1.1 Computación Análoga

Objetivo: Comprender qué es la computación análoga, cómo funciona y sus principales usos.

Definición

- Modelo de computación que procesa información mediante magnitudes físicas continuas (voltaje, presión o movimiento).
- No trabaja con ceros y unos, sino con valores que cambian de manera constante dentro de un rango.

Características

- Usa dispositivos mecánicos, eléctricos o hidráulicos.
- Ofrece resultados aproximados.
- Muy usada antes de la aparición de las computadoras digitales.

Ejemplos

- Regla de cálculo para operaciones matemáticas.
- Relojes de manecillas.
- Termómetros de mercurio.

1.2 Introducción a métodos digitales

Objetivo: Comprender qué son los métodos digitales, cómo funcionan y su relevancia en la computación moderna y la ciencia de la información.

Definición

- Conjunto de técnicas y procedimientos que permiten representar, procesar, analizar y manipular información utilizando sistemas discretos.
- Convierte datos del mundo real en valores discretos (generalmente binarios) para su almacenamiento, transmisión y procesamiento mediante computadoras.
- Base de la computación digital, presente en smartphones, sistemas de control industrial, análisis de datos clínicos y prácticamente toda tecnología moderna.

Características

- Trabaja con información discreta en lugar de continua, usando ceros y unos como lenguaje fundamental.
- Permite precisión, reproducibilidad y facilidad de manipulación de datos.
- Capacidad de aplicar algoritmos complejos para resolver problemas que serían difíciles o imposibles de abordar con métodos análogos.
- Facilita el almacenamiento, la comunicación y el control de información de manera confiable y eficiente.
- Sirve de soporte a simulaciones, modelado y análisis de datos en diversas áreas del conocimiento.

1.3 Descripción de componentes

Objetivo: Conocer los principales componentes de un sistema digital, su funcionamiento y cómo interactúan para procesar información.

Definición

- Los componentes de un sistema digital son los elementos físicos y lógicos que permiten la representación, procesamiento, almacenamiento y comunicación de información en forma discreta.
- Incluyen tanto hardware (circuitos, procesadores, memoria, periféricos) como software (sistemas operativos y programas) que controlan el flujo y la manipulación de datos.
- Son la base de cualquier dispositivo digital moderno, desde microcontroladores hasta supercomputadoras.

Principales Componentes

- **Unidad Central de Procesamiento (CPU):** Componente encargado de ejecutar instrucciones, procesar datos y coordinar el funcionamiento de todo el sistema.
- **Memoria:** Almacena datos e instrucciones de manera temporal (RAM) o permanente (ROM, discos, memorias flash).
- **Dispositivos de Entrada:** Permiten la interacción con el sistema, como teclados, sensores, micrófonos y cámaras.
- **Dispositivos de Salida:** Permiten mostrar o transmitir información procesada, como pantallas, impresoras o actuadores en sistemas de control.
- **Bus de Datos:** Conjunto de líneas que transportan información entre la CPU, memoria y periféricos.
- **Controladores y Interfaces:** Administran la comunicación con dispositivos externos y aseguran que los datos se transfieran correctamente.

- **Software de Sistema:** Sistemas operativos y firmware que gestionan recursos y permiten ejecutar aplicaciones.

Características

- Interconexión eficiente entre componentes, permitiendo un flujo de información rápido y confiable.
- Capacidad de procesamiento secuencial y paralelo mediante CPU y otros procesadores especializados.
- Almacenamiento jerárquico, que permite acceso rápido a datos frecuentes y conservación a largo plazo de información crítica.
- Modularidad, facilitando la actualización y sustitución de componentes sin afectar el funcionamiento general.
- Compatibilidad entre hardware y software para ejecutar algoritmos y procesos de manera correcta.

Ejemplos

- Un microcontrolador Arduino que integra CPU, memoria y puertos de entrada/salida para proyectos electrónicos.
- Computadoras personales con CPU, RAM, disco duro, tarjeta gráfica y periféricos conectados mediante buses.
- Servidores de datos que utilizan múltiples CPUs, memorias de alta velocidad y sistemas redundantes para garantizar disponibilidad.
- Smartphones que combinan procesador, memoria, sensores y sistemas operativos móviles en un solo dispositivo compacto.

1.4 Lectura de entrada

Objetivo: Entender cómo los sistemas digitales reciben información del mundo exterior y la preparan para su procesamiento interno.

Definición

- La lectura de entrada se refiere al proceso mediante el cual un sistema digital captura datos desde dispositivos o sensores externos para ser procesados por la CPU.
- Convierte señales físicas (como presión, luz, sonido o movimiento) en información discreta que puede ser entendida y manipulada por el sistema.
- Es el primer paso para la interacción entre el mundo real y los sistemas digitales, permitiendo la automatización, control y análisis de datos.

Principales Dispositivos de Entrada

- **Teclados y ratones:** Permiten ingresar datos y comandos de forma directa.
- **Sensores:** Capturan información física como temperatura, presión, luz o humedad.
- **Micrófonos y cámaras:** Digitalizan señales de audio y video para procesamiento y almacenamiento.
- **Lectores de código:** Incluyen códigos de barras y QR que convierten información codificada en datos legibles por el sistema.
- **Dispositivos táctiles y pantallas interactivas:** Permiten la interacción directa con interfaces gráficas y aplicaciones.
- **Interfaces de red:** Reciben información proveniente de otros sistemas digitales a través de protocolos de comunicación.

Características

- Transforma señales analógicas o físicas en información digital interpretable.
- Precisión y fidelidad en la captura de datos para asegurar un procesamiento confiable.
- Velocidad de respuesta suficiente para mantener sincronización con el procesamiento interno.
- Compatibilidad con los estándares de comunicación y formatos de datos del sistema.
- Posibilidad de integración con múltiples dispositivos simultáneamente para sistemas complejos.

Ejemplos

- Un sensor de temperatura conectado a un microcontrolador para controlar un sistema de climatización.
- La cámara de un smartphone que captura imágenes y video para aplicaciones de reconocimiento facial.
- Lectores de tarjetas RFID que permiten el acceso seguro a instalaciones o sistemas electrónicos.
- Teclados y pantallas táctiles en cajeros automáticos que registran la información del usuario.
- Micrófonos digitales que capturan audio para asistentes de voz y sistemas de dictado.

1.5 Unidad de memoria

Objetivo: Comprender la función, tipos y características de la memoria en un sistema computacional.

Definición

- La unidad de memoria es el componente del computador responsable de almacenar datos e instrucciones para su posterior procesamiento.
- Actúa como un almacén temporal o permanente de información que puede ser accedida rápidamente por la CPU.
- Es fundamental para el funcionamiento de cualquier sistema digital, ya que determina la capacidad de almacenamiento y la velocidad de acceso a la información.

Tipos de Memoria

- **Memoria Principal (RAM):** Memoria de acceso aleatorio volátil que almacena temporalmente datos y programas en ejecución.
- **Memoria ROM:** Memoria de solo lectura no volátil que contiene instrucciones fundamentales para el arranque del sistema.
- **Memoria Caché:** Memoria de alta velocidad que almacena copias de datos frecuentemente accedidos para acelerar el procesamiento.
- **Memoria Virtual:** Extensión de la memoria principal que utiliza espacio en el almacenamiento secundario.
- **Memoria Flash:** Memoria no volátil utilizada en dispositivos de almacenamiento como SSDs y unidades USB.

Características

- **Capacidad:** Cantidad de información que puede almacenar, medida en bytes (KB, MB, GB, TB).
- **Velocidad de acceso:** Tiempo que tarda la CPU en acceder a los datos almacenados.
- **Volatilidad:** Determina si la información se mantiene o se pierde al apagar el sistema.
- **Método de acceso:** Forma en que se accede a la información (aleatorio, secuencial, directo).
- **Tecnología:** Implementación física de la memoria (semiconductores, magnética, óptica).

Ejemplos

- Los módulos DDR4 en computadoras modernas que permiten multitarea eficiente.
- La memoria ROM BIOS que contiene el firmware necesario para iniciar el sistema operativo.
- Memoria caché L3 en procesadores Intel Core i7 que acelera el acceso a datos frecuentes.
- Memoria flash en tarjetas SD utilizadas en cámaras digitales y smartphones.
- Memoria virtual implementada en el disco duro para extender la capacidad de la RAM.

1.6 Almacenamiento en código

Objetivo: Analizar los métodos y formatos utilizados para almacenar información en sistemas digitales mediante codificación.

Definición

- El almacenamiento en código se refiere a la representación de información mediante sistemas de codificación que permiten su preservación y recuperación.
- Implica la transformación de datos en formatos digitales que pueden ser interpretados por los sistemas computacionales.
- Es fundamental para la persistencia de datos y la interoperabilidad entre diferentes sistemas y aplicaciones.

Métodos de Codificación

- **Sistemas numéricos:** Representación de información usando binario, octal, hexadecimal y decimal.
- **Códigos de caracteres:** Sistemas como ASCII, Unicode y UTF-8 para representar texto.
- **Formatos de archivo:** Estructuras específicas para diferentes tipos de datos (texto, imagen, audio, video).
- **Compresión de datos:** Técnicas para reducir el espacio de almacenamiento (lossless y lossy).
- **Codificación para detección y corrección de errores:** Métodos como paridad, Hamming y CRC.

Características

- **Eficiencia:** Optimización del espacio requerido para almacenar información.
- Integridad: Capacidad de detectar y corregir errores en los datos almacenados.
- Interoperabilidad: Compatibilidad entre diferentes sistemas y plataformas.
- Seguridad: Protección de datos mediante técnicas de cifrado y codificación.
- Escalabilidad: Adaptación a volúmenes crecientes de información.

Ejemplos

- El código ASCII que representa caracteres alfanuméricos como números binarios.
- Formatos de imagen como JPEG y PNG que utilizan diferentes algoritmos de compresión.
- Archivos comprimidos ZIP que reducen el tamaño mediante algoritmos lossless.
- Base de datos que almacenan información en tablas con relaciones codificadas.
- Sistemas de cifrado como AES que codifican información para proteger su confidencialidad.

En esencia, el almacenamiento en código es la base para que la información digital sea duradera, manejable y útil en el mundo computacional.

1.7 Números binarios

Objetivo: Comprender el sistema binario, su importancia en computación y las operaciones fundamentales con números binarios.

Definición

- El sistema binario es un sistema de numeración posicional que utiliza solo dos dígitos: 0 y 1.
- Es la base del funcionamiento de los sistemas digitales, ya que los componentes electrónicos pueden representar fácilmente dos estados (encendido/apagado, verdadero/falso).
- Permite representar cualquier tipo de información (números, texto, imágenes, sonido) mediante combinaciones de bits.

Conceptos Fundamentales

- **Bit:** Unidad básica de información que representa un dígito binario (0 o 1).
- **Byte:** Grupo de 8 bits que representa un carácter o unidad de información básica.
- **Conversión entre sistemas:** Métodos para transformar números entre binario, decimal, octal y hexadecimal.
- **Aritmética binaria:** Operaciones matemáticas (suma, resta, multiplicación, división) con números binarios.
- **Representación de números negativos:** Sistemas como signo-magnitud, complemento a uno y complemento a dos.
- **Punto flotante:** Representación de números reales mediante notación científica en binario.

Características

- Simplicidad de implementación en circuitos electrónicos.
- Fiabilidad en la representación y transmisión de datos.
- Eficiencia en operaciones lógicas y aritméticas.
- Escalabilidad para representar información compleja.
- Universalidad como estándar en sistemas digitales.

Ejemplos

- Representación del número decimal 10 como 1010 en binario.
- Suma binaria: 1010 (10) + 0110 (6) = 10000 (16).
- Representación del carácter 'A' en ASCII como el número binario 01000001.
- Uso del complemento a dos para representar números negativos.
- Almacenamiento de imágenes mediante valores binarios que representan píxeles.

1.8 Organización de los componentes fundamentales del computador

---

Objetivo: Identificar y comprender la interconexión y funcionamiento de los componentes principales de un sistema computacional.

Definición

- La organización de componentes se refiere a la disposición estructural y las interconexiones entre las unidades funcionales de un computador.
- Define cómo los diferentes componentes (CPU, memoria, E/S) interactúan para ejecutar instrucciones y procesar datos.

- Establece la arquitectura básica que determina el rendimiento, eficiencia y capacidades del sistema.

Componentes Principales

- **Unidad Central de Procesamiento (CPU):** Cerebro del computador que ejecuta instrucciones y procesa datos.
- **Memoria Principal:** Almacena datos e instrucciones necesarios para la ejecución de programas.
- **Dispositivos de Entrada/Salida:** Permiten la comunicación entre el computador y el exterior.
- **Buses del Sistema:** Canales de comunicación que interconectan los componentes del computador.
- **Unidad de Control:** Coordina las actividades de todos los componentes del sistema.
- **Almacenamiento Secundario:** Provee capacidad de almacenamiento no volátil a largo plazo.

Características

- Arquitectura definida por el conjunto de instrucciones y organización de hardware.
- Jerarquía de memoria que equilibra velocidad, capacidad y costo.
- Mecanismos de interrupción para gestionar eventos asíncronos.
- Técnicas de paralelismo para mejorar el rendimiento.
- Modos de direccionamiento para acceder a datos e instrucciones.

Ejemplos

- Arquitectura Von Neumann con unidad de procesamiento, memoria y buses unificados.
- Arquitectura Harvard con memorias separadas para datos e instrucciones.
- Sistemas multiprocesador con varias CPUs compartiendo recursos.
- Computadoras con buses PCI Express para alta velocidad de transferencia.
- Sistemas con memoria caché multinivel para reducir latencia.

1.9 Instrucciones en lenguaje de máquina

Objetivo: Analizar la estructura, tipos y funcionamiento de las instrucciones en lenguaje de máquina que ejecuta directamente el procesador.

Definición

- Las instrucciones en lenguaje de máquina son comandos en formato binario que la CPU puede ejecutar directamente.
- Representan las operaciones más básicas que puede realizar un computador, específicas para cada arquitectura de procesador.
- Forman el conjunto de instrucciones (instruction set) que define las capacidades fundamentales de un procesador.

Tipos de Instrucciones

- **Transferencia de datos:** Mover información entre memoria, registros y dispositivos.
- **Aritméticas:** Realizar operaciones matemáticas como suma, resta, multiplicación y división.
- **Lógicas:** Operaciones booleanas como AND, OR, NOT y XOR.
- **Control de flujo:** Alterar la secuencia de ejecución (saltos, bifurcaciones, llamadas a subrutinas).
- **Manejo de dispositivos:** Controlar operaciones de entrada/salida con periféricos.

Características

- Codificadas en formato binario específico para cada arquitectura.
- Ejecutadas directamente por el hardware del procesador.
- Organizadas en ciclos de instrucción (fetch, decode, execute).
- Diferentes modos de direccionamiento para acceder a operandos.
- Varían en longitud y complejidad según la arquitectura (CISC vs RISC).

Ejemplos

- Instrucción ADD en x86: 00000011 11000011 (suma los registros AX y BX).
- Instrucción LOAD en arquitectura MIPS: 100011 00001 00010 0000000000000000.
- Salto condicional JZ (Jump if Zero) en ensamblador x86.
- Instrucción MOV para transferencia de datos en múltiples arquitecturas.

- Llamada a interrupción INT en x86 para solicitar servicios del sistema operativo.

2.1. Programación

Objetivo: Comprender los conceptos fundamentales, características y aplicaciones de la programación como disciplina esencial en la informática.

Definición

- La programación es el proceso de crear un conjunto de instrucciones que le dicen a una computadora cómo realizar una tarea específica.
- Consiste en escribir, probar, depurar y mantener el código fuente de programas computacionales.
- Es una disciplina que combina lógica, algoritmos y estructuras de datos para resolver problemas.

Elementos Fundamentales

- **Algoritmos:** Secuencia lógica de pasos para resolver un problema.
- **Lenguajes de programación:** Conjunto de reglas y sintaxis para escribir programas.
- **Estructuras de datos:** Formas de organizar y almacenar información.
- **Paradigmas de programación:** Estilos y metodologías para desarrollar software.
- **Herramientas de desarrollo:** Editores, compiladores, depuradores, etc.

Proceso de Desarrollo de Software

1. Análisis de requisitos: Comprender qué debe hacer el programa.
2. Diseño del algoritmo: Planificar la solución al problema.
3. Codificación: Escribir el código en un lenguaje de programación.
4. Pruebas y depuración: Verificar que el programa funcione correctamente.
5. Documentación: Explicar cómo funciona el programa.
6. Mantenimiento: Actualizar y mejorar el programa.

Paradigmas de Programación Principales

- **Programación estructurada:** Uso de secuencias, selecciones e iteraciones.
- **Programación orientada a objetos (POO):** Basada en objetos y clases.
- **Programación funcional:** Uso de funciones matemáticas puras.
- **Programación imperativa:** Describe cómo lograr un objetivo paso a paso.
- **Programación declarativa:** Describe qué se quiere lograr, no cómo.

Niveles de Lenguajes de Programación

- **Lenguaje máquina:** Código binario ejecutado directamente por el hardware.
- **Lenguaje ensamblador:** Representación simbólica del lenguaje máquina.
- **Lenguajes de alto nivel:** Más cercanos al lenguaje humano (Python, Java, C++).
- **Lenguajes de muy alto nivel:** Especializados en dominios específicos.

Aplicaciones de la Programación

- **Desarrollo web:** Sitios y aplicaciones web.
- **Inteligencia artificial:** Machine learning, redes neuronales.
- **Videojuegos:** Desarrollo de software de entretenimiento.
- **Sistemas embebidos:** Control de dispositivos electrónicos.
- **Ciencia de datos:** Análisis y visualización de datos.
- **Aplicaciones móviles:** Software para smartphones y tablets.

Habilidades del Programador

- Pensamiento lógico y algorítmico.
- Capacidad de resolución de problemas.
- Atención al detalle.
- Trabajo en equipo y colaboración.
- Aprendizaje continuo y adaptación.
- Creatividad e innovación.

2.2. Tipos de proposiciones

Objetivo: Identificar y clasificar los diferentes tipos de proposiciones lógicas según su estructura y características fundamentales.

Definición de Proposición

- Una proposición es una oración declarativa que puede ser calificada como verdadera o falsa, pero no ambas.
- Debe tener un valor de verdad definido (verdadero o falso).
- No puede ser una pregunta, orden o exclamación.

Clasificación Principal

- **Proposiciones simples o atómicas:** No contienen otras proposiciones como partes.
- **Proposiciones compuestas o moleculares:** Formadas por la combinación de proposiciones simples.

Proposiciones Simples

- Características:
    - Expresan una única idea
    - No pueden dividirse en proposiciones más pequeñas
    - Se representan con letras minúsculas (p, q, r, s...)
- Ejemplos:
    - 'El sol es una estrella' (Verdadera)
    - '2 + 3 = 6' (Falsa)
    - 'Lima es la capital de Perú' (Verdadera)

Proposiciones Compuestas

- Conectivos lógicos: Palabras que unen proposiciones simples
- Tipos según el conectivo:
    1. **Conjunción (∧):** 'p y q' - Es verdadera solo cuando ambas son verdaderas
    2. **Disyunción (∨):** 'p o q' - Es falsa solo cuando ambas son falsas
    3. **Condicional (→):** 'Si p entonces q' - Es falsa solo cuando p es verdadera y q falsa
    4. **Bicondicional (↔):** 'p si y solo si q' - Es verdadera cuando ambas tienen el mismo valor
    5. **Negación (¬):** 'No p' - Invierte el valor de verdad

Clasificación por Modalidad

- **Proposiciones afirmativas:** Afirman algo sobre un sujeto
- **Proposiciones negativas:** Niegan algo sobre un sujeto
- **Proposiciones universales:** Aplican a todos los elementos de un conjunto
- **Proposiciones particulares:** Aplican a algunos elementos de un conjunto

Proposiciones según la Lógica Aristotélica

- **Tipo A:** Universal afirmativa - 'Todo S es P'
- **Tipo E:** Universal negativa - 'Ningún S es P'
- **Tipo I:** Particular afirmativa - 'Algún S es P'
- **Tipo O:** Particular negativa - 'Algún S no es P'

Ejemplos de Proposiciones Compuestas

- Conjunción: 'Hace sol y hace calor'
- Disyunción: 'Estudiaré matemáticas o física'

- Condicional: 'Si llueve, entonces cancelaremos el picnic'
- Bicondicional: 'Un triángulo es equilátero si y solo si tiene tres lados iguales'
- Negación: 'No es cierto que la Tierra sea plana'

Reglas para Identificar Proposiciones

1. Verificar que sea una oración declarativa
2. Confirmar que tenga valor de verdad definido
3. Excluir preguntas, órdenes y exclamaciones
4. Identificar si es simple o compuesta
5. Determinar los conectivos lógicos presentes

2.3. Proposiciones de asignación aritmética

Objetivo: Comprender las proposiciones de asignación aritmética en programación, incluyendo su sintaxis, operadores y uso en algoritmos.

Definición

- Una proposición de asignación aritmética es una instrucción que asigna un valor a una variable mediante una expresión aritmética.
- Es una de las operaciones fundamentales en la mayoría de los lenguajes de programación.
- Sigue la estructura: variable = expresión_aritmética

Sintaxis Básica

- **Operador de asignación:** = (igual) en la mayoría de lenguajes
- **Variable destino:** Recibe el resultado de la expresión
- **Expresión aritmética:** Combinación de variables, constantes y operadores
- **Terminador:** ; (punto y coma) en lenguajes como C, Java, JavaScript

Operadores Aritméticos

- **Suma:** + (ejemplo: x = a + b)
- **Resta:** - (ejemplo: y = c - d)
- **Multiplicación:** * (ejemplo: z = e * f)
- **División:** / (ejemplo: w = g / h)
- **Módulo:** % (resto de la división, ejemplo: r = i % j)
- **Potenciación:** ** o ^ (depende del lenguaje)

Tipos de Asignación Aritmética

- **Asignación simple:** Variable = expresión
- **Asignación compuesta:** Combinación con operadores (+=, -=, *=, /=)
- **Asignación múltiple:** Varias variables en una sola instrucción
- **Asignación con incremento/decremento:** ++, -- (prefijo y sufijo)

Reglas de Precedencia de Operadores

1. Paréntesis (siempre tienen la máxima prioridad)
2. Potenciación (si está disponible)
3. Multiplicación, División y Módulo (de izquierda a derecha)
4. Suma y Resta (de izquierda a derecha)
5. Asignación (generalmente la menor prioridad)

Errores Comunes y Consideraciones

- División por cero: Provoca error en tiempo de ejecución
- Desbordamiento: Cuando el resultado excede el rango de la variable
- Precisión en números decimales: Errores de redondeo en operaciones con float/double
- Variables no inicializadas: Usar variables sin valor asignado previamente

Aplicaciones Prácticas

- Cálculos matemáticos en programas científicos
- Procesamiento de datos numéricos
- Contadores y acumuladores en bucles
- Conversiones de unidades
- Cálculos financieros y estadísticos

Buenas Prácticas

1. Usar paréntesis para clarificar la precedencia
2. Inicializar variables antes de usarlas
3. Elegir el tipo de dato apropiado para el cálculo
4. Validar divisiones para evitar división por cero
5. Comentar operaciones complejas para mejor legibilidad

2.4. Proposiciones de asignación lógica

Objetivo: Comprender las proposiciones de asignación lógica en programación, incluyendo operadores booleanos y su aplicación en condiciones y toma de decisiones.

Definición

- Una proposición de asignación lógica es una instrucción que asigna un valor booleano (verdadero o falso) a una variable.
- Se basa en la evaluación de expresiones lógicas que resultan en true o false.
- Sigue la estructura: variable = expresión_lógica

Operadores Lógicos Fundamentales

- **AND (∧):** Verdadero solo si ambos operandos son verdaderos
- **OR (∨):** Falso solo si ambos operandos son falsos
- **NOT (¬):** Invierte el valor de verdad
- **XOR (⊕):** Verdadero si los operandos son diferentes

Tablas de Verdad

- **AND:**
  - true AND true = true
  - true AND false = false
  - false AND true = false
  - false AND false = false
- **OR:**
  - true OR true = true
  - true OR false = true
  - false OR true = true
  - false OR false = false
- **NOT:**
  - NOT true = false
  - NOT false = true

Operadores Relacionales en Asignaciones Lógicas

- **Igualdad:** == o === (depende del lenguaje)
- **Desigualdad:** != o !==
- **Mayor que:** >
- **Menor que:** <
- **Mayor o igual que:** >=

- **Menor o igual que:** <=

Tipos de Asignaciones Lógicas

- **Asignación directa:** variable = true/false
- **Asignación por comparación:** variable = (a > b)
- **Asignación con operadores lógicos:** variable = (x && y) || z
- **Asignación condicional:** variable = (condición) ? valor1 : valor2

Precedencia de Operadores Lógicos

1. Paréntesis (máxima prioridad)
2. Operadores relacionales (>, <, >=, <=, ==, !=)
3. NOT (¬)
4. AND (∧)
5. OR (∨)
6. Asignación (=)

Aplicaciones Prácticas

- Validación de datos de entrada
- Control de flujo en condicionales (if, while)
- Sistemas de reglas y toma de decisiones
- Circuitos lógicos y electrónica digital
- Bases de datos y consultas booleanas
- Inteligencia artificial y sistemas expertos

Consideraciones Importantes

- Evaluación de cortocircuito: Algunos lenguajes evalúan solo lo necesario
- Conversión automática: Cómo se convierten otros tipos a booleanos
- Precisión en comparaciones: Cuidado con comparaciones de punto flotante
- Legibilidad: Usar paréntesis para clarificar expresiones complejas

2.5. Proposición de flujo

Objetivo: Comprender las proposiciones de flujo de control que permiten alterar la secuencia de ejecución de un programa mediante estructuras condicionales e iterativas.

Definición

- Una proposición de flujo es una instrucción que controla el orden de ejecución de las sentencias en un programa.
- Permite alterar la secuencia lineal de ejecución basándose en condiciones o repeticiones.
- Es fundamental para implementar lógica de decisión y bucles en programación.

Tipos de Proposiciones de Flujo

- **Estructuras condicionales:** Permiten ejecutar código basado en condiciones
- **Estructuras iterativas:** Permiten repetir código múltiples veces
- **Estructuras de salto:** Permiten transferir el control a otra parte del código
- **Estructuras de selección múltiple:** Para casos con múltiples alternativas

Instrucciones de Control de Flujo

- **BREAK:** Termina la ejecución de un bucle o switch
- **CONTINUE:** Salta a la siguiente iteración de un bucle
- **RETURN:** Retorna un valor y termina la ejecución de una función
- **GOTO:** Salto incondicional (considerado mala práctica en programación moderna)

Flujo de Ejecución Paso a Paso

1. Evaluación de condición en estructuras condicionales
2. Ejecución selectiva del bloque correspondiente
3. Inicialización y verificación en bucles
4. Repetición controlada con condición de terminación
5. Salida del flujo controlado y continuación del programa

Anidamiento de Estructuras de Flujo

- Los bucles pueden contener condicionales y viceversa
- Los condicionales pueden anidarse para lógica compleja
- Los bucles pueden anidarse para procesar estructuras multidimensionales

- Es importante mantener la legibilidad del código anidado

Buenas Prácticas en el Control de Flujo

1. Evitar bucles infinitos asegurando condiciones de salida
2. Mantener la legibilidad con indentación consistente
3. Usar break y continue con moderación
4. Preferir estructuras claras sobre código complejo anidado
5. Validar condiciones antes de entrar en bucles

Aplicaciones en Algoritmos

- Búsqueda y filtrado de datos
- Validación y procesamiento de entradas
- Implementación de menús y interfaces
- Procesamiento por lotes y iteraciones
- Control de estados y máquinas de estado

2.6. Diagramas

Objetivo: Comprender los diferentes tipos de diagramas utilizados en programación y análisis de sistemas para representar visualmente procesos, estructuras y flujos de información.

Definición

- Un diagrama es una representación gráfica y simbólica que muestra las relaciones entre diferentes componentes de un sistema.
- Sirve como herramienta de comunicación visual para documentar, diseñar y analizar procesos.
- Facilita la comprensión de sistemas complejos mediante abstracción y simplificación.

Tipos de Diagramas en Programación

- **Diagramas de flujo (Flowcharts):** Representan algoritmos y procesos
- **Diagramas UML (Unified Modeling Language):** Para modelado de software orientado a objetos
- **Diagramas de clases:** Muestran la estructura estática del sistema
- **Diagramas de secuencia:** Representan interacciones temporales entre objetos

- **Diagramas de casos de uso:** Describen funcionalidades desde la perspectiva del usuario

Símbolos de Diagramas de Flujo

- **Óvalo:** Inicio/Final del proceso
- **Rectángulo:** Proceso o instrucción
- **Rombo:** Decisión o condición
- **Paralelogramo:** Entrada/Salida de datos
- **Flechas:** Dirección del flujo
- **Círculo:** Conector entre páginas

Diagramas UML Principales

- **Diagrama de clases:** Clases, atributos, métodos y relaciones
- **Diagrama de objetos:** Instancias específicas en un momento dado
- **Diagrama de estados:** Comportamiento de objetos en respuesta a eventos
- **Diagrama de actividades:** Flujo de trabajo entre actividades
- **Diagrama de componentes:** Organización física del sistema
- **Diagrama de despliegue:** Infraestructura hardware del sistema

Reglas para Crear Diagramas Efectivos

1. Definir claramente el propósito del diagrama
2. Mantener la simplicidad y evitar el desorden visual
3. Usar símbolos estándar reconocidos universalmente
4. Asegurar la consistencia en tamaños y estilos
5. Incluir leyendas explicativas cuando sea necesario

Herramientas para Crear Diagramas

- **Software especializado:** Microsoft Visio, Lucidchart, Draw.io
- **Herramientas UML:** Enterprise Architect, StarUML, Visual Paradigm
- **Aplicaciones online:** Diagrams.net, Creately, Gliffy
- **Software libre:** Dia, yEd Graph Editor
- **Integradas en IDEs:** PlantUML, herramientas UML en Eclipse/IntelliJ

Ejemplos de Aplicación

- **Diagrama de flujo para login:**
    - Inicio → Ingresar usuario/contraseña → Validar → ¿Válido? → Sí: Acceder / No: Mostrar error
- **Diagrama de clases para sistema bancario:**

    - Clases: Cuenta, Cliente, Transacción, Banco
    - Relaciones: Herencia, Asociación, Composición
- **Diagrama de secuencia para compra online:**
    - Cliente → Sistema → Base de datos → Pasarela de pago → Confirmación

Ventajas del Uso de Diagramas

- Comunicación clara entre miembros del equipo
- Detección temprana de errores en el diseño
- Documentación visual fácil de entender
- Análisis de procesos y optimización
- Facilita el mantenimiento del sistema

Mejores Prácticas en Diagramación

- **Flujo consistente:** Mantener dirección uniforme (generalmente de arriba hacia abajo)
- **Etiquetado claro:** Usar nombres descriptivos para elementos
- **Agrupación lógica:** Organizar elementos relacionados cercanos
- **Evitar cruces:** Minimizar el entrecruzamiento de líneas
- **Versiones:** Mantener histórico de cambios importantes

Diagramas Específicos por Área

- **Base de datos:** Diagramas Entidad-Relación (ER)
- **Redes:** Diagramas de topología de red
- **Procesos empresariales:** BPMN (Business Process Model and Notation)
- **Arquitectura:** Diagramas de arquitectura de software
- **Datos:** Diagramas de flujo de datos (DFD)

3.1. Proposición de entrada

Objetivo: Comprender las proposiciones de entrada que permiten capturar datos del usuario o de fuentes externas para su procesamiento en programas.

Definición

- Una proposición de entrada es una instrucción que permite leer datos desde dispositivos de entrada hacia la memoria del programa.
- Facilita la interacción usuario-programa mediante la captura de información externa.
- Es esencial para crear programas dinámicos y interactivos que respondan a entradas variables.

Dispositivos de Entrada Comunes

- **Teclado:** Entrada de texto y comandos
- **Mouse:** Entrada de coordenadas y clics
- **Archivos:** Lectura de datos desde almacenamiento
- **Red:** Recepción de datos desde servidores
- **Sensores:** Captura de datos del entorno físico
- **Micrófono:** Entrada de audio

Tipos de Datos de Entrada

- **Texto:** Cadenas de caracteres (strings)
- **Numéricos:** Enteros, decimales, flotantes
- **Booleanos:** Valores verdadero/falso
- **Fechas y horas:** Información temporal
- **Archivos binarios:** Imágenes, documentos, etc.
- **Estructurados:** JSON, XML, CSV

Proceso de Captura de Entrada

1. Solicitud de entrada al usuario o sistema
2. Espera y recepción de los datos
3. Validación y verificación de la entrada
4. Conversión de tipo si es necesario
5. Almacenamiento en variables para su uso

Validación de Entrada

- **Verificación de tipo:** Confirmar que los datos tienen el formato esperado

- **Rangos permitidos:** Validar que los valores estén dentro de límites aceptables
- **Formato específico:** Verificar patrones como emails, teléfonos, etc.
- **Entrada obligatoria:** Asegurar que se proporcionen datos requeridos
- **Sanitización:** Eliminar caracteres peligrosos o maliciosos

Manejo de Errores en Entrada

- **Excepciones por tipo incorrecto:** Capturar errores de conversión
- **Valores fuera de rango:** Mensajes de error específicos
- **Entrada vacía o nula:** Manejo de casos especiales
- **Archivos no encontrados:** Verificación de existencia
- **Timeouts:** Límites de tiempo para respuesta

Buenas Prácticas en Entrada de Datos

1. Siempre validar la entrada antes de procesarla
2. Proporcionar mensajes claros al solicitar datos
3. Usar tipos de datos apropiados para cada entrada
4. Manejar excepciones adecuadamente
5. Limitar y sanitizar entradas por seguridad

Aplicaciones de Entrada de Datos

- Formularios web y aplicaciones de escritorio
- Sistemas de autenticación y login
- Procesamiento de archivos y bases de datos
- Interfaces de usuario gráficas (GUI)
- Sistemas de control y monitoreo
- Aplicaciones móviles con entrada táctil

Consideraciones de Seguridad

- Prevención de inyección SQL en bases de datos
- Validación contra XSS (Cross-Site Scripting)
- Sanitización de entradas HTML
- Límites de tamaño para prevenir desbordamientos
- Autenticación y autorización de fuentes de entrada

3.2. Proposición de salida

Objetivo: Comprender las proposiciones de salida que permiten mostrar resultados, información y datos desde el programa hacia el usuario o dispositivos externos.

Definición

- Una proposición de salida es una instrucción que envía datos desde el programa hacia dispositivos de salida o almacenamiento.
- Permite comunicar resultados e información generada por el programa al usuario o a otros sistemas.
- Es fundamental para visualizar y persistir los resultados del procesamiento.

Dispositivos de Salida Comunes

- **Pantalla/Monitor:** Visualización de texto e interfaces gráficas
- **Impresora:** Salida física en papel
- **Archivos:** Almacenamiento permanente de datos
- **Red:** Envío de datos a servidores o otros dispositivos
- **Altavoces:** Salida de audio y sonidos
- **Puertos serie/paralelo:** Comunicación con dispositivos externos

3.3. Proposición de control

Objetivo: Comprender y aplicar las proposiciones de control que permiten dirigir el flujo de ejecución de un programa mediante estructuras de decisión y bucles.

Definición

- Una proposición de control es una instrucción que determina el orden de ejecución de otras instrucciones en un programa.
- Permite alterar el flujo secuencial basándose en condiciones o repeticiones.
- Es esencial para implementar lógica de programación compleja y algoritmos no lineales.

Tipos de Proposiciones de Control

- **Estructuras condicionales:** if, if-else, if-else if, switch-case
- **Estructuras iterativas:** for, while, do-while, for-each
- **Estructuras de salto:** break, continue, return, goto
- **Manejo de excepciones:** try-catch, throw, finally

Instrucciones de Control de Flujo

- **BREAK:** Termina la ejecución de un bucle o switch
- **CONTINUE:** Salta a la siguiente iteración de un bucle
- **RETURN:** Retorna valor y termina ejecución de función
- **GOTO:** Salto incondicional (evitar en programación moderna)
- **THROW:** Lanza una excepción para manejo de errores

Anidamiento de Estructuras de Control

1. Bucles dentro de condicionales y viceversa
2. Múltiples niveles de anidamiento para lógica compleja
3. Estructuras try-catch alrededor de código propenso a errores
4. Switch dentro de bucles para procesamiento por casos
5. Condicionales dentro de funciones para control de retorno

Buenas Prácticas en Control de Flujo

1. Evitar bucles infinitos con condiciones de salida claras
2. Mantener la legibilidad con indentación consistente
3. Limitar el anidamiento excesivo para mayor claridad
4. Usar break y continue con moderación y claridad
5. Manejar todas las excepciones posibles

Patrones Comunes de Control

- **Validación de entrada:** if anidados para verificar datos
- **Búsqueda en colecciones:** for/while con break al encontrar
- **Menús interactivos:** do-while con switch-case
- **Procesamiento por lotes:** for para iterar sobre arrays
- **Control de estados:** switch para máquinas de estado

Consideraciones de Rendimiento

- Complejidad algorítmica de estructuras anidadas
- Optimización de condiciones más probables primero
- Uso eficiente de bucles para grandes volúmenes
- Evitar condiciones redundantes o innecesarias
- Cacheo de resultados en condiciones costosas

3.4. Proposición de formato

Objetivo: Comprender y aplicar las proposiciones de formato que permiten controlar la presentación y estructuración de datos en la salida de programas.

Definición

- Una proposición de formato es una instrucción que especifica cómo deben presentarse los datos en la salida.
- Permite controlar la apariencia y estructura de la información mostrada al usuario.
- Es esencial para crear salidas legibles y profesionales en aplicaciones.

Elementos de Formato Comunes

- **Alineación:** Izquierda, derecha, centrado, justificado
- **Longitud de campo:** Ancho fijo para valores
- **Precisión decimal:** Número de dígitos después del punto
- **Relleno:** Caracteres para completar espacios (espacios, ceros)
- **Separadores:** Comas para miles, puntos para decimales
- **Notación:** Decimal, científica, hexadecimal, octal

Especificadores de Formato Principales

- **Enteros:**
- **Flotantes:**
- **Cadenas:**
- **Booleanos:**
- **Fechas:**

Proceso de Aplicación de Formato

1. Identificar el tipo de dato a formatear
2. Seleccionar el especificador apropiado
3. Definir opciones de formato (ancho, precisión, etc.)
4. Aplicar el formato a los datos
5. Validar el resultado formateado

Formatos Específicos por Tipo de Dato

- **Monetarios:** Símbolo de moneda, separadores, decimales
- **Fechas y horas:** Día/mes/año, formato 12/24 horas
- **Porcentajes:** Multiplicación por 100, símbolo %

- **Números telefónicos:** Formatos internacionales
- **Direcciones IP:** Cuatro octetos separados por puntos

Buenas Prácticas en Formato

1. Mantener consistencia en todo el programa
2. Considerar la localización del usuario
3. Validar formatos personalizados exhaustivamente
4. Documentar formatos complejos para mantenimiento
5. Usar constantes para formatos reutilizables

Formatos Internacionales (Localización)

- **Fecha:** EE.UU. (mm/dd/aaaa) vs Europa (dd/mm/aaaa)
- **Decimales:** Punto (1.5) vs coma (1,5)
- **Separadores de miles:** Coma (1,000) vs punto (1.000)
- **Moneda:** Posición del símbolo, separadores
- **Horario:** 12h (AM/PM) vs 24h (00:00-23:59)

Herramientas de Formato Avanzado

- **Expresiones regulares:** Para validación y transformación
- **Librerías de internacionalización:** Para soporte multiidioma
- **Plantillas y templates:** Para documentos estructurados
- **Formatos estándar:** JSON, XML, CSV para intercambio
- **CSS/HTML:** Para formato en aplicaciones web

Consideraciones de Rendimiento

- Costo computacional de operaciones de formato
- Reutilización de objetos de formato cuando sea posible
- Cacheo de resultados para formatos costosos
- Optimización de concatenaciones de cadenas
- Uso de StringBuilder para múltiples operaciones

3.5. Formato I

Objetivo: Comprender los conceptos fundamentales y técnicas básicas de formato para la presentación de datos en programas, incluyendo alineación, padding y especificadores simples.

Definición

- Formato I se refiere a las técnicas básicas de formateo que controlan la presentación visual de datos en salida.
- Enfocado en operaciones fundamentales de formato como alineación y relleno.
- Proporciona la base esencial para formatos más avanzados.

Conceptos Fundamentales

- **Campo de salida:** Espacio reservado para mostrar un valor
- **Ancho de campo:** Número total de caracteres ocupados
- **Alineación:** Posición del texto dentro del campo
- **Carácter de relleno:** Carácter usado para espacios vacíos
- **Precisión:** Número de dígitos decimales mostrados

Alineación Básica

- **Alineación a la izquierda:** Texto comienza en el borde izquierdo
- **Alineación a la derecha:** Texto termina en el borde derecho
- **Centrado:** Texto equidistante de ambos bordes
- **Justificado:** Texto expandido para llenar todo el ancho

Proceso de Formato Básico

1. Identificar el tipo de dato a formatear
2. Seleccionar el especificador apropiado
3. Determinar el ancho de campo necesario
4. Aplicar alineación y relleno
5. Generar la salida formateada

uenas Prácticas en Formato Básico

1. Mantener consistencia en ancho de columnas
2. Usar alineación apropiada para cada tipo de dato
3. Evitar truncamiento accidental de información
4. Considerar la legibilidad del resultado final

5. Documentar formatos complejos

Errores Comunes en Formato Básico

- Ancho insuficiente para datos largos
- Precisión excesiva que genera ruido visual
- Mezcla de alineaciones en la misma columna
- Uso incorrecto de especificadores de tipo
- Falta de separación entre columnas

Aplicaciones Prácticas

- Reportes tabulares simples
- Listados de datos alineados
- Formatos de ticket y recibos
- Interfaces de línea de comandos
- Logs y mensajes del sistema

3.6. Formato F

Objetivo: Dominar el formato F para números de punto flotante, incluyendo precisión decimal, notación científica y representación de valores reales en programación.

Definición

- Formato F se refiere específicamente al formateo de números de punto flotante en programación.
- Utiliza el especificador %f para representar valores reales con parte decimal.
- Permite controlar precisión, redondeo y presentación de números decimales.

Características del Formato F

- **Redondeo automático:** Ajusta al número de decimales especificado
- **Relleno con ceros:** Completa decimales faltantes con ceros
- **Separador decimal:** Usa punto (.) como estándar internacional
- **Manejo de signo:** Controla visualización de signo positivo/negativo
- **Alineación:** Por defecto alineado a la derecha

Proceso de Formateo F

1. Identificar el valor flotante a formatear
2. Determinar la precisión requerida (número de decimales)

3. Seleccionar la notación (fija, científica, automática)
4. Aplicar redondeo y ajustes según especificaciones
5. Generar la representación final

Aplicaciones Específicas

- **Cálculos financieros:** Siempre 2 decimales para moneda
- **Mediciones científicas:** Precisión variable según instrumento
- **Gráficos y visualizaciones:** Etiquetas de ejes formateadas
- **Reportes técnicos:** Consistencia en número de decimales
- **Interfaces de usuario:** Formato amigable para usuarios

Consideraciones de Precisión

- Errores de redondeo: Acumulación en operaciones múltiples
- Límites de representación: Precisión limitada de float/double
- Comparaciones seguras: Usar tolerancias en lugar de igualdad exacta
- Propagación de error: En cálculos en cadena
- Precisión vs. rendimiento: Trade-off en aplicaciones críticas

Buenas Prácticas en Formato F

1. Usar precisión consistente en todo el sistema
2. Evitar decimales excesivos que no aportan información
3. Considerar el contexto de uso para elegir notación
4. Validar rangos esperados antes de formatear
5. Documentar decisiones de precisión

3.7. Formato E

Objetivo: Dominar el formato E para notación científica, comprendiendo su sintaxis, aplicaciones y uso en la representación de números muy grandes o muy pequeños.

Definición

- Formato E se refiere a la notación científica o exponencial para representar números.
- Utiliza el especificador %e para valores en formato mantisa-exponente.
- Es ideal para números extremadamente grandes o pequeños donde la notación decimal es impráctica.

Estructura de la Notación Científica

- **Mantisa:** Parte decimal normalizada (1.0 ≤ |mantisa| < 10.0)
- **Exponente:** Potencia de 10 que escala la mantisa

Ventajas del Formato E

- **Compactación:** Representa números grandes en poco espacio
- **Claridad:** Facilita comparación de órdenes de magnitud
- **Prevención de errores:** Evita confusiones con ceros
- **Estandarización:** Formato universalmente reconocido
- **Precisión relativa:** Mantiene cifras significativas

Proceso de Conversión a Notación Científica

1. Identificar el número a convertir
2. Mover el punto decimal hasta tener un dígito antes del punto
3. Contar las posiciones movidas para determinar el exponente
4. Aplicar redondeo según la precisión requerida
5. Formatear la representación final

Campos de Aplicación

- **Física:** Constantes fundamentales, medidas atómicas
- **Química:** Número de Avogadro, constantes de equilibrio
- **Ingeniería:** Tolerancias muy pequeñas, medidas nanométricas
- **Astronomía:** Distancias interestelares, masas planetarias
- **Computación científica:** Algoritmos numéricos, análisis de error

Comparación con Otros Formatos

- **vs. Formato F:** E para rango amplio, F para valores cotidianos
- **vs. Formato G:** E siempre científico, G elige automáticamente
- **vs. Notación de ingeniería:** E usa múltiplos de 3 en exponente
- **vs. Formato decimal:** E más compacto para extremos

Consideraciones de Uso

- Audiencia objetivo: Usuarios técnicos vs. generales
- Contexto de presentación: Gráficos, tablas, texto
- Consistencia: Mantener mismo formato en conjuntos de datos
- Precisión adecuada: Ni muy poca ni excesiva
- Límites del sistema: Rangos de exponente soportados

Buenas Prácticas en Formato E

1. Usar para valores fuera del rango 0.001 a 1000
2. Mantener cifras significativas consistentes
3. Incluir siempre el signo del exponente
4. Validar que la mantisa esté normalizada
5. Documentar el significado de valores específicos

3.8. Capacidades adicionales de formato

Objetivo: Explorar funcionalidades avanzadas de formato que permiten personalización extensiva, localización internacional y técnicas especializadas para presentación de datos.

Definición

- Capacidades adicionales de formato incluyen funcionalidades avanzadas más allá de los especificadores básicos.
- Permiten personalización extensiva y adaptación a necesidades específicas de presentación.
- Incluyen técnicas para internacionalización, alineación compleja y formatos especializados.

Formatos Personalizados Avanzados

- **Formatos condicionales:** Cambian según el valor del dato
- **Formatos compuestos:** Combinan múltiples tipos en una salida
- **Formatos anidados:** Incluyen formatos dentro de formatos
- **Formatos dinámicos:** Se generan en tiempo de ejecución
- **Plantillas reutilizables:** Patrones de formato almacenados

Internacionalización y Localización (i18n/l10n)

- **Configuración regional (Locale):** Formatos específicos por país/idioma
- **Moneda:** Símbolos, posición y separadores locales
- **Fechas y horas:** Formatos culturalmente apropiados
- **Números:** Separadores decimales y de miles locales
- **Texto bidireccional:** Soporte para idiomas RTL (derecha a izquierda)

Formatos Especializados por Tipo de Dato

- **Direcciones IP:** Formato IPv4 (192.168.1.1) e IPv6

- **Números telefónicos:** Formatos internacionales E.164
- **ISBN/ISSN:** Formatos estandarizados para publicaciones
- **Códigos de barras:** Representación numérica especializada
- **Coordenadas geográficas:** Grados, minutos, segundos o decimales

Técnicas de Alineación Avanzada

- **Alineación decimal:** Puntos decimales alineados verticalmente
- **Alineación mixta:** Diferentes alineaciones en la misma línea
- **Justificación completa:** Texto expandido para llenar ancho
- **Alineación visual:** Basada en contenido semántico
- **Tablas complejas:** Múltiples niveles de anidamiento

Proceso de Formato Avanzado

1. Analizar requisitos de presentación específicos
2. Seleccionar técnicas de formato apropiadas
3. Configurar parámetros de localización
4. Implementar lógica de formato condicional
5. Validar y probar el resultado final

3.9. Código Hollerith

Objetivo: Comprender el código Hollerith como sistema histórico de representación de datos en tarjetas perforadas y su influencia en el desarrollo de la informática moderna.

Definición

- El código Hollerith es un sistema de codificación de caracteres utilizado en tarjetas perforadas para el procesamiento de datos.
- Fue desarrollado por Herman Hollerith para el censo de Estados Unidos de 1890.
- Representa el primer sistema mecanizado de procesamiento de datos que condujo a la fundación de IBM.

Contexto Histórico

- **1890:** Primera aplicación exitosa en el censo estadounidense
- **1896:** Hollerith funda la Tabulating Machine Company
- **1924:** La empresa se convierte en International Business Machines (IBM)
- **1950s-1970s:** Uso extensivo en computación mainframe

- **1980s:** Declive gradual con la llegada de medios magnéticos

Estructura de la Tarjeta Perforada

- **Dimensiones estándar:** 7⅜ × 3¼ pulgadas (187 × 82 mm)
- **Columnas:** 80 columnas para caracteres
- **Filas:** 12 filas de posición de perforación (0-9 más 2 zonas)
- **Material:** Cartulina de alta calidad y durabilidad
- **Capacidad:** 80 caracteres por tarjeta

Sistema de Codificación

- **Zonas:** Filas 12 (Zona superior) y 11 (Zona 9-8)
- **Dígitos:** Filas 0-9 para números y combinaciones
- **Letras:** Combinación de zona + dígito (ej: Zona 12 + Fila 1 = A)
- **Símbolos:** Combinaciones especiales de perforaciones
- **Carácter nulo:** Ausencia de perforaciones

Proceso de Codificación y Decodificación

1. Identificar el carácter a representar
2. Determinar la combinación de zona y dígito correspondiente
3. Perforar las posiciones específicas en la tarjeta
4. Verificar la perforación mediante inspección visual o mecánica
5. Leer mediante sensores mecánicos o ópticos

Ejemplos de Codificación Hollerith

- **Número 5:** Perforación única en fila 5
- **Letra A:** Perforaciones en zona 12 y fila 1
- **Letra J:** Perforaciones en zona 11 y fila 1
- **Símbolo +:** Perforaciones en zona 12 y filas 6, 8
- **Espacio:** Sin perforaciones en la columna

Equipos y Máquinas Hollerith

- **Perforadora:** Máquina para crear las tarjetas
- **Clasificadora:** Ordenaba tarjetas según criterios
- **Tabuladora:** Contaba y sumaba datos de las tarjetas
- **Reproductora:** Copiaba tarjetas perforadas
- **Intérprete:** Imprimía caracteres en el borde de la tarjeta

Ventajas del Sistema Hollerith

- **Alta velocidad:** Procesamiento más rápido que manual
- **Precisión:** Reducción de errores humanos
- **Reutilizabilidad:** Tarjetas podían reprocesarse
- **Almacenamiento:** Archivo físico permanente
- **Estandarización:** Formato universalmente aceptado

Limitaciones y Desventajas

- **Capacidad limitada:** Solo 80 caracteres por tarjeta
- **Fragilidad:** Daño por manipulación o condiciones ambientales
- **Velocidad:** Lenta comparada con medios electrónicos
- **Costo:** Equipos especializados y mantenimiento
- **Error humano:** Perforaciones incorrectas o dañadas

Legado e Influencia

- **IBM:** Base del imperio empresarial de IBM
- **Formatos modernos:** Influencia en codificaciones como EBCDIC
- **Términos sobrevivientes:** 'Bug' por insectos atrapados en máquinas
- **Arquitectura:** Conceptos que influyeron en diseño de computadoras
- **Museos:** Preservación como patrimonio tecnológico

Aplicaciones Históricas

- **Censos poblacionales:** Estados Unidos y otros países
- **Nóminas y contabilidad:** Empresas y gobiernos
- **Investigación científica:** Procesamiento de datos estadísticos
- **Gestión de inventarios:** Control de almacenes y producción
- **Reservaciones:** Aerolíneas y sistemas de transporte

Buenas Prácticas en el Uso Histórico

1. Verificar perforaciones antes del procesamiento
2. Almacenar tarjetas en condiciones controladas
3. Utilizar secuencias de control para validación
4. Mantener equipos con limpieza y calibración regular
5. Documentar formatos de codificación específicos

Transición a Tecnologías Modernas

- **Cintas magnéticas:** Mayor capacidad y velocidad
- **Discos duros:** Acceso aleatorio y mayor densidad
- **Medios ópticos:** CD, DVD para almacenamiento masivo

- **Almacenamiento sólido:** Memorias flash y SSD
- **Cloud computing:** Almacenamiento distribuido y remoto

4.1 Introducción

Objetivo: Comprender la importancia de la programación modular y cómo las funciones y subrutinas mejoran la organización del código.

¿Por qué necesitamos funciones?

- Imagina que estás construyendo una casa: en lugar de hacer todo de una vez, divides el trabajo en partes (cimientos, paredes, techo).
- En programación pasa igual: las funciones nos permiten dividir un programa grande en partes más pequeñas y manejables.
- Esto hace que el código sea más fácil de entender, mantener y reparar.

Ventajas de usar funciones

- **Reutilización:** Escribes una vez, usas muchas veces.
- **Organización:** Código más ordenado y fácil de leer.
- **Depuración más fácil:** Si hay un error, sabes dónde buscar.
- **Trabajo en equipo:** Cada programador puede trabajar en una función diferente.

4.2 Definiciones

Objetivo: Conocer los conceptos básicos y la terminología relacionada con funciones y subprogramas.

¿Qué es un subprograma?

- Es un **bloque de código** que realiza una tarea específica.
- Es como una **mini-receta** dentro de una receta más grande.
- Se puede **llamar** (usar) desde diferentes partes del programa principal.

Términos importantes

- **Función:** Subprograma que devuelve un valor.
- **Subrutina:** Subprograma que NO devuelve un valor (solo realiza acciones).
- **Parámetros:** Datos que le pasamos al subprograma para que trabaje con ellos.

- **Argumentos:** Los valores concretos que pasamos cuando llamamos al subprograma.
- **Retorno:** Valor que devuelve una función después de ejecutarse.

4.3 Tipos de subprogramas

Objetivo: Identificar los diferentes tipos de subprogramas y cuándo usar cada uno.

Clasificación por lo que devuelven

- **Funciones (devuelven valor):** Como una calculadora que siempre te da un resultado.
- **Subrutinas (no devuelven valor):** Como una alarma que suena pero no te da información.

Clasificación por cómo se usan

- **Integradas (built-in):** Vienen con el lenguaje de programación.
- **Definidas por el usuario:** Las crea el programador según sus necesidades.

Clasificación por su visibilidad

- **Públicas:** Se pueden usar desde cualquier parte del programa.
- **Privadas:** Solo se pueden usar dentro de cierto contexto.

4.4 Formación de un subprograma

Objetivo: Aprender las partes que componen un subprograma y cómo crearlo correctamente.

Partes de un subprograma

- **Encabezado/Cabecera:** El 'nombre' y 'firma' del subprograma.
- **Parámetros:** Las 'variables' que recibe para trabajar.
- **Cuerpo:** Las instrucciones que realiza.
- **Valor de retorno (opcional):** Lo que devuelve al terminar.

Pasos para crear un subprograma

- **Paso 1:** Decidir qué va a hacer el subprograma.
- **Paso 2:** Elegir un nombre descriptivo.
- **Paso 3:** Definir qué parámetros necesita.

- **Paso 4:** Escribir las instrucciones en el cuerpo.
- **Paso 5:** Decidir si devuelve algún valor.

Buenas prácticas

- Usar nombres descriptivos.
- Un subprograma debe hacer solo UNA cosa bien.
- Documentar qué hace el subprograma.
- Manejar posibles errores en los parámetros.

4.5 Subprogramas de función

Objetivo: Dominar el uso de funciones que devuelven valores y entender su utilidad.

Características de las funciones

- **Siempre devuelven un valor** (de lo contrario, serían subrutinas).
- Se pueden usar en expresiones matemáticas.
- Pueden llamarse a sí mismas (recursión).
- Pueden devolver diferentes tipos de datos.

¿Cuándo usar una función?

- Cuando necesitas **calcular algo** y obtener un resultado.
- Cuando una operación se **repite muchas veces** en tu programa.
- Cuando quieres **simplificar cálculos complejos**.

Ventajas específicas de las funciones

- Permiten crear **expresiones más legibles**.
- Facilitan las **pruebas unitarias**.
- Mejoran el **rendimiento** al evitar código repetido.

4.6 Subprogramas de subrutinas

Objetivo: Comprender el uso de subrutinas y cuándo son más apropiadas que las funciones.

¿Qué son las subrutinas?

- Son subprogramas que **realizan acciones** pero **no devuelven valores**.

- Se enfocan en **hacer** algo en lugar de **calcular** algo.
- También se llaman **procedimientos** o **métodos void**.

¿Cuándo usar una subrutina?

- Cuando necesitas **mostrar información** en pantalla.
- Para **modificar variables globales** o estructuras de datos.
- Cuando realizas una **serie de pasos** que no producen un resultado calculable.
- Para **configurar** o **inicializar** partes del programa.

Diferencias clave con las funciones

- **Funciones:** Se usan en expresiones (derecha del =).
- **Subrutinas:** Se llaman como instrucciones independientes.
- **Funciones:** Devuelven un valor que puedes guardar.
- **Subrutinas:** Realizan acciones directamente.

4.7 Subprogramas combinados

Objetivo: Aprender a combinar diferentes tipos de subprogramas para crear soluciones más complejas.

¿Qué son los subprogramas combinados?

- Son **sistemas** donde múltiples subprogramas trabajan juntos.
- Las funciones y subrutinas se **llaman entre sí** para resolver problemas complejos.
- Crean una **estructura modular** que facilita el desarrollo de programas grandes.

Ventajas de la combinación

- **Divide y vencerás:** Problemas grandes se dividen en pequeños.
- **Reutilización máxima:** Un subprograma puede usarse en múltiples proyectos.
- **Mantenimiento más fácil:** Cambias una parte sin afectar las demás.
- **Trabajo en equipo:** Cada programador trabaja en subprogramas diferentes.

Estrategias para combinar subprogramas

- **Top-down:** Empiezas con el problema general y lo divides.

- **Bottom-up:** Creas subprogramas pequeños y los unes.
- **Modular:** Agrupas subprogramas relacionados en módulos.