuerza dominio de divisiones enteras, residuos, recorridos y construcción incremental de resultados. Estos conceptos reaparecen en problemas típicos de IS, tales como generación de códigos, validación de identificadores y tratamiento de formatos de entrada variados [5], [7], [37].

Las conversiones entre bases se organizan en familias de procedimientos según su lógica de cálculo. En términos formativos, esta agrupación ayuda a programar las transformaciones con algoritmos simples, verificables y reutilizables, evitando memorizar reglas aisladas [5], [7], [13].

Familia A: conversión *desde decimal* mediante divisiones sucesivas Este procedimiento se aplica cuando el número de entrada está en base 10 y se desea expresarlo en base b (por ejemplo, decimal→binario, decimal→octal, decimal→hexadecimal). Su lógica se apoya en cocientes y residuos, lo cual facilita una implementación directa con bucles [5], [7], [13].

1. Tomar el número decimal N como valor inicial.
2. Dividir N entre la base destino b.
3. Registrar el residuo $r = N$ mód b como dígito siguiente.
4. Actualizar $N \leftarrow \lfloor N/b \rfloor$.
5. Repetir pasos 2–4 hasta que $N = 0$.
6. Leer los residuos en orden inverso (del último al primero) para obtener el número en base b.
7. En base 16, reemplazar residuos 10–15 por A–F.

Casos cubiertos por esta familia:

- **Decimal → Binario** (base $b = 2$)
- **Decimal → Octal** (base $b = 8$)
- **Decimal → Hexadecimal** (base $b = 16$)

Ejercicios resueltos: Conversión desde decimal mediante divisiones sucesivas

A continuación se presentan dos ejercicios completos para cada caso cubierto por la familia de conversión desde base decimal hacia otra base utilizando divisiones sucesivas. En cada uno se siguen estrictamente los pasos establecidos.

1. Decimal → Binario (base $b = 2$)

Ejercicio 1: Convertir 45_{10} a binario
Paso 1: $N = 45$

N	$N \div 2$	Residuo
45	22	1
22	11	0
11	5	1
5	2	1
2	1	0
1	0	1

Paso final: Leer residuos en orden inverso:

$$45_{10} = 101101_2$$

Ejercicio 2: Convertir 156_{10} a binario

N	$N \div 2$	Residuo
156	78	0
78	39	0
39	19	1
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

$$156_{10} = 10011100_2$$

2. Decimal $\rightarrow$ Octal (base $b = 8$)

Ejercicio 1: Convertir 83_{10} a octal

N	$N \div 8$	Residuo
83	10	3
10	1	2
1	0	1

$$83_{10} = 123_8$$

Ejercicio 2: Convertir 250_{10} a octal

N	$N \div 8$	Residuo
250	31	2
31	3	7
3	0	3

$$250_{10} = 372_8$$

3. Decimal → Hexadecimal (base $b = 16$)

Ejercicio 1: Convertir 254_{10} a hexadecimal

N	$N \div 16$	Residuo
254	15	14
15	0	15

Reemplazando residuos:

$$14 = E, \quad 15 = F$$

$$254_{10} = FE_{16}$$

Ejercicio 2: Convertir 1023_{10} a hexadecimal

N	$N \div 16$	Residuo
1023	63	15
63	3	15
3	0	3

Reemplazando residuos:

$$15 = F$$

$$1023_{10} = 3FF_{16}$$

Familia B: conversión *hacia decimal* mediante expansión posicional Este procedimiento se aplica cuando el número de entrada está en base b y se desea su equivalente en base 10 (por ejemplo, binario→decimal, octal→decimal, hexadecimal→decimal). Se fundamenta en el sistema posicional, donde cada dígito pondera una potencia de la base [13], [34], [37].

La conversión también funciona como ejercicio de pruebas. El estudiante puede verificar propiedades de reversibilidad (convertir y retornar), detectar discrepancias por redondeo y documentar supuestos sobre rangos. Esta práctica fortalece hábitos de verificación y depuración [4], [5], [23].

1. Identificar la base b del número de entrada (2, 8 o 16).

2. Enumerar los dígitos de derecha a izquierda, asignando exponente 0 al dígito menos significativo.

3. Para cada dígito d_i, calcular $d_i \cdot b^i$.

4. Sumar todos los productos obtenidos.

5. El resultado total corresponde al valor en base decimal.

Casos cubiertos por esta familia:

- **Binario → Decimal** (base $b = 2$)
- **Octal → Decimal** (base $b = 8$)
- **Hexadecimal → Decimal** (base $b = 16$, con $A = 10, \ldots, F = 15$)

Ejercicios resueltos: Conversión hacia decimal mediante expansión posicional

A continuación se presentan dos ejercicios completos para cada caso cubierto por la familia de conversión hacia base decimal utilizando expansión posicional. En cada ejercicio se siguen estrictamente los pasos establecidos: identificar la base, enumerar dígitos de derecha a izquierda, multiplicar por potencias de la base y sumar los productos.

1. Binario → Decimal (base $b = 2$)

Ejercicio 1: Convertir 101101_2 a decimal
Paso 1: Base identificada: $b = 2$
Paso 2: Enumerar dígitos de derecha a izquierda (exponente 0 al menos significativo)

Dígito (d_i)	Posición i	Potencia 2^i	Producto $d_i \cdot 2^i$
1	0	$2^0 = 1$	$1 \cdot 1 = 1$
0	1	$2^1 = 2$	$0 \cdot 2 = 0$
1	2	$2^2 = 4$	$1 \cdot 4 = 4$
1	3	$2^3 = 8$	$1 \cdot 8 = 8$
0	4	$2^4 = 16$	$0 \cdot 16 = 0$
1	5	$2^5 = 32$	$1 \cdot 32 = 32$

Paso final: Sumar productos:

$$1 + 0 + 4 + 8 + 0 + 32 = 45$$

$$101101_2 = 45_{10}$$

Ejercicio 2: Convertir 10011100_2 a decimal
Paso 1: Base identificada: $b = 2$

Dígito (d_i)	Posición i	Potencia 2^i	Producto $d_i \cdot 2^i$
0	0	$2^0 = 1$	$0 \cdot 1 = 0$
0	1	$2^1 = 2$	$0 \cdot 2 = 0$
1	2	$2^2 = 4$	$1 \cdot 4 = 4$
1	3	$2^3 = 8$	$1 \cdot 8 = 8$
1	4	$2^4 = 16$	$1 \cdot 16 = 16$
0	5	$2^5 = 32$	$0 \cdot 32 = 0$
0	6	$2^6 = 64$	$0 \cdot 64 = 0$
1	7	$2^7 = 128$	$1 \cdot 128 = 128$

Paso final: Sumar productos:

$$0 + 0 + 4 + 8 + 16 + 0 + 0 + 128 = 156$$

$$10011100_2 = 156_{10}$$

2. Octal → Decimal (base $b = 8$)

Ejercicio 1: Convertir 372_8 a decimal
Paso 1: Base identificada: $b = 8$

Dígito (d_i)	Posición i	Potencia 8^i	Producto $d_i \cdot 8^i$
2	0	$8^0 = 1$	$2 \cdot 1 = 2$
7	1	$8^1 = 8$	$7 \cdot 8 = 56$
3	2	$8^2 = 64$	$3 \cdot 64 = 192$

Paso final: Sumar productos:

$$2 + 56 + 192 = 250$$

$$372_8 = 250_{10}$$

Ejercicio 2: Convertir 123_8 a decimal

Paso 1: Base identificada: $b = 8$

Dígito (d_i)	Posición i	Potencia 8^i	Producto $d_i \cdot 8^i$
3	0	$8^0 = 1$	$3 \cdot 1 = 3$
2	1	$8^1 = 8$	$2 \cdot 8 = 16$
1	2	$8^2 = 64$	$1 \cdot 64 = 64$

Paso final: Sumar productos:

$$3 + 16 + 64 = 83$$

$$123_8 = 83_{10}$$

3. Hexadecimal → Decimal (base $b = 16$)

Ejercicio 1: Convertir FE_{16} a decimal

Paso 1: Base identificada: $b = 16$, con $F = 15$ y $E = 14$

Dígito (d_i)	Posición i	Potencia 16^i	Producto $d_i \cdot 16^i$
$E = 14$	0	$16^0 = 1$	$14 \cdot 1 = 14$
$F = 15$	1	$16^1 = 16$	$15 \cdot 16 = 240$

Paso final: Sumar productos:

$$14 + 240 = 254$$

$$FE_{16} = 254_{10}$$

Ejercicio 2: Convertir $3FF_{16}$ a decimal

Paso 1: Base identificada: $b = 16$, con $F = 15$

Dígito (d_i)	Posición i	Potencia 16^i	Producto $d_i \cdot 16^i$
$F = 15$	0	$16^0 = 1$	$15 \cdot 1 = 15$
$F = 15$	1	$16^1 = 16$	$15 \cdot 16 = 240$
3	2	$16^2 = 256$	$3 \cdot 256 = 768$

Paso final: Sumar productos:

$$15 + 240 + 768 = 1023$$

$$3FF_{16} = 1023_{10}$$

Familia C: conversiones entre bases potencia de 2 mediante agrupación de bits Cuando las bases son potencias de 2, las conversiones pueden realizarse por *agrupación de bits* sin pasar por decimal, lo cual simplifica depuración y permite implementar conversores eficientes [13], [34], [35]. En particular, $8 = 2^3$ y $16 = 2^4$.

C1. Binario $\leftrightarrow$ Octal (grupos de 3 bits) Binario $\rightarrow$ Octal

1. Separar el número binario en grupos de 3 bits desde la derecha.
2. Si el grupo más a la izquierda queda incompleto, rellenar con ceros a la izquierda hasta completar 3 bits.
3. Convertir cada grupo de 3 bits a su valor decimal (0 a 7).
4. Escribir esos valores consecutivamente como dígitos octales.

Octal $\rightarrow$ Binario

1. Tomar cada dígito octal por separado (0 a 7).
2. Convertir cada dígito a su forma binaria de 3 bits (por ejemplo, $5_8 \rightarrow 101_2$).
3. Concatenar los grupos de 3 bits en el mismo orden para formar el binario final.
4. Si se desea una forma normalizada, eliminar ceros a la izquierda que no aporten valor.

C2. Binario $\leftrightarrow$ Hexadecimal (grupos de 4 bits) Binario $\rightarrow$ Hexadecimal

1. Separar el número binario en grupos de 4 bits desde la derecha.
2. Rellenar con ceros a la izquierda el grupo más significativo si fuera necesario.
3. Convertir cada grupo de 4 bits a su valor (0 a 15).

4. Reemplazar 10–15 por A–F.

5. Concatenar los dígitos resultantes para obtener el número hexadecimal.

Hexadecimal → Binario

1. Tomar cada dígito hexadecimal por separado (0–9, A–F).

2. Convertirlo a su valor entero (A=10, ..., F=15).

3. Representar ese valor como binario de 4 bits.

4. Concatenar los grupos de 4 bits en el mismo orden.

5. Normalizar quitando ceros a la izquierda si corresponde.

Ejercicios resueltos: Conversión entre bases potencia de 2 con longitudes no divisibles

En los siguientes ejercicios se emplean cantidades de bits que no son múltiplos de 3 ni de 4 (por ejemplo, 8, 16 y 32 bits). Esto obliga a aplicar correctamente el paso de relleno con ceros a la izquierda antes de agrupar, lo cual es relevante cuando se trabaja con registros completos de memoria o tipos de datos estándar en programación (byte de 8 bits, palabra de 16 bits, entero de 32 bits).

1. Binario → Octal (8, 16 y 32 bits)

Ejercicio 1: Convertir 10110110_2 **(8 bits) a octal**
Paso 1: Agrupar desde la derecha en bloques de 3 bits:

$$10\ 110\ 110$$

Paso 2: Rellenar con ceros a la izquierda:

$$010\ 110\ 110$$

Paso 3: Convertir cada grupo:

$$010_2 = 2, \quad 110_2 = 6, \quad 110_2 = 6$$

Paso 4: Resultado:

$$10110110_2 = 266_8$$

Observación adicional: En programación, un byte (8 bits) no coincide naturalmente con bloques de 3 bits; por ello, el relleno garantiza que la representación octal conserve el valor exacto del patrón binario.

Ejercicio 2: Convertir 1100110010101101_2 **(16 bits) a octal**
Paso 1:

$$110\ 011\ 001\ 010\ 110\ 1$$

Paso 2: Rellenar:

$$001\ 100\ 110\ 010\ 101\ 101$$

Paso 3:

$$001 = 1,\ 100 = 4,\ 110 = 6,\ 010 = 2,\ 101 = 5,\ 101 = 5$$

Paso 4:

$$1100110010101101_2 = 146255_8$$

Observación adicional: En registros de 16 bits, la agrupación produce más dígitos octales que bytes disponibles, lo que demuestra que el tamaño de palabra y la base elegida influyen en la forma externa del número.

2. Octal → Binario (resultados de 8, 16 y 32 bits)

Ejercicio 1: Convertir 347_8 **a binario**
Paso 1:

$$3,\ 4,\ 7$$

Paso 2:

$$3 = 011,\ 4 = 100,\ 7 = 111$$

Paso 3:

$$011100111_2$$

Paso 4: Normalizar:

$$11100111_2$$

Observación adicional: El resultado tiene 8 bits tras eliminar el cero inicial, coincidiendo con un byte completo.

Ejercicio 2: Convertir 15723_8 a binario
Paso 1:

$$1,\ 5,\ 7,\ 2,\ 3$$

Paso 2:

$$001,\ 101,\ 111,\ 010,\ 011$$

Paso 3:

$$001101111010011_2$$

Paso 4: Normalizar:

$$1101111010011_2$$

Observación adicional: Este resultado no es múltiplo de 8 bits; al almacenarlo en memoria real, el compilador ajustaría el tamaño al tipo de dato correspondiente.

3. Binario → Hexadecimal (8, 16 y 32 bits)

Ejercicio 1: Convertir 10101101_2 (8 bits) a hexadecimal
Paso 1: Agrupar en bloques de 4 bits:

$$1010\ 1101$$

Paso 2: No requiere relleno.
Paso 3:

$$1010 = 10 = A, \quad 1101 = 13 = D$$

Paso 4:

$$10101101_2 = AD_{16}$$

Observación adicional: Un byte equivale exactamente a dos dígitos hexadecimales; por ello, hexadecimal resulta especialmente práctico para inspección de memoria.

Ejercicio 2: Convertir 1100101011110001_2 (16 bits) a hexadecimal
Paso 1:

$$1100\ 1010\ 1111\ 0001$$

Paso 2:

$$1100 = 12 = C,\ 1010 = 10 = A,\ 1111 = 15 = F,\ 0001 = 1$$

Paso 3:

$$1100101011110001_2 = CAF1_{16}$$

Observación adicional: En 16 bits se obtienen cuatro dígitos hexadecimales; esta relación directa simplifica la depuración en lenguajes de bajo nivel.

4. Hexadecimal → Binario (32 bits)

Ejercicio 1: Convertir $1A3F_{16}$ a binario
Paso 1:

$$1,\ A,\ 3,\ F$$

Paso 2:

$$0001,\ 1010,\ 0011,\ 1111$$

Paso 3:

$$0001101000111111_2$$

Paso 4: Normalizar:

$$1101000111111_2$$

Observación adicional: Aunque el número puede escribirse sin ceros iniciales, en memoria podría almacenarse en 16 o 32 bits según el tipo de dato.

Ejercicio 2: Convertir $89ABCDEF_{16}$ a binario (32 bits)
Paso 1:

$$8,\ 9,\ A,\ B,\ C,\ D,\ E,\ F$$

Paso 2:

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Paso 3:

$$10001001101010111100110111101111_2$$

Observación adicional: Cada dígito hexadecimal representa exactamente 4 bits; por ello, 8 dígitos hexadecimales equivalen a 32 bits completos, tamaño típico de un entero estándar en muchos lenguajes de programación.

Familia D: conversiones Octal ↔ Hexadecimal mediante puente binario Octal y hexadecimal no son potencias directas entre sí, aunque ambas son potencias de 2. Un procedimiento estable consiste en convertir primero a binario y luego a la base destino, manteniendo pasos sistemáticos y fáciles de programar [5], [7], [34].

D1. Octal → Hexadecimal

1. Convertir cada dígito octal a 3 bits y concatenar para obtener el binario (procedimiento C1).
2. Agrupar el binario en bloques de 4 bits desde la derecha (procedimiento C2).
3. Convertir cada bloque a hexadecimal y concatenar.

D2. Hexadecimal → Octal

1. Convertir cada dígito hexadecimal a 4 bits y concatenar para obtener el binario (procedimiento C2).
2. Agrupar el binario en bloques de 3 bits desde la derecha (procedimiento C1).
3. Convertir cada bloque a octal y concatenar.

Mapa completo de conversiones (con familia aplicable)

- Decimal → Binario: Familia A
- Binario → Decimal: Familia B
- Decimal → Octal: Familia A
- Octal → Decimal: Familia B
- Decimal → Hexadecimal: Familia A
- Hexadecimal → Decimal: Familia B
- Binario → Octal: Familia C1
- Octal → Binario: Familia C1
- Binario → Hexadecimal: Familia C2
- Hexadecimal → Binario: Familia C2
- Octal → Hexadecimal: Familia D (puente binario)
- Hexadecimal → Octal: Familia D (puente binario)

Ejercicios resueltos: Conversión Octal ↔ Hexadecimal mediante puente binario

A continuación se presentan dos ejercicios completos para cada caso de la Familia D. En cada uno se sigue estrictamente el procedimiento: conversión intermedia a binario, reagrupación y conversión final. Se emplean longitudes que no son múltiplos directos entre 3 y 4 para evidenciar la necesidad de relleno y normalización.

1. Octal → Hexadecimal (puente binario)

Ejercicio 1: Convertir 735_8 a hexadecimal
Paso 1: Convertir cada dígito octal a 3 bits

$$7 = 111, \quad 3 = 011, \quad 5 = 101$$

$$735_8 \rightarrow 111011101_2$$

Paso 2: Agrupar en bloques de 4 bits desde la derecha

$$1110\ 1110\ 1$$

Rellenar a la izquierda:

$$0001\ 1101\ 1101$$

Paso 3: Convertir cada bloque a hexadecimal

$$0001 = 1, \quad 1101 = D, \quad 1101 = D$$

$$735_8 = 1DD_{16}$$

Observación adicional: Aunque el número original tenía 9 bits intermedios, el relleno garantiza alineación correcta en bloques de 4 bits, lo cual es indispensable al programar conversores automáticos.

Ejercicio 2: Convertir 12647_8 a hexadecimal
Paso 1:

$$1 = 001, \quad 2 = 010, \quad 6 = 110, \quad 4 = 100, \quad 7 = 111$$

$$12647_8 \rightarrow 001010110100111_2$$

Paso 2: Agrupar en bloques de 4 bits

$$0010\ 1011\ 0100\ 111$$

Rellenar:

$$0001\ 0101\ 1010\ 0111$$

Paso 3:

$$0001 = 1,\ 0101 = 5,\ 1010 = A,\ 0111 = 7$$

$$12647_8 = 15A7_{16}$$

Observación adicional: Este procedimiento evidencia que la longitud binaria intermedia no coincide necesariamente con 8, 16 o 32 bits; la alineación depende del patrón generado por la base original.

2. Hexadecimal → Octal (puente binario)

Ejercicio 1: Convertir $2AF_{16}$ a octal
Paso 1: Convertir cada dígito hexadecimal a 4 bits

$$2 = 0010, \quad A = 1010, \quad F = 1111$$

$$2AF_{16} \rightarrow 001010101111_2$$

Paso 2: Agrupar en bloques de 3 bits desde la derecha

$$001\ 010\ 101\ 111$$

Paso 3: Convertir cada bloque a octal

$$001 = 1,\ 010 = 2,\ 101 = 5,\ 111 = 7$$

$$2AF_{16} = 1257_8$$

Observación adicional: Obsérvese que 3 dígitos hexadecimales (12 bits) producen exactamente 4 dígitos octales (también 12 bits), lo cual facilita validaciones cruzadas en programas.

Ejercicio 2: Convertir $4C3D_{16}$ a octal
Paso 1:

$$4 = 0100, \quad C = 1100, \quad 3 = 0011, \quad D = 1101$$

$$4C3D_{16} \rightarrow 0100110000111101_2$$

Paso 2: Agrupar en bloques de 3 bits

$$010\ 011\ 000\ 011\ 110\ 1$$

Rellenar:

$$001\ 001\ 100\ 001\ 111\ 101$$

Paso 3:

$$001 = 1,\ 001 = 1,\ 100 = 4,\ 001 = 1,\ 111 = 7,\ 101 = 5$$

$$4C3D_{16} = 114175_8$$

Observación adicional: En este caso, 16 bits generan 6 dígitos octales tras el relleno; esta diferencia estructural es relevante cuando se diseñan funciones de conversión en lenguajes que manipulan enteros de tamaño fijo. No obstante, la conversión presenta límites cuando intervienen números reales. En esos casos, la representación depende de estándares de punto flotante, donde ciertos valores no pueden representarse exactamente. Reconocer esta limitación evita errores interpretativos en cálculos financieros o métricas operativas [11], [12], [38].

Conversión y representación de valores reales en distintos sistemas de numeración

Hasta este punto, los procedimientos descritos se han aplicado principalmente a números enteros. No obstante, en SI gran parte de los datos procesados corresponden a cantidades reales: montos financieros, promedios, tasas, porcentajes y métricas continuas. La conversión y representación de estos valores requiere extender los pasos anteriores, incorporando

el tratamiento de la parte fraccionaria y considerando las limitaciones inherentes a la representación en punto flotante [11], [12], [38].

1. Extensión del modelo posicional a números reales. En un sistema posicional de base b, un número real puede expresarse como:

$$N = \sum_{i=-m}^{n} d_i \cdot b^i$$

donde los exponentes negativos representan posiciones fraccionarias. Por ejemplo, en base decimal:

$$45.375_{10} = 4 \cdot 10^1 + 5 \cdot 10^0 + 3 \cdot 10^{-1} + 7 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

Esta formulación generaliza el esquema utilizado para enteros e introduce un aspecto formativo clave: la parte fraccionaria se construye mediante potencias negativas de la base [5], [13], [37]. Para quien aprende a programar, este modelo permite diseñar algoritmos sistemáticos de conversión que tratan por separado parte entera y parte decimal.

2. Conversión de decimal real a otra base. Cuando se convierte un número decimal real a base b (por ejemplo, decimal $\rightarrow$ binario), el procedimiento se divide en dos fases:
Parte entera (igual que antes):

1. Aplicar divisiones sucesivas entre la base b.
2. Registrar residuos.
3. Leer residuos en orden inverso.

Parte fraccionaria:

1. Tomar la parte decimal f.
2. Multiplicar f por la base b.
3. El entero resultante constituye el siguiente dígito fraccionario.
4. Conservar la nueva parte fraccionaria.
5. Repetir hasta que la fracción sea 0 o hasta alcanzar la precisión deseada.

Ejemplo conceptual: convertir 0.625_{10} a binario.

$$0.625 \times 2 = 1.25 \Rightarrow 1$$

$$0.25 \times 2 = 0.5 \Rightarrow 0$$

$$0.5 \times 2 = 1.0 \Rightarrow 1$$

Resultado: 0.101_2

Este procedimiento enseña al estudiante a manejar bucles controlados por precisión, introduciendo nociones de tolerancia y límite de iteraciones [5], [7], [38].

Ejemplos resueltos: Conversión de números reales (con parte fraccionaria)

A continuación se presentan dos ejemplos completos de conversión de números reales desde base decimal hacia otra base. En ambos casos se aplican las dos fases del procedimiento: conversión de la parte entera mediante divisiones sucesivas y conversión de la parte fraccionaria mediante multiplicaciones sucesivas. El primer ejemplo produce una expansión finita; el segundo genera una expansión infinita periódica. Para representar una serie infinita se puede usar la siguiente notación $0.\overline{133}_3$, ***133* se repite infinitamente y su base es *3***.

Ejemplo 1: Conversión finita

Convertir 13.625_{10} a binario

Parte entera

$N = 13$

N	$N \div 2$	Residuo
13	6	1
6	3	0
3	1	1
1	0	1

Leer residuos en orden inverso:

$$13_{10} = 1101_2$$

Parte fraccionaria

$f = 0.625$

$$0.625 \times 2 = 1.25 \Rightarrow 1$$

$$0.25 \times 2 = 0.5 \Rightarrow 0$$

$$0.5 \times 2 = 1.0 \Rightarrow 1$$

La fracción se hace 0, por lo tanto la conversión termina.

$$0.625_{10} = 0.101_2$$

Resultado final

$$13.625_{10} = 1101.101_2$$

Observación adicional: La conversión es finita porque $0.625 = \frac{5}{8}$ y el denominador es potencia de 2. Cuando la fracción decimal puede expresarse como $k/2^n$, su representación binaria termina.

Ejemplo 2: Conversión infinita periódica

Convertir 7.1_{10} a binario

Parte entera

$N = 7$

N	$N \div 2$	Residuo
7	3	1
3	1	1
1	0	1

$$7_{10} = 111_2$$

Parte fraccionaria

$f = 0.1$

$$0.1 \times 2 = 0.2 \Rightarrow 0$$

$$0.2 \times 2 = 0.4 \Rightarrow 0$$

$$0.4 \times 2 = 0.8 \Rightarrow 0$$

$$0.8 \times 2 = 1.6 \Rightarrow 1$$

$$0.6 \times 2 = 1.2 \Rightarrow 1$$

$$0.2 \times 2 = 0.4 \Rightarrow 0$$

Se observa que reaparece el valor 0.2, por lo que el patrón se repite.

$$0.1_{10} = 0.0001100110011\ldots_2$$

Resultado final

$$7.1_{10} = 111.0001100110011\ldots_2$$

Observación adicional: La expansión no termina porque $0.1 = \frac{1}{10}$ y el denominador contiene factores primos distintos de 2. En sistemas digitales, esta representación infinita se aproxima con un número finito de bits, lo que explica errores de redondeo en cálculos con punto flotante.

3. Conversión desde otra base hacia decimal (valores reales). El proceso inverso aplica expansión posicional incluyendo potencias negativas:

Ejemplo: convertir 101.101_2 a decimal.

$$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

$$= 4 + 0 + 1 + 0.5 + 0 + 0.125 = 5.625_{10}$$

El algoritmo consiste en:

1. Separar parte entera y fraccionaria.

2. Aplicar expansión posicional con exponentes positivos y negativos.

3. Sumar todos los términos.

Este procedimiento refuerza el entendimiento del sistema posicional extendido y promueve implementación modular en código [7], [13], [37].

Ejercicios resueltos: Conversión desde otra base hacia decimal (valores reales)

A continuación se presentan dos ejercicios completos aplicando expansión posicional con exponentes positivos y negativos. En ambos casos se separa parte entera y fraccionaria, y luego se suman todos los términos.

Ejemplo 1: Conversión finita

Convertir 1011.011_2 **a decimal**

Paso 1: Separar parte entera y fraccionaria

$$1011_2 \quad \text{y} \quad 0.011_2$$

Paso 2: Expandir parte entera

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$= 8 + 0 + 2 + 1 = 11$$

Paso 3: Expandir parte fraccionaria

$$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

$$= 0 + 0.25 + 0.125 = 0.375$$

Resultado final

$$1011.011_2 = 11.375_{10}$$

Observación adicional: La conversión es finita porque la fracción binaria tiene un número limitado de dígitos; toda fracción binaria finita produce un decimal exacto.

Ejemplo 2: Conversión finita

Convertir 27.34_8 a decimal

Paso 1: Separar parte entera y fraccionaria

$$27_8 \quad \text{y} \quad 0.34_8$$

Paso 2: Expandir parte entera

$$2 \cdot 8^1 + 7 \cdot 8^0$$

$$= 16 + 7 = 23$$

Paso 3: Expandir parte fraccionaria

$$3 \cdot 8^{-1} + 4 \cdot 8^{-2}$$

$$= 3 \cdot 0.125 + 4 \cdot 0.015625$$

$$= 0.375 + 0.0625 = 0.4375$$

Resultado final

$$27.34_8 = 23.4375_{10}$$

Observación adicional: Las fracciones en base 8 también son potencias de 2, por lo que su representación decimal es exacta.

Ejemplo 3: Conversión periódica

Convertir $0.333\ldots_3$ a decimal

Paso 1: Interpretar la fracción

$$0.333\ldots_3$$

Paso 2: Expandir con potencias negativas

$$3 \cdot 3^{-1} + 3 \cdot 3^{-2} + 3 \cdot 3^{-3} + \ldots$$

$$= 3\left(\frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \ldots\right)$$

Se trata de una serie geométrica infinita:

$$= 3 \cdot \frac{\frac{1}{3}}{1 - \frac{1}{3}} = 3 \cdot \frac{\frac{1}{3}}{\frac{2}{3}} = 3 \cdot \frac{1}{2} = 1.5$$

Resultado final

$$0.333\ldots_3 = 1.5_{10}$$

Observación adicional: Aunque la representación en base 3 es infinita, el valor decimal converge a un número racional exacto.

Ejemplo 4: Conversión periódica

Convertir $0.\overline{1}_3$ a decimal

Paso 1: Expansión

$$1 \cdot 3^{-1} + 1 \cdot 3^{-2} + 1 \cdot 3^{-3} + \ldots$$

$$= \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \ldots$$

Serie geométrica:

$$= \frac{\frac{1}{3}}{1 - \frac{1}{3}} = \frac{\frac{1}{3}}{\frac{2}{3}} = \frac{1}{2}$$

Resultado final

$$0.\overline{1}_3 = 0.5_{10}$$

Observación adicional: La periodicidad en una base distinta puede producir un decimal exacto; este fenómeno ilustra cómo la representación depende de la relación entre los factores primos de la base original y la base destino.

4. Limitaciones fundamentales: representaciones no exactas. No todos los números reales pueden representarse exactamente en todas las bases. Por ejemplo, 0.1_{10} no tiene representación finita en binario. Este fenómeno se debe a que la fracción depende de factores primos distintos de la base. En consecuencia, la representación en binario puede ser infinita o periódica, y el sistema debe aproximarla [11], [12], [38].

En programación, esta limitación explica resultados como:

$$0.1 + 0.2 \neq 0.3$$

cuando se utiliza punto flotante binario. Este comportamiento no constituye un error del lenguaje, sino una consecuencia matemática de la representación digital [7], [11], [12].

5. Estándar IEEE 754 y punto flotante. La mayoría de los lenguajes utilizan el estándar IEEE 754 para representar números reales en binario. Este estándar define:

- 1 bit para el signo.
- Bits para exponente.
- Bits para mantisa.

La representación permite manejar un rango amplio de valores, aunque introduce redondeo y errores acumulativos en operaciones repetidas [11], [12], [34].

Para quien aprende a programar en IS, esta información resulta indispensable al trabajar con:

- Cálculos financieros.
- Indicadores estadísticos.
- Promedios y porcentajes.

6. Buenas prácticas en programación con reales. El manejo responsable de valores reales incluye:

1. Evitar comparaciones directas de igualdad.
2. Utilizar tolerancias: $|a - b| < \varepsilon$.
3. Controlar redondeo antes de mostrar resultados.
4. Documentar precisión esperada según dominio organizacional.

Estas prácticas se alinean con principios de calidad y robustez en ingeniería de software [4], [23], [24].

7. Implicación formativa. La representación de números reales amplía la visión del estudiante sobre la naturaleza finita de los sistemas digitales. Programar con conciencia de precisión fortalece criterio técnico y evita interpretaciones erróneas de resultados en entornos organizacionales. Este aprendizaje trasciende ejercicios académicos y prepara al desarrollador para diseñar soluciones confiables dentro de Sistemas de Información [2], [4], [15].

Operaciones bit a bit, máscaras y estados

Las operaciones bit a bit ofrecen un mecanismo directo para manipular estados codificados. En IS, estados, permisos y banderas pueden representarse en bits para compactar información y habilitar verificaciones rápidas mediante AND, OR, XOR y desplazamientos [13], [34], [37].

Operadores bit a bit. Las operaciones bit a bit permiten manipular directamente los bits que representan un número entero. En programación, esto permite activar, desactivar o verificar estados codificados dentro de un valor numérico.

1. Operador OR (|) a nivel de bit **Problema:** Dado dos números enteros positivos, combinar sus bits utilizando la operación OR bit a bit y mostrar el resultado.

```
Algoritmo OperadorOR
Var
   a, b, resultado : entero
Inicio
   a <- 10        // 1010 en binario
   b <- 12        // 1100 en binario

   resultado <- a | b

   Escribir("Resultado OR bit a bit : ", resultado)
Fin
```

Listing 1.1: Operador OR (|) a nivel de bit

Explicación:

```
10 (1010)
12 (1100)
OR produce 1110 = 14
```

Si alguno de los bits es 1, el resultado es 1.

2. Operador AND (&) a nivel de bit **Problema:** Determinar qué bits están activos simultáneamente en dos números.

```
Algoritmo OperadorAND
Var
   a, b, resultado : entero
inicio
   a <- 10        // 1010
   b <- 12        // 1100

   resultado <- a & b

   Escribir("Resultado AND: ", resultado)
Fin
```

Listing 1.2: Operador AND (&) a nivel de bit

Explicación:

```
Solo los bits que son 1 en ambos números permanecen en 1.
1010
1100
AND produce 1000 = 8
```

3. Operador XOR (^) Problema: Determinar qué bits son diferentes entre dos números.

```
Algoritmo OperadorXOR
Var
   a, b, resultado : entero
Inicio
   a <- 10        // 1010
   b <- 12        // 1100

   resultado <- a XOR b

   Escribir("Resultado XOR: ", resultado)
Fin
```

Listing 1.3: Determininar los bits diferentes entre dos números

Explicación: El resultado es 1 cuando los bits son distintos.

```
1010
1100
XOR produce 0110 = 6
```

4. Desplazamiento Izquierdo Problema: Multiplicar un número entero por 2 usando desplazamiento de bits.

```
Algoritmo DesplazamientoIzquierdo
Var
   numero, resultado : entero
```

```
   Inicio
      numero <- 5      // 0101

      resultado <- numero << 1

      Escribir("Resultado desplazamiento izquierdo: ", resultado)
   Fin
```

Listing 1.4: Multiplicación entero por 2 con desplazamiento

Explicación: Desplazar a la izquierda equivale a multiplicar por 2. `0101 << 1` produce `1010 = 10`

5. Desplazamiento Derecho **Problema:** Dividir un número entero entre 2 usando desplazamiento de bits.

```
   Algoritmo DesplazamientoDerecho
   Var
   numero, resultado : ENTERO
   Inicio
   numero <- 8      // 1000

   resultado <- numero >> 1

   Escribir("Resultado desplazamiento derecho: ", resultado)
   Fin
```

Listing 1.5: Disión de enteros para 2 por desplazamiento

Explicación: Desplazar a la derecha equivale a dividir entre 2.

```
1000 >> 1 produce 0100 = 4
```

6. Diferencia entre | y ||

Problema: Mostrar la diferencia entre OR lógico y OR bit a bit.

```

   Algoritmo DiferenciaOR
   Var
```

```
4       a, b : entero
5    inicio
6       a <- 5
7       b <- 0
8
9       Si (a | b) > 0 Entonces
10         Escribir("OR bit a bit produjo valor distinto de 0")
11      FinSi
12
13      Si (a > 0) OR (b > 0) Entonces
14         Escribir("OR logico verdadero")
15      FinSi
16   Fin
```

Listing 1.6: Diferencia entre OR bit a bit y OR lógico

Explicación:

- `|` combina bits.
- `||` evalúa condiciones lógicas.
- `||` produce `VERDADERO` o `FALSO`.
- `|` produce un número entero.

Estos algoritmos permiten que el estudiante observe cómo los números se manipulan a nivel binario, reforzando la relación entre representación digital y programación estructurada.
Para quien aprende a programar, las máscaras introducen un repertorio práctico de técnicas. Verificar si un estado está activo, activar o desactivar banderas y construir combinaciones controladas permiten ejercitar operadores lógicos y condicionales con un sentido operativo inmediato [5], [7], [35].

El uso de estados codificados se relaciona con diseño de datos. Definir qué bit representa qué condición requiere disciplina de especificación y documentación, lo que favorece programas más legibles y sostenibles. Esta práctica se conecta con principios de calidad y mantenibilidad en ingeniería de software [4], [23], [24].

Destacando la dimensión organizacional, las máscaras permiten modelar reglas sin inflar estructuras. En sistemas con gran cantidad de registros, representar atributos booleanos

como bits reduce almacenamiento y acelera filtrado, lo que incide en desempeño del SI [2], [34], [36].

En síntesis, aprender operaciones bit a bit amplía la visión del lector sobre cómo el software controla estados reales y cómo una representación cuidadosa habilita soluciones robustas y eficientes [5], [7], [13].

1.5.2. Representación de texto, números y contenidos multimedia

Los SI manejan contenidos heterogéneos: nombres, direcciones, descripciones, cantidades, fechas e incluso imágenes o audio. Cada tipo de contenido exige una codificación adecuada para preservar significado, permitir búsquedas y sostener reglas de negocio [2], [27], [28].

La representación de texto y números constituye una fuente frecuente de defectos cuando se programa sin criterios. Incompatibilidades de codificación, supuestos sobre longitudes o errores por redondeo pueden degradar integridad del sistema y generar resultados que usuarios interpretan como inconsistencias operativas [6], [11], [12].

Para el aprendizaje, estos contenidos permiten ejercicios orientados a IS: normalización de entradas textuales, validación de campos, tratamiento de identificadores, cálculo de métricas y generación de reportes. Cada ejercicio vincula estructuras de datos con reglas organizacionales [1], [5], [7].

De manera complementaria, la codificación de contenidos multimedia aparece en escenarios de gestión documental y sistemas que almacenan evidencias. Aunque su implementación completa suele ser avanzada, una introducción conceptual motiva al estudiante a reconocer que un archivo representa bytes con formato, cabeceras y restricciones de interpretación [2], [10], [34].

Este bloque se organiza en cinco líneas: codificación de caracteres, enteros, reales, formatos multimedia y representación temporal. Cada línea ofrece un motivo técnico para programar con precisión y un puente hacia prácticas de validación y prueba [4], [23], [24].

Codificación de caracteres

La codificación de caracteres define cómo símbolos textuales se representan en bytes. En IS, texto incluye nombres, descripciones y mensajes; una codificación inadecuada introduce pérdida de información, errores de visualización y fallos en comparaciones o búsquedas [2], [6], [10].

En programación, el lector debe reconocer que una cadena es una secuencia de códigos y que operaciones como longitud, recorte o concatenación dependen de la codificación

subyacente. Esta perspectiva orienta a diseñar validaciones coherentes y evita suposiciones sobre “un carácter igual a un byte” en entornos multilingües [5], [7], [35].

La codificación también afecta intercambio entre sistemas. Cuando un SI integra fuentes diversas, la uniformidad en codificación resulta indispensable para mantener consistencia y evitar corrupción de datos en almacenamiento o transmisión [2], [27], [28].

No obstante, el problema no se limita a visualización. Una codificación inconsistente puede alterar validaciones, ordenamientos y reglas de negocio basadas en texto, lo que se traduce en comportamientos incorrectos del sistema [1], [4], [6].

BUSCAR

Representación de números enteros

Los enteros suelen representarse en formato binario con un número fijo de bits, lo cual determina rangos y comportamiento ante desbordamiento. En IS, cantidades discretas como conteos, identificadores y estados se apoyan en enteros, por lo que el programador debe elegir tamaños coherentes con el dominio [5], [7], [34].

Para aprender a programar, los enteros constituyen un terreno idóneo para entrenar validaciones. Definir rangos, evitar conversiones peligrosas y controlar operaciones de incremento y multiplicación fortalece disciplina y reduce defectos previsibles [4], [6], [23].

El desbordamiento enseña un principio útil: la corrección depende de supuestos explícitos. Cuando el estudiante documenta límites y aplica verificaciones antes de operar, mejora robustez del programa y preserva integridad de resultados [5], [6], [24].

Dentro de este panorama, el uso de enteros se conecta con estructuras de datos. Índices, tamaños de arreglos y conteos de iteraciones dependen de enteros; un error en este nivel altera recorridos y acceso a memoria, generando fallos difíciles de depurar [7], [13], [36].

De manera complementaria, los enteros se usan en codificación de estados y máscaras. Esta práctica integra lógica, representación y control, proporcionando ejercicios motivadores para aprender operadores bit a bit con significado organizacional [2], [13], [37].

Representación de números reales

Los números reales se representan habitualmente mediante punto flotante, siguiendo estándares que definen signo, exponente y mantisa. Esta representación ofrece rango amplio, aunque introduce aproximaciones y errores de redondeo que el programador debe anticipar [11], [12], [34].

En IS, el impacto aparece en cálculos financieros, promedios y métricas de rendimiento. Comparaciones directas entre reales pueden fallar por diferencias mínimas de representación; por ello, resulta aconsejable entrenar criterios de tolerancia, redondeo controlado y pruebas con casos límite [11], [12], [38].

Aprender a programar con estos criterios forma un hábito técnico valioso: evaluar qué tipo numérico se ajusta al problema. En dominios sensibles, el uso de enteros escalados o representaciones decimales controladas reduce ambigüedad y mejora consistencia de resultados [5], [23], [38].

No obstante, la selección de representación exige coherencia con almacenamiento y salida. Un valor que se calcula con punto flotante puede mostrarse redondeado para usuario y guardarse con otra precisión; el estudiante debe documentar y alinear estas decisiones para evitar discrepancias entre reporte y persistencia [2], [27], [28].

En contraste con el tratamiento de enteros, el punto flotante obliga a pensar en error numérico. Esta práctica fortalece criterio científico-técnico y promueve pruebas que evalúen estabilidad del resultado, no únicamente ejecución del programa [4], [11], [12].

Representación de imágenes, audio y video

Los contenidos multimedia se almacenan como secuencias de bytes que siguen formatos con cabeceras, metadatos y reglas de decodificación. En IS, estos contenidos aparecen en gestión documental, evidencia digital, sistemas educativos y registros audiovisuales asociados a procesos [2], [10], [34].

Para el aprendizaje, los formatos multimedia permiten introducir el concepto de "estructura dentro del archivo". Leer una cabecera, verificar un identificador de formato y calcular tamaños de bloques enseña programación orientada a especificaciones, con validaciones precisas [5], [6], [7].

La compresión constituye otro punto formativo. Aunque el diseño de códecs es avanzado, reconocer que compresión implica transformar datos para reducir tamaño sin perder propiedades relevantes aporta criterio para elegir formatos en sistemas organizacionales [2], [24], [34].

Destacando el vínculo con integridad, un archivo multimedia corrupto puede ser ilegible o inducir fallas de procesamiento. Validar longitudes, controlar lecturas y verificar condiciones de consistencia se alinea con prácticas de robustez que deben entrenarse desde programación básica [4], [6], [23].

Representación temporal: fechas, horas y zonas

Los SI dependen de tiempo para auditoría, secuenciación de eventos y análisis histórico. Representar fechas y horas exige definir convenciones: zona horaria, formato, precisión y reglas de calendario, dado que discrepancias temporales afectan reportes y control de procesos [1], [2], [27].

En programación, el tiempo se expresa mediante tipos específicos o marcas numéricas. Esta representación obliga a manejar conversiones, comparaciones y cálculos de duración, aportando ejercicios ricos para reforzar estructuras de control y validación [5], [7], [13].

La consistencia temporal requiere validar entradas. Formatos ambiguos, fechas inexistentes o diferencias de zona introducen fallos sutiles; por ello, se recomienda diseñar rutinas de parseo y normalización con pruebas que cubran casos límite [5], [6], [23].

De manera complementaria, la representación temporal se vincula con almacenamiento y consulta. Índices por fecha, filtros por período y agregaciones temporales requieren coherencia entre formato persistido y formato de presentación, lo cual fortalece disciplina de diseño [2], [28], [29].

1.5.3. Riesgos de representación y prácticas de programación robusta

La representación digital ofrece potencia, aunque introduce riesgos previsibles cuando se programa sin criterios: desbordamientos, pérdidas de precisión, incompatibilidades de codificación y discrepancias de interpretación entre componentes. En SI, estos riesgos afectan integridad, trazabilidad y confianza del usuario en resultados [2], [4], [6].

Desde la formación, tratar estos riesgos aporta un beneficio directo: el estudiante aprende a programar con pruebas, validaciones y documentación de supuestos. Este marco garantista mejora capacidad de depuración y fortalece calidad del código desde etapas iniciales [7], [23], [24].

El diseño robusto también requiere reconocer que un SI integra múltiples tecnologías. Datos pueden circular entre aplicaciones, bases de datos y dispositivos; una representación válida en un componente puede fallar en otro si no existe alineación de formatos, rangos y codificaciones [1], [10], [28].

No obstante, la robustez no se limita a prevención de fallos. Programar con representaciones explícitas facilita mantenimiento: reglas bien documentadas y conversiones centralizadas reducen cambios dispersos y disminuyen defectos al evolucionar el sistema [4],

[24], [33].

Este bloque se ha consolidado como una guía práctica para el lector: elegir tipos con criterio, validar entradas, controlar conversiones y probar casos límite. Tales prácticas acompañan a quien aprende a programar en problemas académicos y en escenarios organizacionales [5], [6], [7].

Desbordamiento, subdesbordamiento y rangos

El desbordamiento aparece cuando una operación excede el rango representable por un tipo. En enteros, puede producir valores inesperados; en punto flotante, puede generar infinitos o subdesbordamientos hacia cero. En IS, estos efectos alteran métricas, cálculos acumulativos y validaciones de reglas [6], [7], [34].

Para aprender a programar, este riesgo enseña a definir precondiciones. Verificar rangos antes de operar, usar tipos adecuados y registrar errores cuando un valor se sale del dominio fortalece el hábito de programar con contratos explícitos [4], [5], [23].

El tratamiento riguroso incluye pruebas. Diseñar casos en los límites del rango y observar comportamiento permite al estudiante indagar fallas que, de otro modo, se manifestarían en producción como inconsistencias difíciles de atribuir [6], [7], [24].

Orden de bytes, alineación y serialización

El **orden de bytes** (endianness) describe cómo se almacenan bytes de un valor multibyte. Sistemas distintos pueden ordenar de manera diferente; sin una serialización explícita, datos intercambiados pueden interpretarse de forma incorrecta [2], [10], [34].

Ejemplos prácticos de orden de bytes (Endianness). El orden de bytes determina cómo se almacenan en memoria los bytes que componen un valor multibyte (por ejemplo, enteros de 16, 32 o 64 bits). Existen dos esquemas principales:

- **Big-endian**: el byte más significativo se almacena primero.
- **Little-endian**: el byte menos significativo se almacena primero.

Ejemplo 1: Entero de 32 bits. Considérese el valor hexadecimal:

$$0x12345678$$

Este número está compuesto por 4 bytes:

$$12 \quad 34 \quad 56 \quad 78$$

Almacenamiento en Big-endian

Dirección 0	Dirección 1	Dirección 2	Dirección 3
12	34	56	78

Almacenamiento en Little-endian

Dirección 0	Dirección 1	Dirección 2	Dirección 3
78	56	34	12

Obsérvese que el valor lógico es el mismo, pero el orden físico en memoria cambia.

Ejemplo 2: Entero de 16 bits. Valor:

$$0x1234$$

Big-endian

$$12 \quad 34$$

Little-endian

$$34 \quad 12$$

Si un sistema little-endian envía estos bytes sin especificar el orden y un sistema big-endian los interpreta directamente, el valor leído sería:

$$0x3412$$

lo cual representa un número completamente distinto.

Ejemplo 3: Comunicación en redes. En redes, el estándar TCP/IP define el uso de **network byte order**, que es big-endian.

Si un procesador Intel (little-endian) envía el número:

$$0x12345678$$

debe convertirlo antes de transmitirlo.

En lenguaje C, esto se realiza con:

```
uint32_t valor = 0x12345678;
uint32_t red = htonl(valor);
```

Si no se realiza esta conversión, el receptor interpretará bytes invertidos.

Ejemplo 4: Archivos binarios. Supóngase que se guarda en un archivo binario el entero decimal 305419896, que en hexadecimal es:

$$0x12345678$$

Un sistema little-endian lo almacenará como:

$$78\ 56\ 34\ 12$$

Si el archivo se abre en un sistema big-endian sin conversión, el valor leído será:

$$0x78563412$$

equivalente a:

$$2018915346_{10}$$

Lo que demuestra cómo la ausencia de serialización explícita produce corrupción lógica de datos.

Ejemplo 5: Caso con UTF-16. El carácter Unicode:

$$U + 00F1$$

En UTF-16:

Big-endian

$$00\ F1$$

Little-endian

$$F1\ 00$$

Para indicar el orden, algunos archivos incluyen un **BOM (Byte Order Mark)**:

FE FF (Big-endian)

FF FE (Little-endian)

Conclusión técnica. El problema del endianness no altera el valor abstracto, sino su representación física. Cuando se intercambian datos entre sistemas heterogéneos (arquitecturas x86, ARM, sistemas de red, almacenamiento binario), es imprescindible:

- Definir un orden estándar (ej. network byte order).
- Utilizar funciones de conversión.
- Especificar el formato de serialización.

En programación, la **serialización** traduce estructuras en secuencias de bytes para almacenamiento o transmisión. Implementarla enseña a definir formatos, incluir longitudes y validar consistencia, lo que fortalece habilidades de diseño orientadas a especificación [5], [6], [7].

Ejemplos prácticos de serialización En programación, la **serialización** consiste en transformar una estructura de datos en una secuencia ordenada de bytes para su almacenamiento o transmisión.

Diseñar un formato de serialización implica definir explícitamente:

- El orden de los campos.
- El tamaño en bytes de cada campo.
- El orden de bytes (endianness).
- La inclusión de longitudes para datos variables.
- Mecanismos de validación (checksum, versión, delimitadores).

Ejemplo 1: Serialización binaria simple. Supóngase la siguiente estructura:

```
struct Persona {
  uint32_t id;
  uint8_t edad;
};
```

Si:

$$id = 1000_{10} = 0x000003E8$$

$$edad = 25_{10} = 0x19$$

Formato definido (big-endian):

- id: 4 bytes
- edad: 1 byte

Secuencia serializada:

$$00\ 00\ 03\ E8\ 19$$

Explicación. El diseño es explícito: siempre se esperan 5 bytes. Si se reciben menos, el mensaje es inválido.

Ejemplo 2: Inclusión de longitud en cadenas. Supóngase:

```
struct Usuario {
  uint16_t longitudNombre;
  char nombre[];
};
```

Si el nombre es:

$$"ANA"$$

En ASCII:

$$41\ 4E\ 41$$

Serialización

$$00\ 03\ 41\ 4E\ 41$$

Interpretación

- 00 03 indica que vienen 3 bytes.
- Luego se leen exactamente 3 bytes.

Este mecanismo evita ambigüedades y permite validar consistencia.

Ejemplo 3: Error por ausencia de longitud. Si solo se transmitiera:

$$41\ 4E\ 41$$

El receptor no sabría:

- Cuándo termina el nombre.
- Si vienen más campos después.
- Si los datos están incompletos.

Esto demuestra la importancia de definir especificaciones formales.

Ejemplo 4: Serialización con versión y checksum. Formato definido:

- 1 byte: versión
- 4 bytes: id
- 1 byte: edad
- 1 byte: checksum (suma módulo 256)

Datos:

$$versin = 01$$

$$id = 00\ 00\ 03\ E8$$

$$edad = 19$$

Suma de bytes:

$$01 + 00 + 00 + 03 + E8 + 19$$

Checksum:

$$(01 + 00 + 00 + 03 + E8 + 19) \text{ mód } 256 = F3$$

Mensaje final:

$$01\ 00\ 00\ 03\ E8\ 19\ F3$$

Si el receptor calcula un checksum distinto, el mensaje se descarta.

Ejemplo 5: Serialización en formato textual (JSON). Estructura:

```
{
  "id": 1000,
  "edad": 25
}
```

Ventaja

- Legible por humanos.
- Independiente de endianness.

Desventaja

- Mayor tamaño.
- Requiere análisis sintáctico.

Ejemplo 6: Problema de endianness. Si un sistema little-endian serializa:

$$0x12345678$$

como:

$$78\ 56\ 34\ 12$$

y otro sistema asume big-endian, interpretará:

$$0x78563412$$

lo cual altera completamente el valor.
Por ello, las especificaciones deben definir:

- Orden de bytes.
- Tamaño fijo de campos.
- Convenciones de codificación.

Conclusión conceptual. Implementar serialización enseña:

- Pensamiento estructural.
- Definición formal de formatos.
- Control de tamaño y memoria.
- Validación de integridad.
- Diseño orientado a especificación.

Estos principios son fundamentales en sistemas distribuidos, redes, bases de datos y arquitecturas IoT.

La **alineación de datos en memoria** también influye en desempeño y corrección cuando se trabaja cerca del hardware. Aunque muchos lenguajes abstraen esta complejidad, reconocer su existencia prepara al lector para interpretar fallos y optimizar estructuras en escenarios avanzados [10], [34], [35].

Ejemplos prácticos de alineación de datos en memoria. La **alineación** (alignment) describe la relación entre la dirección de memoria donde comienza un dato y el tamaño natural de acceso que la arquitectura utiliza para leerlo o escribirlo. En términos operativos, un dato de 4 bytes suele considerarse *alineado* si su dirección es múltiplo de 4; un dato de 8 bytes, si su dirección es múltiplo de 8. Esta regla surge de la forma en que la CPU y el subsistema de memoria transfieren información en bloques, y su incumplimiento puede introducir penalizaciones de desempeño o, en algunas arquitecturas, fallos de acceso [10], [34], [35].

Ejemplo 1: Alineación básica por múltiplos (16, 32 y 64 bits). Objetivo del ejemplo: identificar cuándo una dirección es alineada para distintos tamaños de palabra y por qué importa al leer datos multibyte [10], [34].

Supóngase un bloque de memoria donde se almacenan bytes en direcciones consecutivas:

Dir	1000	1001	1002	1003	1004	1005	1006	1007
Byte	·	·	·	·	·	·	·	·

Paso 1: considerar un entero de 32 bits (4 bytes). Si empieza en 1000, ocupa 1000–1003. Como 1000 mód $4 = 0$, el entero está alineado a 4 bytes.

Paso 2: si el mismo entero empieza en 1001, ocupa 1001–1004. Como 1001 mód $4 \neq 0$, el entero está *desalineado*.

Paso 3: para un entero de 64 bits (8 bytes), la dirección inicial debería cumplir *dir* mód $8 = 0$. En el ejemplo, 1000 no es múltiplo de 8, por lo que un valor de 8 bytes comenzando en 1000 ya sería desalineado.

i **Observación adicional**

Aun cuando una CPU permita accesos desalineados, puede requerir lecturas internas adicionales y reensamblado, introduciendo latencias [10], [34].

Ejemplo 2: Estructuras y relleno (padding) automático Problema a resolver: explicar por qué una estructura ocupa más bytes de lo esperado cuando el compilador inserta relleno para mantener alineación [34], [35].
Considérese la estructura:

Listing 1.7: Estructura con posibles bytes de relleno

```
typedef struct {
  char  c;   // 1 byte
  int   i;   // 4 bytes
  char  d;   // 1 byte
} Registro;
```

Paso 1: *c* ocupa 1 byte. Si la estructura inicia en una dirección múltiplo de 4, después de *c* el siguiente byte queda en una dirección que normalmente no es múltiplo de 4.

Paso 2: para que i (4 bytes) comience en una dirección múltiplo de 4, el compilador inserta **3 bytes de relleno** tras c.

Paso 3: i ocupa 4 bytes. Luego d ocupa 1 byte.

Paso 4: para que el tamaño total sea múltiplo del mayor alineamiento del bloque (usualmente 4 en este caso), el compilador puede insertar relleno final (por ejemplo, 3 bytes) tras d.

Resultado típico: la estructura puede ocupar 12 bytes:

$$1\ (\text{c}) + 3\ (\text{padding}) + 4\ (\text{i}) + 1\ (\text{d}) + 3\ (\text{padding}) = 12$$

i **Observación adicional**

Este relleno mejora el desempeño porque evita accesos desalineados repetidos cuando se procesan arreglos de estructuras [10], [34].

Ejemplo 3: Reordenamiento de campos para optimizar tamaño Problema a resolver: reducir el tamaño de una estructura sin perder información, aplicando un criterio de diseño que también se aprende al programar con disciplina: ordenar campos por tamaño decreciente [10], [35]. A partir de 1.7, se propone:

Listing 1.8: Estructura reordenada para disminuir padding

```
typedef struct {
  int   i;   // 4 bytes
  char  c;   // 1 byte
  char  d;   // 1 byte
} RegistroOpt;
```

Paso 1: i comienza alineado (múltiplo de 4).

Paso 2: c y d se ubican después; al agrupar bytes pequeños al final, se reduce el relleno intermedio.

Resultado típico: el tamaño baja, por ejemplo, a 8 bytes:

$$4\ (\text{i}) + 1\ (\text{c}) + 1\ (\text{d}) + 2\ (\text{padding final}) = 8$$

Observación adicional

En arreglos grandes, esta reducción disminuye consumo de memoria y mejora localidad de caché, lo que se traduce en operaciones más eficientes [10], [34].

Ejemplo 4: Corrección al leer bytes como enteros (acceso desalineado) **Problema a resolver:** mostrar cómo un acceso desalineado puede ser incorrecto o fallar según la arquitectura, y cuál es el patrón de solución profesional [10], [34].

Supóngase un arreglo de bytes que representa un encabezado binario de red/archivo:

Listing 1.9: Lectura riesgosa por conversión directa de puntero

```
uint8_t buf[] = {0x01, 0x02, 0x03, 0x04, 0xFF, 0xEE};
uint32_t *p = (uint32_t*)&buf[1]; // comienza en dirección no múltiplo de 4
uint32_t x = *p;                  // acceso potencialmente desalineado
```

Paso 1: `&buf[1]` no garantiza alineación a 4 bytes.

Paso 2: el acceso `*p` puede:

funcionar con penalización (CPU que soporta unaligned),

generar excepción/fallo (CPU/ABI que no lo permite),

producir resultados no portables (según compilador/optimización).

Solución directa y portable: copiar bytes con `memcpy` a una variable alineada y luego interpretar.

Listing 1.10: Lectura segura usando memcpy

```
uint32_t x;
memcpy(&x, &buf[1], sizeof(uint32_t)); // copia a variable alineada
```

Observación adicional

Esta práctica introduce una disciplina útil: separar representación en bytes de la representación como tipos, lo que prepara al lector para depuración en redes, archivos binarios y serialización [10], [34].

Ejemplo 5: Alineación y desempeño en arreglos (localidad y caché) Problema a resolver: explicar por qué estructuras más compactas mejoran iteraciones y reducen fallos de caché, conectando el concepto con desempeño observable [10], [34].
Considérese un arreglo con un millón de registros. Si cada registro ocupa 12 bytes, el conjunto total consume más memoria que si ocupa 8. Al recorrer el arreglo secuencialmente, se cargan líneas de caché (por ejemplo, 64 bytes) que contienen menos registros cuando cada uno es más grande.

Paso 1: estructura grande ⇒ menos elementos por línea de caché.

Paso 2: menos elementos por línea ⇒ más lecturas desde memoria principal.

Paso 3: más lecturas ⇒ mayor latencia acumulada.

i **Observación adicional**

Aunque esta optimización no se exige en ejercicios iniciales, reconocer el fenómeno permite interpretar por qué un programa "equivalente" puede rendir distinto tras un cambio de estructura o compilador [10], [34], [35].

Cierre pedagógico La alineación y el relleno no se introducen para complicar el aprendizaje, sino para fortalecer una habilidad valiosa: razonar sobre cómo una abstracción (tipos y estructuras) se materializa en memoria. Esta visión orienta mejores decisiones al diseñar formatos binarios, estructuras de datos y módulos de serialización, y facilita diagnosticar fallos que aparecen únicamente en ciertos equipos o arquitecturas [10], [34], [35].

De manera complementaria, un formato de serialización bien definido mejora mantenibilidad. Centralizar conversiones reduce duplicación y facilita evolución de estructuras sin romper compatibilidad [4], [24], [33].

Normalización y consistencia de codificaciones

Las codificaciones inconsistentes producen fallos en comparación, búsqueda y validación. Un SI puede almacenar texto con una codificación y recibir entradas con otra; sin normalización, se generan registros duplicados y reglas que fallan por desigualdad de símbolos equivalentes [2], [6], [27].

Para el aprendizaje, esta situación refuerza la práctica de normalizar entradas. Recortar espacios, unificar mayúsculas/minúsculas según política y validar caracteres permitidos convierten ejercicios simples en prácticas profesionales aplicables a formularios y registros [1], [5], [7].

No obstante, normalizar requiere documentar criterios. Una regla de normalización cambia el significado percibido por los usuarios; por ello, se recomienda registrar decisiones y asegurar coherencia entre presentación y almacenamiento [2], [4], [24].

Ejemplo 1: Diferencias invisibles en Unicode Considérese el nombre:

$$Jos$$

Este puede almacenarse de dos formas distintas en Unicode:

Forma 1 (precompuesta):

$$U + 00E9$$

Forma 2 (descompuesta):

$$U + 0065\ U + 0301$$

Visualmente son idénticas, pero a nivel binario son distintas.

Problema práctico Si un sistema almacena la forma precompuesta y recibe la forma descompuesta, una comparación directa de bytes devuelve falso.

Solución Aplicar normalización Unicode (NFC o NFD) antes de almacenar o comparar.

Ejercicio guiado

1. Recibir dos cadenas visualmente iguales.
2. Mostrar sus códigos Unicode.
3. Comparar byte a byte.
4. Aplicar normalización NFC.
5. Comparar nuevamente.

6. Verificar igualdad.

Este ejercicio enseña que la igualdad visual no implica igualdad binaria.

Ejemplo 2: Espacios invisibles y registros duplicados Entrada A:

$$"ANA"$$

Entrada B:

$$"ANA\ "$$

La segunda contiene un espacio final.

Problema Almacenar ambas entradas genera dos registros distintos.

Solución paso a paso

1. Recortar espacios iniciales y finales.
2. Validar longitud mínima.
3. Aplicar política uniforme antes de persistir.

Ejercicio práctico Diseñar una función que:

- Elimine espacios laterales.
- Convierta a mayúsculas.
- Valide que solo contenga letras.

Probar con:

$$"ana", \quad "ANA", \quad "Ana"$$

Verificar que todas se almacenen como:

$$"ANA"$$

Ejemplo 3: Sensibilidad a mayúsculas y reglas de negocio Supóngase un sistema donde:

$$"admin" \neq "ADMIN"$$

Problema En un módulo, la comparación es sensible a mayúsculas; en otro no.

Esto genera:

- Inconsistencia en autenticación.
- Errores de búsqueda.
- Resultados inesperados.

Ejercicio analítico

1. Definir una política: ¿el sistema será case-sensitive?
2. Documentar la decisión.
3. Implementar función uniforme.
4. Probar búsquedas con combinaciones de mayúsculas/minúsculas.

Este ejercicio conecta decisiones técnicas con impacto organizacional.

Ejemplo 4: Codificaciones mixtas (UTF-8 vs ISO-8859-1) Supóngase que un sistema almacena:

$$Seor$$

En UTF-8:

$$53\ 65\ C3\ B1\ 6F\ 72$$

Pero otro módulo interpreta esos bytes como ISO-8859-1, mostrando:

$$Se\pm or$$

Problema La base de datos ahora contiene valores aparentemente distintos.

Ejercicio diagnóstico

1. Mostrar bytes almacenados.
2. Interpretar con codificación incorrecta.
3. Detectar incoherencia.
4. Forzar conversión explícita a UTF-8.

Este ejercicio refuerza la importancia de especificar codificación en interfaces.

Ejemplo 5: Normalización en bases de datos Tabla de clientes:

ID	Nombre
1	José
2	Jose
3	JOSE

Problema Búsquedas por nombre devuelven resultados inconsistentes.

Ejercicio de diseño

1. Definir política (conservar acentos o no).
2. Aplicar transformación antes de insertar.
3. Crear índice sobre campo normalizado.
4. Comparar rendimiento y consistencia.

Este ejercicio integra programación, reglas de negocio y diseño de datos.

Conclusión pedagógica Normalizar no es simplemente "modificar texto"; es definir una política coherente que garantice:

- Igualdad semántica.
- Integridad de datos.
- Coherencia entre módulos.

- Comportamiento predecible en comparaciones.

La práctica sistemática de normalización fortalece el pensamiento orientado a especificación y previene errores que, en entornos reales, pueden escalar a fallos operacionales significativos.

Precisión numérica, redondeo y comparación segura

La precisión numérica afecta cualquier indicador calculado a partir de reales. Operaciones repetidas acumulan error de redondeo; comparaciones directas pueden fallar al evaluar igualdad. Estas propiedades son inherentes al punto flotante, por lo que deben tratarse como parte del diseño [11], [12], [38].

Para aprender a programar, la comparación segura introduce un patrón didáctico: usar tolerancias y verificar magnitud de la diferencia. Este patrón se integra con pruebas y con documentación de supuestos sobre precisión requerida por el dominio [4], [5], [12].

La elección de estrategia de redondeo también importa. En reportes, un redondeo distinto puede generar discrepancias entre sistemas o entre períodos, afectando confianza en el SI. Un diseño coherente alinea cálculo interno, almacenamiento y presentación [2], [11], [27].

Ejemplo 1: Representación imperfecta de 0.1 **Problema a demostrar:** por qué 0.1 no se representa exactamente en binario.

$$0.1_{10} = 0.00011001100110011\ldots_2$$

La expansión es infinita periódica, por lo que debe truncarse en memoria (IEEE 754).

Ejercicio paso a paso

1. Convertir 0.1 a binario mediante multiplicaciones sucesivas.

2. Observar repetición del patrón 0011.

3. Explicar que el almacenamiento es aproximado.

4. Mostrar que:

$$0.1 + 0.2 \neq 0.3$$

5. Calcular:

$$0.1 + 0.2 = 0.30000000000000004$$

Este ejercicio evidencia que la comparación directa puede fallar.

Ejemplo 2: Comparación insegura Código conceptual

```
double a = 0.1 + 0.2;
if (a == 0.3) {
  printf("Iguales");
}
```

Resultado esperado: la condición puede no cumplirse.

Ejercicio correctivo

1. Calcular la diferencia:

$$|a - 0.3|$$

2. Definir tolerancia:

$$\epsilon = 10^{-9}$$

3. Comparar:

$$|a - 0.3| < \epsilon$$

4. Verificar que ahora la condición es verdadera.

Este patrón introduce el concepto de comparación segura.

Ejemplo 3: Acumulación de error Problema a demostrar: error acumulado en sumas repetidas.

```
double suma = 0.0;
for (int i = 0; i < 1000; i++) {
  suma += 0.1;
}
```

Resultado teórico esperado:

$$100.0$$

Resultado real aproximado:

$$99.99999999998$$

Ejercicio analítico

1. Explicar que cada suma introduce pequeño error.
2. Multiplicar error por número de iteraciones.
3. Relacionar con indicadores financieros o científicos.

Este ejercicio muestra cómo operaciones repetidas amplifican desviaciones.

Ejemplo 4: Estrategias de redondeo Considérese el valor:

$$2.345$$

Redondeo tradicional (half-up)

$$2.35$$

Redondeo hacia abajo

$$2.34$$

Redondeo bancario (half-even)

$$2.34 \quad \text{si el dígito anterior es par}$$

Ejercicio práctico

1. Calcular promedio mensual con distintos métodos de redondeo.
2. Comparar totales anuales.
3. Analizar discrepancias acumuladas.

Este ejercicio demuestra que la estrategia de redondeo impacta resultados organizacionales.

Ejemplo 5: Precisión en almacenamiento y presentación Supóngase un sistema financiero:

$$\text{Monto real} = 1234.56789$$

Almacenamiento interno: double con alta precisión.

Presentación al usuario: 2 decimales.

$$1234.57$$

Ejercicio de coherencia

1. Definir precisión interna.
2. Definir precisión de almacenamiento en base de datos.
3. Definir formato de presentación.
4. Documentar política.

Este ejercicio integra diseño técnico y confianza organizacional.

i **Conclusión aplicada**

La precisión numérica no es un detalle de implementación sino una propiedad estructural del punto flotante.
Aplicar tolerancias, definir estrategias de redondeo y documentar supuestos convierte ejercicios básicos en prácticas de ingeniería rigurosa. Esta disciplina previene errores invisibles que, en sistemas reales, pueden generar inconsistencias financieras, científicas o administrativas.

Integridad de datos: detección de errores y validación cruzada

La integridad requiere detectar errores durante captura, transmisión y almacenamiento. Validaciones de formato, verificaciones de longitud y controles cruzados entre campos reducen la probabilidad de registrar información defectuosa [2], [6], [28]. En programación básica,

estas prácticas se implementan mediante funciones de validación y pruebas sistemáticas. El estudiante aprende a separar reglas de negocio de lógica de entrada, lo que mejora legibilidad y facilita mantenimiento [5], [7], [33].

De manera complementaria, la integridad se beneficia de restricciones en almacenamiento: claves, dominios y relaciones preservan coherencia cuando múltiples módulos escriben datos. Esta visión conecta programación con fundamentos de bases de datos [27], [28], [29]. No obstante, la validación cruzada debe diseñarse con criterio para evitar reglas contradictorias. Documentar decisiones y alinear validaciones con el proceso organizacional reduce fricción y fortalece trazabilidad [1], [4], [16].

Ejemplo 1: Validación de formato **Problema:** capturar una dirección de correo electrónico válida.

Regla de formato Debe contener:

- Un símbolo @
- Un dominio posterior
- Sin espacios

Ejercicio práctico

1. Solicitar entrada de usuario.
2. Verificar presencia de @.
3. Verificar que la longitud sea mayor que 5 caracteres.
4. Rechazar entradas como:

$$\text{"}usuario@\text{"}, \quad \text{"}@dominio.com\text{"}, \quad \text{"}usuariodominio.com\text{"}$$

Este ejercicio enseña validación de formato antes de almacenar.

Ejemplo 2: Verificación de longitud **Problema:** un código de producto debe tener exactamente 8 caracteres.

Ejercicio paso a paso

1. Leer código.
2. Calcular longitud.
3. Si longitud $\neq 8$, mostrar error.
4. Permitir reingreso.

Esto previene registros incompletos o inconsistentes.

Ejemplo 3: Validación cruzada entre campos **Problema:** en un sistema académico, un estudiante no puede tener estado "Graduado" si créditos aprobados $<$ créditos requeridos.

Ejercicio de control cruzado

1. Leer créditos aprobados.
2. Leer estado académico.
3. Si estado = "Graduado" y créditos < 240:

   Error de coherencia

4. Registrar inconsistencia.

Este ejercicio demuestra que la validación no depende de un solo campo, sino de relaciones entre ellos.

Ejemplo 4: Separación de reglas de negocio **Problema:** no mezclar validación con lógica principal.

Diseño adecuado

- Función validarEdad(edad)
- Función validarCorreo(correo)
- Función registrarUsuario()

Ejercicio estructural

1. Crear función validarEdad (18–65).
2. Crear función validarCorreo.
3. En programa principal, llamar validaciones.
4. Si todas son correctas, registrar.

Esto mejora mantenimiento y claridad del código.

Ejemplo 5: Restricciones en almacenamiento (Base de datos) Supóngase una tabla de clientes:

ID	Email

Restricciones necesarias

- ID como clave primaria.
- Email como único (UNIQUE).
- Dominio restringido (CHECK).

Ejercicio conceptual

1. Intentar insertar dos clientes con mismo email.
2. Verificar que la base rechaza duplicado.
3. Explicar por qué la restricción protege integridad.

Este ejercicio conecta programación con fundamentos de bases de datos.

Ejemplo 6: Validación contradictoria **Problema:** un módulo exige edad mínima 18, otro módulo acepta desde 16.

Ejercicio de análisis

1. Detectar inconsistencias entre módulos.
2. Documentar política oficial.
3. Ajustar validaciones para coherencia.

Este caso muestra la importancia de alinear reglas con el proceso organizacional.

Conclusión aplicada La integridad de datos no depende de una única validación, sino de un conjunto coherente de controles en múltiples niveles:

- Validación de entrada.
- Control cruzado.
- Restricciones de almacenamiento.
- Documentación formal de reglas.

Estos ejercicios transforman conceptos teóricos en prácticas aplicables a sistemas reales, fortaleciendo la capacidad del estudiante para diseñar software robusto y coherente.

Implicaciones de seguridad en la representación

La representación afecta seguridad cuando el programa interpreta entradas sin control, convierte tipos de manera insegura o asume longitudes sin verificarlas. Estas situaciones abren espacio a defectos y ataques basados en entradas maliciosas, incluso en programas sencillos [4], [6], [23].

Para aprender a programar, la seguridad se entrena como un hábito: validar, limitar, registrar y fallar de manera controlada. Esta práctica fortalece robustez y alinea el desarrollo con estándares que catalogan vulnerabilidades comunes y recomiendan mitigaciones [5], [6], [24].

El tratamiento de archivos y de formatos constituye un caso ilustrativo. Leer longitudes sin verificar o procesar cabeceras sin validar puede conducir a lecturas fuera de rango o a corrupción de memoria en lenguajes que exponen esos riesgos. La disciplina de validación reduce probabilidad de fallas [6], [10], [34].

En síntesis, la seguridad vinculada a representación invita al lector a programar con criterio profesional desde el inicio. La inversión en validación y pruebas produce sistemas más estables, facilita mantenimiento y fortalece confianza en resultados del SI [2], [4], [23].

Ejemplo 1: Conversión insegura de tipos **Problema:** convertir entrada de usuario directamente a entero sin validar.

```
char entrada[10];
scanf("%s", entrada);
int valor = atoi(entrada);
```

Si el usuario ingresa:

"999999999999"

Puede producir desbordamiento o comportamiento indefinido.

Ejercicio correctivo

1. Verificar que la entrada contenga solo dígitos.
2. Validar longitud máxima.
3. Usar función segura (por ejemplo, `strtol`).
4. Comprobar rango antes de asignar.

Este ejercicio demuestra que convertir tipos sin control abre vulnerabilidades.

Ejemplo 2: Desbordamiento por asumir longitud **Problema:** copiar cadena sin verificar tamaño.

```
char destino[8];
strcpy(destino, entrada);
```

Si `entrada` contiene más de 7 caracteres, se produce sobrescritura de memoria.

Ejercicio preventivo

1. Medir longitud antes de copiar.
2. Usar `strncpy`.
3. Limitar tamaño máximo permitido.
4. Registrar intento inválido.

Este caso ilustra cómo la representación incorrecta puede derivar en corrupción de memoria.

Ejemplo 3: Lectura insegura de archivo binario Supóngase un archivo cuyo encabezado contiene un campo de longitud:

```
1              uint32_t longitud;
2              fread(&longitud, sizeof(uint32_t), 1, archivo);
3              char buffer[100];
4              fread(buffer, 1, longitud, archivo);
5
```

Problema: si el archivo indica longitud = 5000, se intenta leer más allá del tamaño del buffer.

Ejercicio de validación

1. Verificar que longitud $\leq$ tamaño del buffer.
2. Validar que longitud sea coherente con tamaño real del archivo.
3. Rechazar archivo si condición falla.

Este ejercicio conecta representación binaria con riesgo de explotación.

Ejemplo 4: Interpretación incorrecta de formato **Problema:** asumir que todos los datos están en UTF-8.

Un atacante envía datos con codificación distinta, provocando interpretación errónea o bypass de filtros.

Ejercicio de mitigación

1. Detectar codificación.
2. Convertir explícitamente a formato interno estándar.
3. Validar caracteres permitidos.

Este ejercicio demuestra que la representación textual también impacta seguridad.

Ejemplo 5: Validación cruzada en autenticación **Problema:** aceptar credenciales sin validar coherencia entre usuario y rol.

Escenario Usuario: `"admin"` Rol: `"usuario"`

Ejercicio

1. Validar que el rol asignado corresponda al perfil almacenado.
2. Registrar intento inconsistente.
3. Rechazar operación.

Este caso muestra que la seguridad no depende solo del formato, sino de la coherencia semántica.

Ejemplo 6: Fallo controlado **Problema:** programa termina abruptamente ante error de entrada.

Ejercicio de diseño robusto

1. Detectar entrada inválida.
2. Mostrar mensaje claro.
3. Registrar evento.
4. Permitir reintento.

Fallar de manera controlada reduce impacto y mejora experiencia de usuario.

i **Conclusión aplicada**

La representación no es neutra: afecta memoria, interpretación, seguridad y confiabilidad.
Los ejercicios anteriores muestran que validar longitudes, tipos y coherencia semántica transforma programas simples en sistemas robustos. La disciplina de validar antes de procesar constituye una práctica profesional que previene defectos y mitiga vulnerabilidades desde etapas tempranas del desarrollo.

1.6. Formulación del problema informacional

La formulación del problema informacional establece la relación explícita entre una necesidad organizacional y una solución programable. En Ingeniería en Sistemas de Información, el software se concibe como un artefacto socio-técnico: integra procesos, datos y reglas operativas para producir resultados verificables que apoyan coordinación, control y toma de decisiones. Por ello, el problema informacional se define como la especificación de *qué* hechos del negocio deben registrarse, *bajo qué* reglas deben transformarse y *cómo* se presentan sus resultados con utilidad operativa y gerencial [1], [2], [24].

En formación inicial, esta actividad orienta el aprendizaje hacia programación con propósito. Cuando el estudiante parte de un enunciado organizacional y lo traduce a entradas, procesos y salidas, adquiere un criterio de diseño que trasciende la sintaxis: decide estructuras de datos por significado, elabora validaciones por política y delimita responsabilidades del programa por función. Esta práctica enlaza pensamiento algorítmico con análisis y diseño de sistemas, facilitando que el código refleje de manera rastreable el comportamiento esperado del proceso que representa **SWEBOK2014**, [23].

Una formulación rigurosa también fortalece verificabilidad. Estándares de ingeniería de requisitos insisten en claridad, ausencia de ambigüedad, trazabilidad y posibilidad de prueba; estos atributos permiten derivar casos de prueba, criterios de aceptación y evidencia de cumplimiento. Incluso en ejercicios básicos, adoptar esta disciplina reduce errores de interpretación, mejora la consistencia de soluciones y proporciona un marco estable para iterar sin perder control sobre el alcance **ISO29148**, **IEEE12207**.

Desde la perspectiva de construcción del programa, formular el problema equivale a definir un contrato operativo: qué condiciones deben cumplir las entradas, qué transformaciones deben ejecutarse y qué salidas deben producirse con formato y significado precisos. De manera complementaria, esta formalización delimita estados del sistema y escenarios de error (datos faltantes, formatos inválidos, combinaciones inconsistentes), lo que conduce a programas más robustos, más fáciles de probar y con depuración más directa, porque cada fallo puede asociarse a un punto de la especificación [5], [7], [13].

En Sistemas de Información Transaccionales (Transactional Information Systems TIS), la formulación incorpora propiedades que condicionan la programación desde el inicio. Cada operación relevante se expresa como transacción que modifica datos persistentes y, por tanto, debe preservar integridad y consistencia entre registros bajo uso concurrente. Este requisito obliga a especificar validaciones cruzadas, reglas de unicidad, relaciones entre entidades y comportamiento ante fallos parciales, conectando programación con fundamentos de

bases de datos y control de errores. Dentro de este panorama, la formulación del problema informacional se consolida como una guía para implementar transacciones correctas y auditables, coherentes con reglas de negocio y con la estructura del almacenamiento [27], [28], [29].

Entradas del sistema

Las entradas del sistema constituyen el punto de contacto entre el entorno organizacional y el modelo computacional que lo representa. En Sistemas de Información, toda transacción o proceso depende de datos cuya calidad, formato y consistencia determinan la validez de los resultados posteriores. Por ello, el diseño de las entradas no se limita a la lectura técnica de datos, sino que implica definir reglas de validación, mecanismos de verificación cruzada y estrategias de control que preserven integridad y coherencia desde el momento mismo de la captura [1], [2], [24]. En el contexto formativo, comprender la naturaleza y diversidad de las entradas permite al estudiante anticipar errores, estructurar validaciones y desarrollar programas más robustos y confiables [5], [7].

Ingreso manual de datos (formularios e interfaces gráficas). En entornos transaccionales, el ingreso manual ocurre en puntos de operación como cajas, ventanillas administrativas o módulos académicos. El usuario introduce identificadores, cantidades, fechas o selecciones que deben cumplir restricciones de dominio y formato. Programar este tipo de entrada exige implementar validaciones de obligatoriedad, control de rangos (por ejemplo, cantidades mayores que cero), verificación de formatos (identificaciones, fechas, códigos) y mensajes de retroalimentación claros que faciliten la corrección inmediata [5], [6], [7]. Esta práctica refuerza el principio de que la integridad comienza en la interfaz de captura.

Archivos estructurados (CSV, JSON, XML, TXT). La carga desde archivos se utiliza en procesos por lote, migraciones de datos y conciliaciones contables. Aquí el programa debe validar estructura, número de columnas, tipos de datos, delimitadores y codificación de caracteres antes de incorporar la información al sistema. Además, resulta esencial registrar eventos anómalos (líneas incompletas, campos inválidos) para auditoría y trazabilidad [23], [28]. Este medio introduce al estudiante en la lectura secuencial, el manejo de excepciones y la importancia de verificar coherencia antes de persistir datos.

Bases de datos (consultas y recuperación persistente). En OLTP, muchas entradas no provienen del exterior inmediato, sino del propio almacenamiento persistente. Por ejemplo, registrar una venta requiere consultar la existencia del cliente, verificar disponibilidad de inventario y recuperar precios vigentes. Estas operaciones deben garantizar consistencia referencial y sincronización con el estado actual de los datos [27], [28], [29]. Desde la perspectiva pedagógica, este medio enfatiza la diferencia entre datos temporales y datos persistentes, así como la necesidad de validar la existencia y unicidad de registros antes de proceder.

Consumo de servicios web (APIs e integraciones externas). Las integraciones con sistemas externos —como verificación de identidad, pasarelas de pago o facturación electrónica— representan una fuente crítica de entrada. Programar estas interacciones implica manejar tiempos de espera, respuestas parciales, códigos de estado HTTP, autenticación y validación de esquemas de datos serializados **Fielding2000REST**, [23], [24]. En formación, este medio permite comprender que la estabilidad del sistema depende no solo del código propio, sino también del comportamiento de servicios externos, lo que exige mecanismos de reintento y control de errores.

Mensajería y colas de eventos. En arquitecturas orientadas a eventos, las entradas llegan como notificaciones asincrónicas (por ejemplo, confirmación de pago). En estos casos, el programa debe implementar procesamiento idempotente para evitar duplicidades y garantizar que una transacción no se ejecute más de una vez ante reenvíos del mismo mensaje [6], [24], [27]. Este enfoque introduce conceptos de concurrencia y control transaccional que amplían la comprensión del estudiante sobre integridad en entornos distribuidos.

Dispositivos y captura automatizada (lectores, sensores, terminales). Algunos sistemas integran dispositivos físicos como lectores de códigos de barras, balanzas o sensores ambientales. Estas entradas exigen validación de tramas, control de rangos y sincronización inmediata con el flujo transaccional [10], [34]. Desde el punto de vista formativo, este medio ilustra la interacción entre hardware y software, y la necesidad de verificar coherencia antes de aceptar datos generados automáticamente.

Importante...

En conjunto, las entradas del sistema no deben entenderse como simples valores iniciales para un algoritmo, sino como representaciones formales de hechos organizacionales cuya precisión condiciona todo el ciclo de procesamiento. Diseñarlas adecuadamente implica aplicar criterios de validación, consistencia y control que reflejen las políticas del dominio y las restricciones técnicas del entorno. Así, el estudiante aprende que la calidad del resultado depende, en gran medida, de la calidad del dato capturado y validado desde su origen [2], [23], [29].

Salidas esperadas

Las salidas de un Sistema de Información representan la materialización observable del procesamiento realizado sobre los datos de entrada. En sistemas transaccionales (OLTP), la salida no se limita a mostrar un resultado, sino que constituye evidencia formal de que una operación fue validada, ejecutada y registrada conforme a reglas organizacionales y restricciones técnicas previamente definidas. Por ello, el diseño de las salidas exige precisión estructural, control de formato y definición explícita del medio de entrega, garantizando consistencia, trazabilidad y protección de la información según el rol del usuario [2], [23], [24].

Desde una perspectiva funcional, una primera categoría corresponde a **salidas interactivas en pantalla**. Estas incluyen confirmaciones, advertencias y detalles inmediatos de la transacción (por ejemplo, número de comprobante, subtotal, impuestos y total). Su correcta implementación requiere coherencia entre validaciones y mensajes, claridad en la retroalimentación y consistencia semántica con el estado del proceso ejecutado. En el ámbito formativo, este tipo de salida permite al estudiante consolidar la relación entre estructuras condicionales, control de errores y presentación de resultados comprensibles para el usuario final [1], [5], [7].

Una segunda categoría comprende la **persistencia estructurada de resultados**. En OLTP, la salida principal suele consistir en la escritura consistente de registros en bases de datos: inserción de cabeceras y detalles, actualización de inventarios o generación de bitácoras. Estas salidas son fundamentales para auditoría, trazabilidad histórica y generación posterior de reportes. La implementación exige garantizar integridad referencial, unicidad de identificadores y coherencia transaccional, aspectos que conectan programación

con fundamentos de bases de datos y control de concurrencia [27], [28], [29].

Otra forma relevante corresponde a la generación de **documentos formales**, tales como facturas, tickets, certificados o reportes en PDF. Aquí se evidencia la separación entre cálculo y presentación: el programa realiza operaciones numéricas con precisión interna y posteriormente aplica reglas de formato, redondeo y representación monetaria para producir un documento legible y normativamente válido. Este proceso introduce al estudiante en consideraciones sobre precisión numérica y consistencia de representación, particularmente en contextos financieros [2], [11], [38].

Asimismo, las salidas pueden materializarse como **archivos de intercambio** (CSV, JSON, XML) destinados a conciliaciones contables o integración con otros sistemas. La correcta generación de estos archivos requiere controlar delimitadores, orden estable de campos, codificación de caracteres y validación de totales agregados para evitar discrepancias en procesos posteriores. Desde el punto de vista pedagógico, este formato refuerza la noción de interoperabilidad y la importancia de respetar contratos de datos definidos entre sistemas [6], [27], [28].

En arquitecturas distribuidas, la salida puede expresarse como **respuestas de servicios** o **mensajes asincrónicos**. En el primer caso, el sistema devuelve estructuras serializadas acompañadas de códigos de estado que indican éxito o error; en el segundo, publica eventos que desencadenan acciones en otros módulos (por ejemplo, notificaciones de stock bajo). Estas modalidades introducen conceptos de contrato de interfaz, consistencia eventual y control de idempotencia, ampliando la comprensión del estudiante sobre sistemas interconectados [6], [23], [24].

i **Resumiendo...**

El diseño de salidas debe considerar la **segmentación por roles organizacionales**. La información generada no se distribuye indiscriminadamente: mientras un operador visualiza detalles operativos, un directivo requiere indicadores agregados, y un usuario final accede únicamente a datos que le competen. Este principio articula programación con gobierno de datos y control de acceso, reforzando la dimensión organizacional del sistema [1], [2], [27].

En conjunto, las salidas esperadas no constituyen meros productos finales del algoritmo, sino componentes estructurales que garantizan verificabilidad, consistencia y utilidad

organizacional. Diseñarlas adecuadamente implica integrar criterios de precisión numérica, control transaccional, interoperabilidad y segmentación de información, consolidando así una visión profesional del desarrollo de Sistemas de Información.

Procesos

Procesos

En el contexto de los Sistemas de Información Transaccionales (TIS), el proceso constituye el núcleo transformador que convierte entradas validadas en resultados consistentes y persistentes. Desde una perspectiva formal, un proceso se define como una secuencia estructurada de operaciones gobernadas por reglas explícitas del dominio organizacional. En programación básica, esta estructura se materializa mediante decisiones condicionales, ciclos iterativos, funciones auxiliares y mecanismos de control de errores; sin embargo, en TIS el proceso adquiere una dimensión adicional al estar vinculado con estados persistentes y con la necesidad de preservar integridad bajo uso concurrente [2], [23], [24].

En términos operativos, el proceso transaccional se organiza como una transacción: una secuencia ordenada que (i) valida entradas, (ii) consulta el estado actual del almacenamiento persistente, (iii) aplica reglas de negocio, y (iv) confirma cambios de manera consistente. Esta estructura introduce al estudiante en principios fundamentales como atomicidad, consistencia e aislamiento, esenciales cuando múltiples usuarios interactúan simultáneamente con el sistema [27], [28], [29]. De este modo, el proceso deja de ser únicamente una transformación lógica y se convierte en un mecanismo formal de preservación del estado organizacional.

Desde una perspectiva didáctica, resulta conveniente estructurar el proceso como un *pipeline* de etapas claramente delimitadas: normalización de entradas, validación de formato y dominio, verificación cruzada contra datos persistentes, cálculo de resultados y persistencia controlada. Esta descomposición facilita pruebas unitarias, depuración incremental y trazabilidad entre requisito y código implementado [5], [7], [24]. Además, fomenta la modularidad y la claridad estructural, atributos recomendados en prácticas de ingeniería de software.

Un aspecto crítico en TIS es el tratamiento sistemático de excepciones. Condiciones como inexistencia de registros, duplicidad de identificadores, insuficiencia de recursos o fallos de comunicación externa deben gestionarse mediante rutas alternativas claramente definidas, evitando efectos colaterales no deseados. Programar excepciones fortalece la robustez del sistema y enseña a documentar condiciones límite, en coherencia con recomendaciones de

programación segura [6], [10], [23].

Asimismo, la construcción de procesos requiere definir invariantes explícitos del dominio. Por ejemplo, en una venta transaccional: el total debe coincidir con la suma de sus ítems, el inventario no puede quedar en valores negativos y el identificador del comprobante debe ser único. Estas invariantes, implementadas como validaciones o aserciones, previenen inconsistencias silenciosas y refuerzan la disciplina de verificación interna del sistema [6], [27], [28].

i **Importante...**

En escenarios multiusuario, los procesos deben considerar concurrencia y sincronización. Dos operaciones simultáneas sobre el mismo registro pueden generar condiciones de carrera si no existen mecanismos adecuados de aislamiento. El análisis y simulación de estos escenarios en el aula permite introducir nociones de bloqueo, control transaccional y manejo de conflictos, ampliando la comprensión del estudiante sobre el comportamiento real de los TIS [27], [29], [34].

Ejemplo aplicado (TIS): Registro de una venta con control de inventario Se considera el siguiente escenario transaccional: un operador registra una venta compuesta por varios productos; el sistema debe validar existencia de cliente y productos, verificar disponibilidad de inventario, calcular subtotales e impuestos, actualizar existencias y generar un comprobante persistente. Este ejemplo integra validación, cálculo, persistencia y control transaccional en una sola unidad coherente [2], [23], [29].

Listing 1.11: Pseudocódigo: Proceso transaccional `registrarVenta`

```
Algoritmo registrarVenta
Entrada: listaItems (productoId, cantidad), clienteId
Salida: comprobanteId, total, estado

// 1) Validaciones iniciales
Si clienteId es vacio Entonces
  devolver Error("Cliente requerido")
FinSi

Para cada item en listaItems Haga
```

```
    Si item.cantidad <= 0 Entonces
      devolver Error("Cantidad invalida")
    FinSi
  FinPara

  // 2) Validacion cruzada contra almacenamiento persistente
  Si no existeCliente(clienteId) Entonces
    devolver Error("Cliente inexistente")
  FinSi

  Para cada item en listaItems Haga
    Si no existeProducto(item.productoId) Entonces
      devolver Error("Producto inexistente")
    FinSi
    Si stock(item.productoId) < item.cantidad Entonces
      devolver Error("Stock insuficiente")
    FinSi
  FinPara

  // 3) Calculo de montos
  total <- 0
  Para cada item en listaItems Haga
    precio <- precioVigente(item.productoId)
    total <- total + (precio * item.cantidad)
  FinPara
  impuestos <- calcularImpuesto(total)
  totalFinal <- total + impuestos

  // 4) Persistencia atomica
  IniciarTransaccion()
  comprobanteId <- insertarVenta(clienteId, totalFinal, impuestos)
  Para cada item en listaItems Haga
    insertarDetalle(comprobanteId, item.productoId, item.cantidad,
   precioVigente(item.productoId))
    actualizarStock(item.productoId, stock(item.productoId) - item.cantidad)
  FinPara
  ConfirmarTransaccion()
```

```
47
48  devolver OK(comprobanteId, totalFinal)
49  FinAlgoritmo
```

Utilidad formativa: este ejemplo permite ejercitar validación estructurada, ciclos, modularización y manejo de errores, a la vez que introduce el principio de atomicidad propio de los TIS: la transacción se aplica íntegramente o no se aplica en absoluto. Esta propiedad garantiza coherencia y evita estados intermedios inconsistentes [6], [27], [29].

Restricciones organizacionales y técnicas

Las restricciones organizacionales incluyen políticas internas, normativas legales y reglas institucionales. Por ejemplo, protección de datos personales según regulaciones vigentes.

Existen restricciones técnicas como capacidad de almacenamiento, rendimiento esperado o compatibilidad con sistemas heredados.

El presupuesto y los recursos humanos también limitan alcance y complejidad.

Las restricciones de seguridad exigen controles de autenticación y autorización.

Finalmente, las restricciones temporales (plazos académicos o administrativos) condicionan la priorización de funcionalidades [24].

1.6.1. Del enunciado organizacional al modelo resoluble

Traducir un enunciado organizacional en modelo computacional implica abstraer la realidad en estructuras de datos y algoritmos. Por ejemplo, “gestionar biblioteca universitaria” se convierte en entidades: Libro, Usuario, Préstamo.

Stakeholders/Interesados

Los stakeholders incluyen usuarios finales, administradores, desarrolladores y directivos. Cada uno tiene expectativas distintas.

Por ejemplo, el estudiante desea rapidez y claridad; el administrador requiere reportes consolidados.

Identificar interesados permite priorizar requisitos y definir criterios de aceptación **ISO29148**.

Supuestos operativos

Los supuestos delimitan el contexto de operación. Ejemplo: “Se asume conexión estable a Internet” o “Todos los usuarios están previamente registrados”.

Documentar supuestos evita malentendidos y facilita validación posterior [24].

Casos de prueba representativos

Un caso de prueba para registro académico puede ser: Entrada: estudiante con identificación válida. Proceso: validación de requisitos académicos. Salida esperada: matrícula aprobada.

Otro caso: estudiante con deuda pendiente. Salida esperada: rechazo con mensaje explícito.

Estos casos permiten verificar funcionalidad antes de despliegue [23].

Criterios de aceptación funcional

Un criterio de aceptación debe ser medible. Ejemplo: “El sistema calcula correctamente el promedio con dos decimales de precisión”.

Otro criterio puede establecer tiempos máximos de respuesta.

Definir criterios explícitos fortalece calidad y coherencia entre requisito y solución implementada **ISO29148**, **SWEBOK2014**.

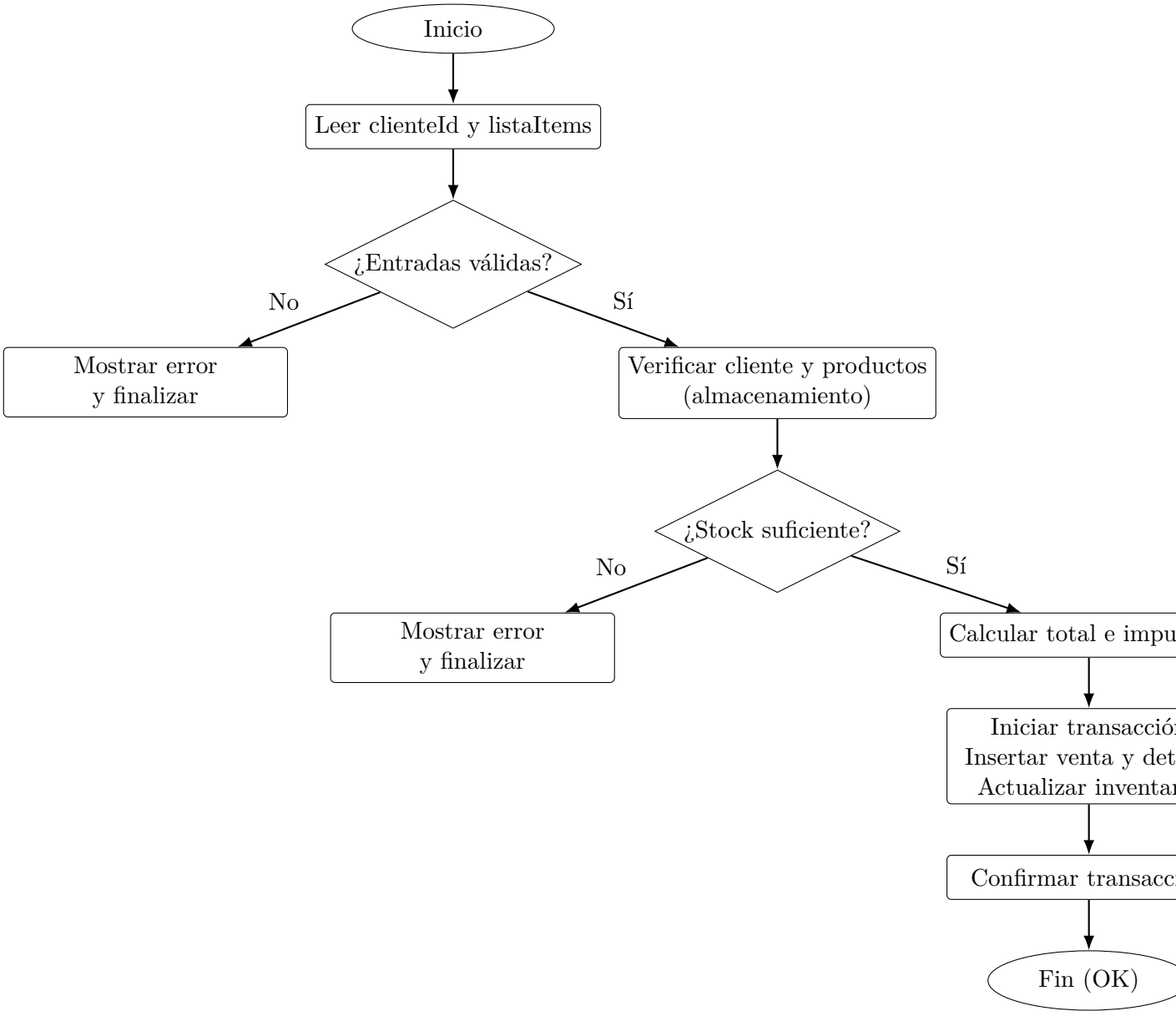

Figura 1.1: Flujo general del proceso transaccional en un Sistema de Información Transaccional (TIS).

Capítulo 2

Algoritmos y Modelado de Soluciones Informacionales

2.1. Proceso de resolución de problemas en Sistemas de Información

2.1.1. Análisis del problema organizacional

Identificación de variables relevantes

Definición de casos límite

Validación de requisitos funcionales

2.1.2. Diseño de soluciones informacionales

Estrategias de descomposición

Refinamiento sucesivo

Selección de estructuras lógicas

2.1.3. Verificación y mejora continua

Pruebas funcionales

Depuración

Optimización básica

2.2. Algoritmos: fundamento para soluciones organizacionales

2.2.1. Definición y propósito en IS

Parte II

Desarrollo de Aplicaciones para Sistemas de Información

Capítulo 3

Lenguajes de Programación y Desarrollo de Aplicaciones

3.1. Del algoritmo a la aplicación organizacional

3.1.1. Estructura mínima de un programa

Entradas y salidas

Bloques y alcance

Ejecución paso a paso

3.1.2. Errores frecuentes en el desarrollo inicial

Errores de sintaxis

Errores lógicos

Errores de ejecución

3.2. Entorno de desarrollo profesional

3.2.1. IDE y flujo de trabajo

Creación de proyectos

Compilación y ejecución

Depuración

3.2.2. Estándares básicos de codificación

Indentación

Nombres significativos

Capítulo 4

Modelado de Datos y Expresiones en Aplicaciones

4.1. Tipos de datos en aplicaciones informacionales

4.1.1. Datos numéricos y lógicos

Enteros

Reales

Booleanos

4.1.2. Texto y estructuras básicas

Caracteres

Cadenas

Operaciones frecuentes

4.2. Variables y gestión del estado

4.2.1. Declaración e inicialización

Buenas prácticas

Ámbito y ciclo de vida

Constantes

4.2.2. Asignación y actualización 112

Asignación simple

Asignación compuesta

Parte III

Control de Flujo y Gestión de Datos

Capítulo 5

Estructuras de Control en Aplicaciones Informacionales

5.1. Estructuras de selección

5.1.1. Selección simple y doble

Condición

Bloques alternativos

Anidamiento

5.1.2. Selección múltiple

Casos

Cobertura

Errores frecuentes

5.2. Estructuras de repetición

5.2.1. Repetición condicional

Precondición

Postcondición

Condiciones de parada

5.2.2. Repetición por contador 116

Contadores

Acumuladores

Capítulo 6

Estructuración y Organización de Datos

6.1. Arreglos unidimensionales

6.1.1. Concepto y uso

Índices

Recorridos

Errores de rango

6.2. Arreglos bidimensionales

6.2.1. Matrices

Filas y columnas

Recorridos

Aplicaciones organizacionales

6.3. Registros y datos compuestos

6.3.1. Estructuras simples

Campos

Acceso

Validación

Parte IV

Modularidad, Calidad e Integración en Sistemas de Información

Capítulo 7

Modularidad y Funciones en Aplicaciones

7.1. Abstracción y descomposición funcional

7.1.1. Diseño modular

Responsabilidad única

Cohesión y acoplamiento

Reutilización

7.2. Funciones y procedimientos

7.2.1. Definición e invocación

Firma

Paso de parámetros

Ámbito

7.3. Pruebas y calidad básica

7.3.1. Estrategia de pruebas

Casos representativos

Casos límite

Regresión

Parte V

Capítulo 8

Proyecto Integrador en Sistemas de Información

8.1. Planteamiento organizacional

8.1.1. Contexto y alcance

Objetivo

Requisitos

Restricciones

8.2. Diseño e implementación

8.2.1. Estructura del sistema

Módulos

Flujo principal

Casos alternativos

Capítulo 9

Gestión básica de versiones en entornos colaborativos

9.1. Fundamentos

9.1.1. Repositorio y control de cambios

Repositorio

Commit

Ramas

Capítulo 10

Glosario y Recursos Académicos

10.1. Términos fundamentales

10.2. Bibliografía y fuentes de consulta

Apéndice A

Operaciones bit a bit, máscaras y estados

Las operaciones bit a bit permiten manipular directamente los bits que componen un valor almacenado en memoria. En lenguajes como C y C++, estas operaciones actúan sobre representaciones binarias completas (palabras de 8, 16, 32 o 64 bits), mientras que los operadores lógicos trabajan sobre valores booleanos derivados de expresiones. Distinguir entre ambos niveles resulta fundamental en programación de SI cuando se diseñan mecanismos de control de estados, permisos y banderas codificadas dentro de variables enteras.

A.1. Operador OR bit a bit |

El operador | realiza una operación OR entre cada par de bits correspondientes de dos operandos. Actúa a nivel de bit dentro de una palabra completa.

A.1.1. Ejemplo 1: OR a nivel de bit (8 bits)

$$A = 01011010_2$$

$$B = 00110101_2$$

Aplicando OR bit a bit:

$$\begin{array}{r} 01011010 \\ 00110101 \\ \hline 01111111 \end{array}$$

$$A \,|\, B = 01111111_2$$

Cada posición binaria se evalúa de manera independiente. El bit resultante es 1 cuando al menos uno de los bits comparados es 1.

A.1.2. Ejemplo 2: OR a nivel de palabra (32 bits en C)

Listing A.1: Uso de OR bit a bit en C

```
1      #include <stdio.h>
2
3      int main() {
4        unsigned int permisos = 0x00000004;  // 00000000 00000000 00000000
   00000100
5        unsigned int escritura = 0x00000002; // 00000000 00000000 00000000
   00000010
6
7        permisos = permisos | escritura;
8
9        printf("%u\n", permisos);
10       return 0;
11     }
```

Resultado binario:

00000000 00000000 00000000 00000110

Aquí se activan simultáneamente dos banderas dentro de una palabra de 32 bits.

A.2. Operador OR lógico ||

El operador || es un operador lógico que evalúa expresiones completas y produce un valor booleano (0 o 1 en C/C++). No actúa bit a bit.

A.2.1. Ejemplo 1: Evaluación lógica con cortocircuito

Listing A.2: Uso de OR lógico en C

```c
#include <stdio.h>

int main() {
  int a = 5;
  int b = 0;

  if (a || b) {
    printf("Verdadero\n");
  }

  return 0;
}
```

Como a es distinto de 0, la expresión es verdadera y b no necesita evaluarse (cortocircuito).

A.2.2. Ejemplo 2: Diferencia práctica entre | y ||

Listing A.3: Comparación entre OR bit a bit y OR lógico

```c
#include <stdio.h>

int main() {
  int x = 2;   // 00000010
  int y = 1;   // 00000001

  int r1 = x | y;   // OR bit a bit
  int r2 = x || y;  // OR lógico

  printf("r1 = %d\n", r1);
```

```
11        printf("r2 = %d\n", r2);
12
13        return 0;
14      }
```

$$x|y = 00000011_2 = 3$$

$$x||y = 1$$

`|` conserva estructura binaria. `||` produce únicamente 0 (falso) o 1 (verdadero).

A.3. Operador AND `&`

El operador `&` realiza una operación AND bit a bit. El bit resultante es 1 únicamente cuando ambos bits son 1.

$$\begin{array}{r} 11001010 \\ 10101100 \\ \hline 10001000 \end{array}$$

Uso típico: verificación de bandera.

Listing A.4: Verificación de una bandera con AND

```
1       if (permisos & 0x04) {
2         // permiso activo
3       }
```

A.4. Operador XOR `^`

El operador `^` produce 1 cuando los bits comparados son diferentes.

$$\begin{array}{r} 1100 \\ 1010 \\ \hline 0110 \end{array}$$

Uso típico: alternar un estado.

Listing A.5: Alternar un bit con XOR

```
1       bandera = bandera ^ 0x01;
```

A.5. Desplazamientos

A.5.1. Desplazamiento a la izquierda <<

$$00000101 << 1 = 00001010$$

Equivale a multiplicar por 2 cuando no hay desbordamiento.

Listing A.6: Desplazamiento a la izquierda

```
1       unsigned int x = 5;  // 00000101
2       x = x << 1;          // 00001010
```

A.5.2. Desplazamiento a la derecha >>

$$00001010 >> 1 = 00000101$$

Equivale a dividir entre 2 para enteros sin signo.

Listing A.7: Desplazamiento a la derecha

```
1       unsigned int y = 10; // 00001010
2       y = y >> 1;          // 00000101
```

A.6. Resumen comparativo

Operador	Nivel de operación
`|`	Bit a bit (palabra completa)
`&`	Bit a bit
`^`	Bit a bit
`<< >>`	Bit a bit (desplazamiento)
`||`	Lógico (booleano)
`&&`	Lógico (booleano)

Importante

Las operaciones bit a bit modifican directamente la representación binaria almacenada en memoria. Los operadores lógicos evalúan expresiones completas y producen valores booleanos. Esta distinción permite diseñar estructuras compactas de control y validación dentro de sistemas computacionales.

Apéndice B

Codificación de caracteres: de ASCII a Unicode

La *codificación de caracteres* establece una correspondencia entre símbolos (letras, dígitos, signos, controles) y valores numéricos que un sistema digital puede almacenar, transmitir y procesar. En programación, este mapeo condiciona tareas tan diversas como validación de entradas, serialización de datos, interoperabilidad entre sistemas, normalización de texto y depuración de errores por interpretación de bytes. Por tal motivo, un aprendizaje riguroso de programación se beneficia cuando el estudiante distingue con precisión entre: (i) *repertorio* de caracteres (qué símbolos existen), (ii) *codificación* (cómo se traducen a números) y (iii) *codificación de bytes* (cómo esos números se representan en memoria y en archivos) [39], [40], [41].

Dentro de este panorama, ASCII se consolidó como base histórica de múltiples convenciones de software al definir un repertorio mínimo de 128 caracteres sobre 7 bits, suficiente para el alfabeto inglés y señales de control heredadas de teletipos. No obstante, la evolución hacia lenguas con acentos, símbolos monetarios y tipografías diversas motivó extensiones de 8 bits (*code pages*) y, posteriormente, la adopción de Unicode como estándar global. Esta progresión explica por qué, en sistemas actuales, UTF-8 domina como codificación de intercambio: mantiene compatibilidad hacia atrás con ASCII en el rango 0–127 y habilita la representación de prácticamente todos los sistemas de escritura [39], [41], [42].

B.0.1. ASCII (7 bits): repertorio base 0–127

ASCII define 128 posiciones numeradas de 0 a 127. Las posiciones 0–31 y 127 se reservan para caracteres de control (no imprimibles), mientras que 32–126 representan caracteres

imprimibles (espacio, signos, dígitos y letras). La tabla siguiente presenta el mapeo completo, expresado en decimal y hexadecimal, destacando el nombre convencional de los controles [40], [43].

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
0	00		NUL	64	40	@	Car. Arroba
1	01		SOH	65	41	A	A
2	02		STX	66	42	B	B
3	03		ETX	67	43	C	C
4	04		EOT	68	44	D	D
5	05		ENQ	69	45	E	E
6	06		ACK	70	46	F	F
7	07		BEL	71	47	G	G
8	08		BS	72	48	H	H
9	09		HT	73	49	I	I
10	0A		LF	74	4A	J	J
11	0B		VT	75	4B	K	K
12	0C		FF	76	4C	L	L
13	0D		CR	77	4D	M	M
14	0E		SO	78	4E	N	N
15	0F		SI	79	4F	O	O
16	10		DLE	80	50	P	P
17	11		DC1	81	51	Q	Q
18	12		DC2	82	52	R	R
19	13		DC3	83	53	S	S
20	14		DC4	84	54	T	T
21	15		NAK	85	55	U	U
22	16		SYN	86	56	V	V
23	17		ETB	87	57	W	W
24	18		CAN	88	58	X	X
25	19		EM	89	59	Y	Y
26	1A		SUB	90	5A	Z	Z
27	1B		ESC	91	5B	[	Corchete izquierdo

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
28	1C		FS	92	5C	\	Barra invertida
29	1D		GS	93	5D	]	Corchete derecho
30	1E		RS	94	5E		ˆ Acento circunflejo
31	1F		US	95	5F	_	Guion bajo
32	20		Espacio	96	60	‘	Acento grave
33	21	!	Signo de exclamación	97	61	a	a
34	22	”	Comillas dobles	98	62	b	b
35	23	#	Signo numeral	99	63	c	c
36	24	$	Signo dólar	100	64	d	d
37	25	%	Signo de porcentaje	101	65	e	e
38	26	&	Ampersand	102	66	f	f
39	27	’	Apóstrofo	103	67	g	g
40	28	(	Paréntesis izquierdo	104	68	h	h
41	29	)	Paréntesis derecho	105	69	i	i
42	2A	*	Asterisco	106	6A	j	j
43	2B	+	Signo más	107	6B	k	k
44	2C	,	Coma	108	6C	l	l
45	2D	-	Guion	109	6D	m	m
46	2E	.	Punto	110	6E	n	n
47	2F	/	Barra diagonal	111	6F	o	o
48	30	0	0	112	70	p	p
49	31	1	1	113	71	q	q
50	32	2	2	114	72	r	r
51	33	3	3	115	73	s	s
52	34	4	4	116	74	t	t
53	35	5	5	117	75	u	u
54	36	6	6	118	76	v	v
55	37	7	7	119	77	w	w
56	38	8	8	120	78	x	x
57	39	9	9	121	79	y	y
58	3A	:	Dos puntos	122	7A	z	z

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
59	3B	;	Punto y coma	123	7B	{	Llave izquierda
60	3C	<	Signo menor que	124	7C	|	Barra vertical
61	3D	=	Signo igual	125	7D	}	Llave derecha
62	3E	>	Signo mayor que	126	7E	~	Virgulilla
63	3F	?	Signo de interrogación	127	7F		Suprimir (DEL)

La tabla ASCII resulta indispensable para lectura de archivos, protocolos y depuración, debido a que múltiples formatos interpretan bytes 0–127 como ASCII. Paralelamente, el estudiante adquiere un criterio operativo: una cadena es, en última instancia, una secuencia de códigos, y el comportamiento de comparaciones, ordenamiento y validaciones depende de esa codificación [39], [43].

Extensiones de 8 bits: por qué “ASCII extendido” no es una tabla única

En la práctica se usa la expresión “ASCII extendido” para referirse a codificaciones de 8 bits que preservan ASCII en 0–127 y asignan caracteres a 128–255. No obstante, ese bloque superior depende de una tabla específica. Dos referencias ampliamente citadas son: (i) ISO/IEC 8859-1 (Latin-1), utilizada históricamente en sistemas UNIX y protocolos antiguos; y (ii) Windows-1252, extensión dominante en entornos Windows que asigna símbolos tipográficos a 0x80–0x9F (posición que en ISO/IEC 8859-1 se reserva para controles) [39], [42], [44].

Tabla completa ISO/IEC 8859-1 (Latin-1) para 128–255

En ISO/IEC 8859-1, el rango 0x80–0x9F corresponde a controles C1. El rango 0xA0–0xFF contiene símbolos y letras latinas acentuadas.

Cuadro B.2: Códigos ASCII extendidos (128–255) según ISO-8859-1

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
128	80		Control	192	C0	À	A grave
129	81		Control	193	C1	Á	A aguda
130	82		Control	194	C2	Â	A circunfleja
131	83		Control	195	C3	Ã	A tilde
132	84		Control	196	C4	Ä	A diéresis
133	85		Control	197	C5	Å	A anillo

Continúa en la siguiente página

Cuadro B.2 (continuación)

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
134	86		Control	198	C6	Æ	AE
135	87		Control	199	C7	Ç	C cedilla
136	88		Control	200	C8	È	E grave
137	89		Control	201	C9	É	E aguda
138	8A		Control	202	CA	Ê	E circunfleja
139	8B		Control	203	CB	Ë	E diéresis
140	8C		Control	204	CC	Ì	I grave
141	8D		Control	205	CD	Í	I aguda
142	8E		Control	206	CE	Î	I circunfleja
143	8F		Control	207	CF	Ï	I diéresis
144	90		Control	208	D0	Ð	Eth mayúscula
145	91		Control	209	D1	Ñ	N con tilde
146	92		Control	210	D2	Ò	O grave
147	93		Control	211	D3	Ó	O aguda
148	94		Control	212	D4	Ô	O circunfleja
149	95		Control	213	D5	Õ	O tilde
150	96		Control	214	D6	Ö	O diéresis
151	97		Control	215	D7	×	Multiplicación
152	98		Control	216	D8	Ø	O tachada
153	99		Control	217	D9	Ù	U grave
154	9A		Control	218	DA	Ú	U aguda
155	9B		Control	219	DB	Û	U circunfleja
156	9C		Control	220	DC	Ü	U diéresis
157	9D		Control	221	DD	Ý	Y aguda
158	9E		Control	222	DE	Þ	Thorn mayúscula
159	9F		Control	223	DF	ß	Eszett
160	A0		Espacio no separable	224	E0	à	a grave
161	A1	¡	Exclamación invertida	225	E1	á	a aguda
162	A2	¢	Centavo	226	E2	â	a circunfleja

Continúa en la siguiente página

Cuadro B.2 (continuación)

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
163	A3	£	Libra esterlina	227	E3	ã	a tilde
164	A4	¤	Moneda	228	E4	ä	a diéresis
165	A5	¥	Yen	229	E5	å	a anillo
166	A6	¦	Barra partida	230	E6	æ	ae
167	A7	§	Sección	231	E7	ç	c cedilla
168	A8	¨	Diéresis	232	E8	è	e grave
169	A9	©	Copyright	233	E9	é	e aguda
170	AA	ª	Ordinal femenino	234	EA	ê	e circunfleja
171	AB	«	Comillas angulares izq.	235	EB	ë	e diéresis
172	AC	¬	Negación	236	EC	ì	i grave
173	AD		Guion suave	237	ED	í	i aguda
174	AE	®	Marca registrada	238	EE	î	i circunfleja
175	AF	¯	Macrón	239	EF	ï	i diéresis
176	B0	°	Grado	240	F0	ð	eth minúscula
177	B1	±	Más/menos	241	F1	ñ	n con tilde
178	B2	²	Superíndice 2	242	F2	ò	o grave
179	B3	³	Superíndice 3	243	F3	ó	o aguda
180	B4	´	Acento agudo	244	F4	ô	o circunfleja
181	B5	µ	Micro	245	F5	õ	o tilde
182	B6	¶	Párrafo	246	F6	ö	o diéresis
183	B7	·	Punto medio	247	F7	÷	División
184	B8	¸	Cedilla	248	F8	ø	o tachada
185	B9	¹	Superíndice 1	249	F9	ù	u grave
186	BA	º	Ordinal masculino	250	FA	ú	u aguda
187	BB	»	Comillas angulares der.	251	FB	û	u circunfleja
188	BC	¼	Un cuarto	252	FC	ü	u diéresis
189	BD	½	Un medio	253	FD	ý	y aguda
190	BE	¾	Tres cuartos	254	FE	þ	thorn minúscula
191	BF	¿	Interrogación invertida	255	FF	ÿ	y diéresis

Tabla completa Windows-1252 para 128–255

Windows-1252 mantiene ASCII en 0–127, pero asigna caracteres tipográficos a 0x80–0x9F. Esta distinción resulta indispensable en programación al interpretar archivos de texto "ANSI**15**" de Windows que contienen comillas curvas, guiones largos o el símbolo del euro [39], [44].

Cuadro B.3: Códigos 128–255 según Windows-1252 (CP1252)

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
128	80	€	Euro	192	C0	À	A grave
129	81		No asignado	193	C1	Á	A aguda
130	82	‚	Comilla baja simple	194	C2	Â	A circunfleja
131	83	f	Florín	195	C3	Ã	A tilde
132	84	„	Comillas bajas dobles	196	C4	Ä	A diéresis
133	85	…	Puntos suspensivos	197	C5	Å	A anillo
134	86	†	Daga	198	C6	Æ	AE
135	87	‡	Daga doble	199	C7	Ç	C cedilla
136	88	ˆ	Acento circunflejo (diacrítico)	200	C8	È	E grave
137	89	‰	Por mil	201	C9	É	E aguda
138	8A	Š	S carón (mayúscula)	202	CA	Ê	E circunfleja
139	8B	‹	Comilla angular izq.	203	CB	Ë	E diéresis
140	8C	Œ	OE (mayúscula)	204	CC	Ì	I grave
141	8D		No asignado	205	CD	Í	I aguda
142	8E	Ž	Z carón (mayúscula)	206	CE	Î	I circunfleja
143	8F		No asignado	207	CF	Ï	I diéresis
144	90		No asignado	208	D0	Ð	Eth (mayúscula)
145	91	‘	Comilla simple izquierda	209	D1	Ñ	N con tilde
146	92	’	Comilla simple derecha	210	D2	Ò	O grave
147	93	“	Comilla doble izquierda	211	D3	Ó	O aguda
148	94	”	Comilla doble derecha	212	D4	Ô	O circunfleja
149	95	•	Viñeta	213	D5	Õ	O tilde
150	96	–	Raya corta (en dash)	214	D6	Ö	O diéresis
151	97	—	Raya larga (em dash)	215	D7	×	Multiplicación

Continúa en la siguiente página

Cuadro B.3 (continuación)

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
152	98	˜	Tilde (diacrítico)	216	D8	Ø	O tachada
153	99	™	Marca registrada (TM)	217	D9	Ù	U grave
154	9A	š	s carón (minúscula)	218	DA	Ú	U aguda
155	9B	›	Comilla angular der.	219	DB	Û	U circunfleja
156	9C	œ	oe (minúscula)	220	DC	Ü	U diéresis
157	9D		No asignado	221	DD	Ý	Y aguda
158	9E	ž	z carón (minúscula)	222	DE	Þ	Thorn (mayúscula)
159	9F	Ÿ	Y diéresis	223	DF	ß	Eszett
160	A0		Espacio no separable	224	E0	à	a grave
161	A1	¡	Exclamación invertida	225	E1	á	a aguda
162	A2	¢	Centavo	226	E2	â	a circunfleja
163	A3	£	Libra esterlina	227	E3	ã	a tilde
164	A4	¤	Signo monetario	228	E4	ä	a diéresis
165	A5	¥	Yen	229	E5	å	a anillo
166	A6	¦	Barra partida	230	E6	æ	ae
167	A7	§	Sección	231	E7	ç	c cedilla
168	A8	¨	Diéresis	232	E8	è	e grave
169	A9	©	Copyright	233	E9	é	e aguda
170	AA	ª	Ordinal femenino	234	EA	ê	e circunfleja
171	AB	«	Comillas angulares izq.	235	EB	ë	e diéresis
172	AC	¬	Negación	236	EC	ì	i grave
173	AD		Guion suave	237	ED	í	i aguda
174	AE	®	Marca registrada	238	EE	î	i circunfleja
175	AF	¯	Macrón	239	EF	ï	i diéresis
176	B0	°	Grado	240	F0	ð	eth (minúscula)
177	B1	±	Más/menos	241	F1	ñ	n con tilde
178	B2	²	Superíndice 2	242	F2	ò	o grave
179	B3	³	Superíndice 3	243	F3	ó	o aguda
180	B4	´	Acento agudo	244	F4	ô	o circunfleja

Continúa en la siguiente página

Cuadro B.3 (continuación)

Dec	Hex	Car.	Nombre	Dec	Hex	Car.	Nombre
181	B5	µ	Micro	245	F5	õ	o tilde
182	B6	¶	Párrafo	246	F6	ö	o diéresis
183	B7	·	Punto medio	247	F7	÷	División
184	B8	¸	Cedilla	248	F8	ø	o tachada
185	B9	¹	Superíndice 1	249	F9	ù	u grave
186	BA	º	Ordinal masculino	250	FA	ú	u aguda
187	BB	»	Comillas angulares der.	251	FB	û	u circunfleja
188	BC	¼	Un cuarto	252	FC	ü	u diéresis
189	BD	½	Un medio	253	FD	ý	y aguda
190	BE	¾	Tres cuartos	254	FE	þ	thorn (minúscula)
191	BF	¿	Interrogación invertida	255	FF	ÿ	y diéresis

En síntesis, cuando un programa recibe un byte con valor 0x93, el símbolo resultante depende de la tabla: en Windows-1252 corresponde a comillas tipográficas (“), mientras que en ISO/IEC 8859-1 ese valor se reserva para control C1. Esta diferencia explica fallas de interoperabilidad, aparición de caracteres “extraños” y necesidad de conversiones explícitas en lectura de archivos [39], [44].

Unicode y UTF-8: estándar global para texto

Unicode define un repertorio amplio de caracteres (códigos U+XXXX) y asigna puntos de código a letras, símbolos, emoji y escrituras completas. En la práctica, UTF-8 codifica esos puntos de código en secuencias de 1 a 4 bytes: para U+0000 a U+007F coincide con ASCII, lo cual facilita compatibilidad con software heredado. Este diseño sugiere una práctica de programación recomendada: conservar UTF-8 como codificación por defecto en archivos y comunicaciones, y convertir a tablas de 8 bits únicamente cuando el contexto lo exija (por ejemplo, archivos legacy) [39], [41].

De manera complementaria, UTF-16 y UTF-32 existen para necesidades específicas: UTF-16 usa unidades de 16 bits (con pares sustitutos en rangos superiores) y UTF-32 usa 32 bits fijos. En términos pedagógicos, la distinción más útil al inicio es que Unicode separa “caracter” (punto de código) de “cómo se almacena” (UTF-8/UTF-16/UTF-32). Esa idea permite al estudiante razonar con rigor sobre longitudes de cadenas, índices, conteos de bytes y validaciones de entrada [39], [45].

Codificación Unicode: UTF-8, UTF-16 y UTF-32

Unicode define un repertorio universal de caracteres asignando a cada uno un **punto de código** único en el rango U+0000 hasta U+10FFFF. Esta asignación está normada por el estándar ISO/IEC 10646 [46] y especificada operativamente en distintos esquemas de codificación, entre ellos UTF-8, UTF-16 y UTF-32 [39], [41].

1. Codificación UTF-8. Es una codificación de longitud variable (1 a 4 bytes) compatible hacia atrás con ASCII.

Estructura general

Rango Unicode	Patrón binario UTF-8
U+0000–007F	0xxxxxxx
U+0080–07FF	110xxxxx 10xxxxxx
U+0800–FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+10000–10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Diagrama visual de bits. Ejemplo: carácter "ñ" (U+00F1)

$$U + 00F1 = 00000000\ 11110001_2$$

Distribución en UTF-8 (2 bytes):

$$\underbrace{11000011}_{\text{Byte 1}} \quad \underbrace{10110001}_{\text{Byte 2}}$$

En hexadecimal:

$$C3\ B1$$

Ejemplo con 4 bytes. Carácter: 😁 (U+1F600)

$$1F600_{16} = 0001\ 1111\ 0110\ 0000\ 0000_2$$

Distribución en patrón UTF-8:

$$\underbrace{11110000}_{F0} \quad \underbrace{10011111}_{9F} \quad \underbrace{10011000}_{98} \quad \underbrace{10000000}_{80}$$

Resultado:

$$F0\ 9F\ 98\ 80$$

2. Codificación UTF-16. Este tipo de codificación utiliza unidades de 16 bits.

Caso simple. Para puntos U+0000--U+FFFF:

$$\text{``ñ''} = 00F1_{16}$$

En memoria:

Big-endian:

$$00\ F1$$

Little-endian:

$$F1\ 00$$

Pares sustitutos (Surrogates). Para puntos mayores que U+FFFF:

Carácter 😜 (U+1F600)

1. Restar 0x10000:

$$1F600 - 10000 = F600$$

2. Dividir en dos bloques de 10 bits.

3. Calcular:

$$\text{High surrogate} = D800 + \text{parte alta}$$

$$\text{Low surrogate} = DC00 + \text{parte baja}$$

4. Resultado:

$$D83D\ DE00$$

En memoria:

Big-endian:

$$D8\ 3D\ DE\ 00$$

Little-endian:

$$3D\ D8\ 00\ DE$$

3. Codificación UTF-32. Almacena directamente el punto de código en 32 bits.

Ejemplo:

$$\text{``ñ''} = 000000F1_{16}$$

En memoria:

Big-endian:

$$00\ 00\ 00\ F1$$

Little-endian:

$$F1\ 00\ 00\ 00$$

No requiere transformación ni pares sustitutos.

4. Diagramas comparativos de almacenamiento

Carácter	UTF-8	UTF-16
ñ	C3 B1	00 F1
😁	F0 9F 98 80	D8 3D DE 00

5. Problemas de codificación: Mojibake. El fenómeno denominado mojibake ocurre cuando una secuencia de bytes se interpreta con una codificación distinta a la original.

Ejemplo clásico:

La palabra ``Señor'' en UTF-8:

$$53\ 65\ C3\ B1\ 6F\ 72$$

Si esos bytes se interpretan como ISO-8859-1, se obtiene:

$$Se\pm or$$

Esto ocurre porque:

- C3 B1 es correcto en UTF-8 para ``ñ''.
- ISO-8859-1 interpreta C3 y B1 como caracteres independientes.

El resultado es una corrupción visual del texto.

6. Consideraciones normativas. UTF-8 está definido formalmente en RFC 3629 [41]. Unicode y su mapeo con ISO/IEC 10646 están regulados por ISO/IEC 10646 [46].

Las diferencias fundamentales son:

- UTF-8: eficiente y dominante en Internet.
- UTF-16: usado históricamente en Windows y Java.

- UTF-32: representación directa, mayor consumo de memoria.

Todas representan exactamente el mismo conjunto Unicode.

Bibliografía

[1] K. E. Kendall y J. E. Kendall, *Systems Analysis and Design*, 11.ª ed. Pearson, 2024, isbn: 9780137947850.

[2] K. C. Laudon y J. P. Laudon, *Management Information Systems: Managing the Digital Firm*, 16.ª ed. Pearson, 2020, isbn: 978-0135191816.

[3] F. M. González-Longatt, «Introducción a los Sistemas de Información: Fundamentos,» *Manuscrito (Universidad Experimental Politécnica de la Fuerza Armada, Venezuela)*, pág. 7, 2007, pdf disponible en línea. dirección: `https://www.uv.mx/personal/artulopez/files/2012/08/fundamentossistemasinformacion.pdf`

[4] C. Ghezzi, M. Jazayeri y D. Mandrioli, *Fundamentals of Software Engineering*, 2.ª ed. Prentice Hall, 2003, isbn: 978-0133056990.

[5] J. R. Hanly y E. B. Koffman, *Problem Solving and Program Design in C*, 8.ª ed. Boston, MA: Pearson, 2015, isbn: 978-1292098814.

[6] ISO/IEC, *Programming languages — Avoiding vulnerabilities in programming languages Part 1: Language-independent catalogue of vulnerabilities*, Geneva, Switzerland: ISO/IEC, 2024. dirección: `https://www.iso.org/standard/83629.html`

[7] T. Gaddis, *Starting Out with Programming Logic and Design*, 6th. Boston, MA: Pearson, 2022, isbn: 978-0137602148.

[8] S. Calzati, «An Ecosystemic View on Information, Data, and Knowledge: Insights on Agential AI and Relational Ethics,» *AI and Ethics*, vol. 5, págs. 3763-3776, 2025. doi: `10.1007/s43681-025-00665-0`

[9] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 11.ª ed. London: Pearson, 2021, pág. 896, isbn: 978-1292420103.

[10] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating System Concepts*, 10.ª ed. Hoboken, NJ: Wiley, 2018, pág. 1040, isbn: 978-1119320913.

[11] IEEE, «IEEE Standard for Floating-Point Arithmetic,» *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, págs. 1-84, 2019. doi: 10.1109/IEEESTD.2019.8766229

[12] D. Goldberg, «What Every Computer Scientist Should Know About Floating-Point Arithmetic,» *ACM Computing Surveys*, vol. 23, n.º 1, págs. 5-48, 1991. doi: 10.1145/103162.103163

[13] N. Wirth, *Algorithms + Data Structures = Programs*, 1.ª ed. Prentice Hall, 1976, pág. 366, Última Actualización Moscow, 2 de febrero 2012-02-22 por Fyodor Tkachov, isbn: 978-0130224187.

[14] P. Leidig, G. Anderson, R. Sooriamurthi y J. Babb, «IS2020: Updating the Information Systems Model Curriculum,» en *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ép. SIGCSE '20, Portland, OR, USA: Association for Computing Machinery, 2020, págs. 803-804, isbn: 9781450367936. doi: 10.1145/3328778.3366999

[15] P. Leidig y H. Salmela, «IS2020: Competency Model for Undergraduate Programs in Information Systems: The Joint ACM/AIS IS2020 Task Force,» 2021. doi: 10.1145/3460863

[16] D. P. Tegarden, B. Samuel, R. Lukyanenko, A. Dennis y B. H. Wixom, *Systems Analysis and Design: An Object-Oriented Approach with UML*, 7.ª ed. Wiley, 2025, isbn: 978-1394331765.

[17] J. M. Wing, «Computational Thinking,» *Communications of the ACM*, vol. 49, n.º 3, págs. 33-35, 2006. doi: 10.1145/1118178.1118215

[18] L. Perković, A. Settle, S. Hwang y J. Jones, «A framework for computational thinking across the curriculum,» en *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ép. ITiCSE '10, Bilkent, Ankara, Turkey: Association for Computing Machinery, 2010, págs. 123-127, isbn: 978-1605588209. doi: 10.1145/1822090.1822126

[19] G. Futschek, «Algorithmic Thinking: The Key for Understanding Computer Science,» en *Informatics Education – The Bridge between Using and Understanding Computers*, ép. Lecture Notes in Computer Science, vol. 4226, Springer, 2006, págs. 159-168. doi: 10.1007/11915355_15

[20] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen y K. A. Houston, *Object-Oriented Analysis and Design with Applications*, 3.ª ed. Addison-Wesley, 2007, pág. 720, isbn: 978-0201895513.

[21] R. Fitzpatrick, «Software quality revisited,» English, en *Proceedings of the Software Measurement European Forum (SMEF)*, ép. Proceedings of the Software Measurement European Forum (SMEF), Milan, Italy: Instituto di Ricerca Internazionale, 2004, págs. 305-315. doi: 10.21427/e2gf-0035

[22] J. H. Bernstein, «The Data–Information–Knowledge–Wisdom Hierarchy and Its Antithesis,» *Journal of Management and Information Systems*, vol. 26, n.º 2, págs. 187-220, 2009, análisis crítico del modelo tradicional DIKW, discutiendo sus límites y proporcionando una visión más realista sobre cómo la información puede (o no) transformarse en conocimiento. doi: 10.2753/MIS0742-1222260206

[23] R. S. Pressman y B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 9.ª ed. New York: McGraw-Hill Education, 2019, pág. 704, isbn: 978-1260548006.

[24] I. Sommerville, *Software Engineering*, 10.ª ed. Harlow, England: Pearson, 2015, pág. 816, obra clásica que define el rol integral del ingeniero de software, desde requisitos hasta mantenimiento, isbn: 978-0133943030.

[25] K. D. Cato, K. McGrow y S. C. Rossetti, «Transforming Clinical Data Into Wisdom: Artificial Intelligence Implications for Nurse Leaders,» *Nursing Management (Springhouse)*, vol. 51, n.º 11, págs. 24-30, 2020. doi: 10.1097/01.NUMA.0000719396.83518.d6

[26] A. Dix, J. Finlay, G. Abowd y R. Beale, *Human-Computer Interaction*, 3.ª ed. Pearson/Prentice-Hall, 2004, pág. 834, isbn: 978-0130461094.

[27] C. J. Date, *An Introduction to Database Systems*, 8.ª ed. Addison-Wesley, 2004, Edición clásica aún ampliamente citada; reimpresiones posteriores, isbn: 978-0321197849.

[28] R. Elmasri y S. B. Navathe, *Fundamentals of Database Systems*, 7th. Pearson, 2016, isbn: 978-0133970777.

[29] A. Silberschatz, H. F. Korth y S. Sudarshan, *Database System Concepts*, 7th. McGraw-Hill Education, 2020, isbn: 978-0078022159.

[30] A. Martín-Navarro, M. P. Lechuga Sancho y J. A. Medina-Garrido, «BPMS for management: a systematic literature review,» *Revista Espanola de Documentacion Cientifica*, vol. 41, n.º 3, 2018, n. art. e213, revisión sistemática reciente sobre los sistemas de gestión de procesos de negocio (BPMS) como tecnología que automatiza procesos organizacionales. doi: 10.3989/redc.2018.3.1532

[31] M. J. Medina, A. D. Santo, P. Oswald y M. Sokhn, «Digital Business Transformation for SMEs: Maturity Model for Systematic Roadmap,» en *Information Systems and Technologies*, ép. Lecture Notes in Networks and Systems, Á. Rocha, H. Adeli, L. P. Reis y S. Costanzo, eds., vol. 801, Springer, Cham, 2024, págs. 229-240. doi: 10.1007/978-3-031-45648-0_23

[32] A.-A. Vărzaru y otros, «Digital Transformation and Innovation: The Influence of Digital Technologies on Turnover from Innovation Activities and Types of Innovation,» *Systems*, vol. 12, n.º 9, 2024, n. art. 359, analiza cómo las tecnologías digitales, habilitadas por software, impulsan innovación y ventajas competitivas. doi: 10.3390/systems12090359

[33] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1.ª ed. Upper Saddle River, NJ: Prentice Hall, 2008, isbn: 9780132350884.

[34] A. S. Tanenbaum y T. Austin, *Structured Computer Organization*, 6.ª ed. Boston, MA: Pearson, 2021, pág. 777, isbn: 978-0137618446.

[35] R. W. Sebesta, *Concepts of Programming Languages*, 12.ª ed. Pearson, 2018, Update 2018, isbn: 978-0134997186.

[36] R. Sedgewick y K. Wayne, *Algorithms*, 4.ª ed. Addison-Wesley, 2011, pág. 976, isbn: 978-0321573513.

[37] J. L. Manzano, *Programación: Algoritmos y estructuras de datos*. Paraninfo, 2017.

[38] S. C. Chapra y R. P. Canale, *Numerical Methods for Engineers*, 7.ª ed. New York, NY: McGraw-Hill Education, 2014, isbn: 978-0073397924.

[39] The Unicode Consortium, *The Unicode® Standard: Core Specification*, ver. 17, South San Francisco, CA, 2025. visitado 27 de ene. de 2026. dirección: https://www.unicode.org/versions/Unicode17.0.0/core-spec/

[40] ISO/IEC. «ISO/IEC 646:1991: Information technology — ISO 7-bit coded character set for information interchange.» Confirmed and reviewed in 2020, visitado 25 de feb. de 2026. dirección: https://www.iso.org/standard/4777.html

[41] F. Yergeau, «RFC 3629: UTF-8, a transformation format of ISO 10646,» RFC Editor, Internet Standard 3629, nov. de 2003. doi: 10.17487/RFC3629 visitado 25 de feb. de 2026. dirección: https://www.rfc-editor.org/rfc/rfc3629

[42] *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, Latin-1 mapping, 0–255 with C0/C1 controls, International Organization for Standardization, 1998. dirección: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:8859:-1:ed-1:v1:en

[43] V. Cerf, *ASCII format for Network Interchange*, RFC - Internet Standard, ver. RFC 20, Superseded by INCITS 4-1986 (R2012), American National Standards Institute, 1969. dirección: https://datatracker.ietf.org/doc/html/rfc20

[44] Microsoft. «Code Page Identifiers,» visitado 25 de feb. de 2026. dirección: https://learn.microsoft.com/en-us/windows/win32/intl/code-page-identifiers

[45] P. Hoffman y F. Yergeau, «RFC 2781: UTF-16, an encoding of ISO 10646,» RFC Editor, Informational 2781, feb. de 2000. doi: 10.17487/RFC2781 visitado 25 de feb. de 2026.

[46] ISO/IEC, *Information technology — Universal coded character set (UCS)*, ISO/IEC, 2020. visitado 27 de ene. de 2026. dirección: https://www.iso.org/standard/76835.html