# Algoritmos y estructura de datos

Asignatura anual, código 082021

**MODULO 1 Conceptos básicos**

Departamento de Ingeniería en Sistemas de Información
**Universidad Tecnológica Nacional FRBA**

# Tabla de contenido

# Conceptos básicos

# 1

**Objetivos de aprendizaje**
Dominando los temas del presente capitulo Usted podrá.

1. Conocer la terminología propia de la disciplina.
2. Definir y comprender claramente conceptos específicos muchas veces mal definidos
3. Comprender el valor de la abstracción.
4. Dar valor a la eficiencia en las soluciones
5. Introducirse en la notación algorítmica y a la forma e encarar los problemas de programación

## Introducción:

Se presenta el alcance del presente trabajo y se introducen conceptos fundamentales de algoritmia y programación, los que servirán de base para el desarrollo de los temas a tratar.

## Contextualizacion de la materia

- Nombre Algoritmos y estructura de datos
- Codigo 082021 – Area Programacion.
- Plantel docente
    - Director de Catedra
        - Dr. Oscar BRUNO
    - Profesores concursados
        - Esp. CC. Marta Ferrari, Esp. Lic. Alejandro Frankel, Ing. Pablo Sznajdleder.
    - Profesores interinos
        - Lic. Hugo Cuello, Esp. Ing. Jose Maria Sola, Lic. Javier Bianchi, Ing. Yamila Zakhem, Ing. Gabriela Sanroman, Ing. Adrian Fiore, Mg. Elena Garcia, Ing. Pablo Mendez.
    - Auxiliares
        - Mg. Marcelo Lipkin, Ing. Natalia Perez Lopez, Ing. Diego Juan, Sr. Flavio Cangini.

Doctor en Educación – Posgrado PosDoctoral
Magister en Doc. Universitaria – Esp. Ing. En Sistemas
Lic. En Sistemas – Prof. Diciplinas Industriales
oscarrbruno@yahoo.com twitter @orbruno

Contextualización Mediante la **M**atriz**T**ransversal**C**ontenidos

El objetivo formar profesionales con dominio profundo del paradigma imperativo

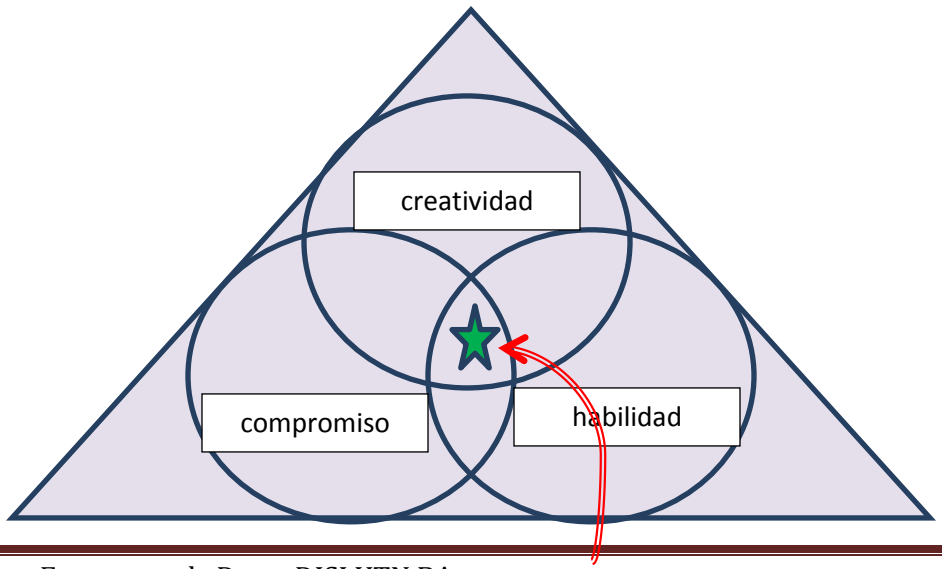
1. Habilidades: los programadores imperativos deben disponer de las siguientes habilidades:
    a. Lógica para informáticos: **(LI)**.
    b. Resolución de problemas: **(RP)**.
    c. Conocimiento de terminología propia de la disciplina **(TD)**.

d. Tipos de datos::
    i. Datos simples **(DS).**
    ii. Estructura registro **(ER).**
    iii. Archivos: Archivos de texto **(AT)** y Archivos Binarios **(AB).**
    iv. Estructuras indexadas **(EI)**.
    v. Estructuras enlazadas lineales **(EL)**.
    vi. Estructuras arbóreas. **(EA)**.
    vii. Grafos **(GR)**
    viii. Combinaciones complejas de estructuras de datos **(CC).**
e. Patrones algorítmicos:
    i. Patrones de inicialización, inserción, búsqueda y recorrido
    ii. Patrones de ordenamiento, agrupaciones, carga y modificaciones
f. Implementación en lenguajes de programación.
    i. Iniciales con utilización de módulos **(IM)**
    ii. Con utilización de bibliotecas (TAD) (**IT)**
    iii. Orientada a objetos **(IO)**
    iv. Otros paradigmas (**OP**)

**Matriz Transversal de Contenidos propuesta:**

| | Generales | | | Tipos de datos | | | | | | | | | | Imp. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LI | RP | TD | DS | ER | AT | AB | EI | EL | EA | EG | CC | IT | T | P |
| MD | ALTA | BAJA | MED. | BAJA | BAJA | BAJA | BAJA | BAJA | BAJA | BAJA | BAJA | NO | NO | NO | NO |
| **AyE** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **ALTA** | **MED.** | **BAJA** | **ALTA** | **MED.** | **MED.** | **IMIT** |
| SSL | ALTA | MED. | ALTA | BAJA | MED. | MED. | MED. | MED. | BAJA | MED. | ALTA | MED. | ALTA | ALTA | IT |
| PP | ALTA | MED. | ALTA | BAJA | BAJA | MED. | MED. | MED. | BAJA | MED. | BAJA | BAJA | ALTA | ALTA | IOOP |

**Aprender programación**



**Metodos**

1. De asimilación y desarrollo
    a. Inductivo → en las clases presenciales, con ejercicios
        i. Básico
        ii. Construcción de conceptos
        iii. De investigación
    b. De instrucción →complementar con material digital (blog, videos)
        i. Transmision
        ii. Significativo
        iii. Debate
    c. De cambio conceptual →con problemas
        i. Discusion
        ii. Propuesta de cambios
2. Para la acción practica → trabajos practicos
    a. Estudio de casos
    b. Solución de problemas
    c. Construcción de problematizaciones
    d. Proyectos
    e. Tutorias
3. Para el desarrollo de habilidades operativas → integración Ejercicio-Problema-TP.
    a. Demostraciones y ejercitaciones
    b. Simulaciones
4. Desarrollo personal → esto es personal
    a. Basado en fortalezas
    b. Fijando metas
    c. Motivación y cambio

## Informática

Disciplina del estudio sistematizado de los procesos algorítmicos que describen y transforman información, su teoría, análisis, diseño, eficiencia, implementación y aplicación.
La informática es una disciplina científica, matemática y una ingeniería; tiene tres formas de pensar propias: Teoría, abstracción y diseño.
Las tres se complementan para la resolución de la mayoría de los problemas.

- ✓ Teoría: Con el pensamiento teórico se describen y prueban relaciones.
- ✓ Abstracción: Recolección de datos y formulación de un modelo, se eliminan los detalles irrelevantes.
- ✓ Diseño: se tienen en cuenta requisitos, especificaciones y se diseñan o analizan mecanismos para resolver problemas. Supone llevar a la práctica los resultados teóricos.

## Programación

La programación es una actividad transversal asociada a cualquier área de la informática, aunque es la ingeniería del software el área específica que se ocupa de la creación del software.

En principio la programación se veía como un arte, solo era cuestión de dominar un lenguaje de programación y aplicar habilidades personales a la resolución de problemas, casi en forma artesanal. El software era visto como algo desarrollado a través de la intuición sin la utilización de métodos de diseño con técnicas para proceder en forma sistemática y sin ningún control de su desarrollo. Con el reconocimiento de la complejidad del desarrollo del software nació la ingeniería del software.
Se considero que al igual que cualquier otra disciplina la creación de software debía ser reconocida como una actividad de ingeniería que requería la aplicación de sólidos principios científicos.
La ingeniería del software es la disciplina que se ocupa de la aplicación del conocimiento científico al diseño y construcción de programas de computación y a todas las actividades asociadas de documentación, operación y mantenimiento, lo que proporciona un enfoque sistemático.
La programación es una actividad en la que la creatividad juega un rol primordial
**Programa**:
Conjunto de instrucciones, ejecutables sobre una computadora, que permite cumplir una función especifica. Se asocia al programa con una determinada función o requerimiento a satisfacer por la ejecución del conjunto de instrucciones que lo forman. En general alcanzan su objetivo en tiempo finito, aunque hay excepciones, por ejemplo los programas de control de un sistema de alarma poseen requerimiento de tiempo infinito. Un programa sin errores que se ejecuta puede no ser correcto si no cumple con los requerimientos.

Definición
**Programa**: conjunto de instrucciones no activas almacenadas en un computador, se vuelve **tarea** a partir de que se selecciona para su ejecución y permite cumplir una función específica. Un **proceso** es un programa en ejecución.

En principio las tareas más importantes a la que se enfrenta quien debe escribir programas en computadoras son:

1. Definir el conjunto de instrucciones cuya ejecución ordenada conduce a la solución.
2. Elegir la representación adecuada de los datos del problema.

La función esencial del especialista informático es explotar el potencial de las computadoras para resolver situaciones del mundo real. Para esto debe analizar los problemas del mundo real, ser capaz de sintetizar sus aspectos principales y poder especificar la función objetivo que se desee. Posteriormente debe expresar la solución en forma de programa, manejando los datos del mundo real mediante una representación valida para una computadora.

## Partes de un programa

Los componentes básicos son las instrucciones y los datos. Las instrucciones o sentencias representan las operaciones que se ejecutaran al interpretar el programa. Todos los lenguajes de programación tienen un conjunto mínimo de operaciones que son las de asignación, selección e iteración. Un lenguaje con solo estas tres instrucciones permite escribir cualquier algoritmo.
Los datos son valores de información de los que se necesita disponer, en ocasiones transformar para ejecutar la función del programa.
Los datos están representados simbólicamente por un nombre que se asocia con una dirección única de memoria.
El contenido de la dirección de memoria correspondiente a un dato constante se asigna solo una vez y solo puede ser modificado en una nueva compilación. En cambio el contenido o valor de la dirección de memoria correspondiente a un dato variable puede ser asignado o modificado en tiempo de ejecución.
Un programa se corresponde con una transformación de datos. A partir de un contexto determinado por las precondiciones.

El programa transforma la información debiendo llegar al resultado esperado produciendo el nuevo contexto caracterizado por las poscondiciones.

## Dato

Representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.

**Definición**
Dato representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.
<dato> -> <objeto><atributo><valor>

## Abstracción

Proceso de análisis del mundo real con el propósito de interpretar los aspectos esenciales de un problema y expresarlo en términos precisos.

## Modelizacion

Abstraer un problema del mundo real y simplificar su expresión, tratando de encontrar los aspectos principales que se pueden resolver, requerimientos, los datos que se han de procesar y el contexto del problema.

## Precondición

Información conocida como verdadera antes de iniciar el programa.

## Poscondición

Información que debiera ser verdadera al cumplir un programa, si se cumple adecuadamente el requerimiento pedido.

## Especificación

Proceso de analizar problemas del mundo real y determinar en forma clara y concreta el objetivo que se desea. Especificar un problema significa establecer en forma univoca el contexto, las precondiciones el resultado esperado, del cual se derivan las poscondiciones.

## Lenguaje de programación

Conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico para la expresión de soluciones de problemas.

## Del problema real a su solución por computadoras

Analizando un problema del mundo real se llega a la *modelización* del problema por medio de la *abstracción*.

A partir del modelo se debe elaborar el análisis de la solución como sistema, esto significa la descomposición en módulos. Estos módulos deben tener una función bien definida.

La modularización es muy importante y no solo se refiere a los procesos a cumplir, sino también a la distribución de los datos de entrada, salida y los datos intermedios necesarios para alcanzar la solución.

**Estudio de los datos del problema.**

Cada módulo debe tener un proceso de refinamiento para expresar su solución en forma ordenada, lo que llevara a la construcción del algoritmo correspondiente.

A partir de los algoritmos se pueden escribir y probar programas en un lenguaje determinado y con un conjunto de datos significativos.

**Etapas de resolución de problemas con computadoras**.

1. Análisis del problema: en su contexto del mundo real.

2. Diseño de la solución: Lo primero es la modularización del problema, es decir la descomposición en partes con funciones bien definidas y datos propios estableciendo la comunicación entre los módulos.
3. Especificación del algoritmo: La elección adecuada del algoritmo para la función de cada modulo es vital para la eficiencia posterior.
4. Escritura del programa: Un algoritmo es una especificación simbólica que debe convertirse en un programa real sobre un lenguaje de programación concreto.
5. Verificación: una vez escrito el programa en un lenguaje real y depurado los errores sintácticos se debe verificar que su ejecución conduzca al resultado deseado con datos representativos del problema real.

**Programación modular – programación estructurada**

Se dice modular porque permite la descomposición del problema en módulos y estructurada solo permite la utilización de tres estructuras: Asignación, selección, repetición.

**Algoritmo**

El termino algoritmo es en honor del matemático árabe del siglo IX, *Abu Jafar Mohamed ibn Musa Al Khowârizmî*. Refiere conjunto de reglas, ordenadas de forma lógica, finito y preciso para la solución de un problema, con utilización o no de un computador.

En la actualidad al término se lo vincula fuertemente con la programación, como paso previo a la realización de un programa de computación aunque en realidad es una metodología de resolución presente en muchas de las actividades que se desarrolla a lo largo de la vida.

Desde los primeros años de escuela se trabaja con algoritmos, en especial en el campo de las matemáticas. Los métodos utilizados para sumar, restar, multiplicar y dividir son algoritmos que cumplen perfectamente las características de precisión, finitud, definición y eficiencia.

Para que el algoritmo pueda ser fácilmente traducido a un lenguaje de programación y luego ser ejecutado la especificación debe ser clara, precisa, que pueda ser interpretada con precisión y corresponda a pocas acciones, si esto no ocurre será necesario acudir a desarrollar un mayor nivel de refinamiento.

La utilización de refinamientos sucesivos es lo que permite alcanzar la solución modular que se propone.

Diseño modular, entonces, es la aplicación del criterio de refinamientos sucesivos, partiendo de un plan de acción, determinando que hacer, por aplicación de los conocimientos estratégicos de resolución pasando luego al como hacerlo con los conocimientos tácticos para la realización del algoritmo.

La programación de algoritmos representa un caso de resolución de problemas que requiere representación mental del mundo real, adaptación para tener una solución computable y criterio para elegir una alternativa eficiente de implementación.

Cuando se analiza un problema, particularmente de programación, y éste es difícil de describir, el plan de acción recomendable para alcanzar la solución es comenzar trazando un esbozo de las formas más gruesas, para que sirvan de andamio a las demás; aunque algunas de ellas se deban cambiar posteriormente. Después, se agregan los detalles, (obteniéndose el algoritmo refinado), para dotar a estos esqueletos de una estructura más realista.

Durante la tarea de integración final, se descartan aquellas primeras ideas provisionales que ya no encajan en la solución. Por lo que, hasta que no se haya visto el conjunto global es imposible encontrarle sentido a ninguna de las partes por sí solas.

Siempre es mejor explicar un misterio en términos de lo que se conoce, pero cuando esto resulta difícil de hacer, se debe elegir entre seguir tratando de aplicar las antiguas teorías, o de descartarlas y probar con otras nuevas. Siguiendo este análisis, se define como reduccionistas a aquellas personas que prefieren trabajar sobre la base de ideas existentes, y como renovadores a

los que les gusta impulsar nuevas hipótesis. En programación debe encontrarse un equilibrio entre ambas posturas.
La programación como toda actividad que requiere creatividad necesita que se produzca un salto mental que se puede sintetizar como señala David Perkins en:

1. Larga búsqueda, se requiere esfuerzo en analizar y buscar.
2. Escaso avance aparente: el salto mental sobreviene tras un avance que parece escaso o no muy evidente, pero sobreviene.
3. Acontecimiento desencadenante: El típico proceso de hacer clic comienza con un acontecimiento que lo desencadena.
4. Chasquido cognitivo: De pronto aparece la solución la que sobreviene con rapidez que hace que las piezas encajen con precisión, aun cuando sea necesario todavía ajustar algunos detalles. Pero la idea generadora apareció.
5. Transformación. Este avance nos va modificando nuestro mundo mental.

En síntesis, la practica del salto de pensamiento requiere en primer lugar de buscar analogías, en segundo lugar juegan un papel importante las conexiones lógicas, formulación de una pregunta crucial ocupa un papel decisivo. El repertorio de acciones tras el salto del pensamiento se expande para incluir no solo la analogía sino una extrapolación lógica y la formulación de la pregunta adecuada
Para encontrar una solución muchas veces se necesita desplazarse bastante por el entorno adecuado. Conforme a esto Thomas Edison declaro que la invención significa 99% de transpiración y 1% de inspiración, en contraposición con Platón que sostenía que las soluciones aparecen por inspiración divina.
Muchos problemas son razonables, cabe razonarlos paso a paso para alcanzar la solución. Otros son irrazonables no se prestan a un a reflexión por etapas.

Definición
**Algoritmo**
Especificación rigurosa (debe expresarse en forma univoca) de la secuencia de pasos, instrucciones, a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito. Esto último supone que el algoritmo empieza y termina, en el caso de los que no son de tiempo finito (ej. Sistemas en tiempo real) deben ser de número finito de instrucciones.

En definitiva un algoritmo es una especificación ordenada de la solución a un problema de la vida real. Son el fundamento de la programación de computadores en el paradigma de programación imperativo.
Bajo este paradigma desarrollar un programa significa indicarle al computador, con precisión, sin ambigüedad y en un lenguaje que este pueda entender, todos y cada uno de los pasos que debe ejecutar para lograr el objetivo propuesto.
Previo a la traducción en un lenguaje de programación es necesario poder entender el problema, conocer las pre condiciones, establecer cual debe ser la pos condición, o aquello que debe ser cierto al finalizar la ejecución del algoritmo, en definitiva entender claramente *Que* es lo que se debe hacer para luego avanzar en *Como* hacerlo. Aquí debe utilizarse todas las herramientas al alcance de la mano para el desarrollo del algoritmo como paso previo a la solución del problema por el computador.
Existen varias técnicas para representar formalmente un algoritmo, una descriptiva llamada pseudo código, y otras graficas como los diagrama de flujo, diagrama Nassi Sneiderman, Diagramas de Lindsay, diagramas de Jackson, entre otros, en este caso se presentara una notación algorítmica similar a la presentada por Piere Scholl en el texto Esquemas algorítmicos fundamentales: Secuencia e iteración.

Definición
**Algoritmo:**
Secuencia finita de instrucciones, reglas o pasos que describen en forma precisa las operaciones que una computadora debe realizar para llevar a cabo una tarea en tiempo finito [Knuth, 1968].
Descripción de un esquema de comportamiento expresado mediante un repertorio finito de acciones y de informaciones elementales, identificadas, bien comprendidas y realizables a priori. Este repertorio se denomina léxico[Scholl, 1988].

Esta formado por reglas, pasos e instrucciones.
Las reglas especifican operaciones.
La computadora es el agente ejecutor.
La secuencia de reglas y la duración de la ejecución son finitas.

## Características de un algoritmo

Un algoritmo debe tener al menos las siguientes características:

1. **Ser preciso**: esto significa que las operaciones o pasos del algoritmo deben desarrollarse en un orden estricto, ya que el desarrollo de cada paso debe obedecer a un orden lógico.
2. **Ser definido**. Ya que en el área de programación, el algoritmo es el paso previo fundamental para desarrollar un programa, es necesario tener en cuenta que el computador solo desarrollará las tareas programadas y con los datos suministrados; es decir, no puede improvisar y tampoco inventará o adivinará el dato que necesite para realizar un proceso. Por eso, el algoritmo debe estar plenamente definido; esto es, que cuantas veces se ejecute, el resultado depende estrictamente de los datos suministrados. Si se ejecuta con un mismo conjunto de datos de entrada, el resultado deberá ser siempre el mismo.
3. **Ser finito**: esta característica implica que el número de pasos de un algoritmo, por grande y complicado que sea el problema que soluciona, debe ser limitado. Todo algoritmo, sin importar el número de pasos que incluya, debe llegar a un final. Para hacer evidente esta característica, en la representación de un algoritmo siempre se incluyen los pasos inicio y fin.
4. **Presentación formal**: para que el algoritmo sea entendido por cualquier persona interesada es necesario que se exprese en alguna de las formas comúnmente aceptadas; pues, si se describe de cualquier forma puede no ser muy útil ya que solo lo entenderá quien lo diseñó. Las formas de presentación de algoritmos son: el pseudo código, diagrama de flujo y diagramas de Nassi/Schneiderman, entre otras. En esta publicación se propondrá una notación algorítmica y se darán las equivalencias entre la propuesta y las existentes y también con las sentencias de los lenguajes de programación, en particular Pascal y C.
5. **Corrección**: el algoritmo debe ser correcto, es decir debe satisfacer la necesidad o solucionar el problema para el cual fue diseñado. Para garantizar que el algoritmo logre el objetivo, es necesario ponerlo a prueba; a esto se le llama verificación o prueba de escritorio.
6. **Eficiencia**: hablar de eficiencia o complejidad de un algoritmo es evaluar los recursos de cómputo que requiere para almacenar datos y para ejecutar operaciones frente al beneficio que ofrece. En cuanto menos recursos requiere será más eficiente el algoritmo.

La vida cotidiana está llena de soluciones algorítmicas, algunas de ellas son tan comunes que no se requiere pensar en los pasos que incluye la solución. La mayoría de las actividades que se realizan

diariamente están compuestas por tareas más simples que se ejecutan en un orden determinado, lo cual genera un algoritmo. Muchos de los procedimientos utilizados para desarrollar tareas cotidianas son algorítmicos, sin embargo, esto no significa que todo lo que se hace está determinado por un algoritmo.
El primer paso en el diseño de un algoritmo es conocer la temática a tratar, el segundo será pensar en las actividades a realizar y el orden en que deben ejecutarse para lograr el objetivo, el tercero y no menos importante es la presentación formal.

## Propiedades de los algoritmos

1. Especificación precisa de la entrada: El algoritmo debe dejar claro el número y tipo de datos de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.
2. Especificación precisa de cada instrucción: cada etapa del algoritmo debe estar definida con precisión, no debe haber ambigüedades sobre las acciones que se deben ejecutar en cada momento.
3. Un algoritmo debe ser exacto y correcto: Un algoritmo se espera que resuelva un problema y se debe poder demostrar que eso ocurre. Si las condiciones de entrada se cumplen y se ejecutan todos los pasos el algoritmo entonces debe producir la salida deseada.
4. Un algoritmo debe tener etapas bien definidas y concretas, un número finito de pasos, debe terminar y debe estar claro la tarea que el algoritmo debe ejecutar.
5. Debe ser fácil de entender, codificar y depurar.
6. Debe hacer uso eficiente de los recursos de la computadora

Finitud: en longitud y duración.
Precisión: Determinar sin ambigüedad las operaciones que se deben ejecutar.
Efectividad: las reglas pueden ejecutarse sin el ordenador obteniéndose el mismo resultado.
Generalidad: Resolver una clase de problema y no un problema particular.
Entradas y salidas: puede tener varias entradas pero una sola salida, el resultado que se debe obtener.

## Eficiencia de un algoritmo

Se pueden tener varias soluciones algorítmicas para un mismo problema, sin embargo el uso de recursos y la complejidad para cada una de las soluciones puede ser muy diferente.
La eficiencia puede definirse como una métrica de calidad de los algoritmos asociada con la utilización optima de los recursos del sistema de cómputo donde se ejecutara el algoritmo, su claridad y el menor grado de complejidad que se pueda alcanzar. *Hacer todo tan simple como se pueda, no más (Albert Einstein).*
La eficiencia como factor espacio temporal debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad del mantenimiento.
Medidas de eficiencia para N = 10.0000

| Eficiencia | | Iteraciones | Tiempo estimado |
|---|---|---|---|
| Logarítmica | $Log_2\ N$ | 14 | Microsegundos |
| Lineal | N | 10.000 | 0.1 segundo |
| Logarítmica lineal | $N * Log_2\ N$ | 140.000 | 2 segundos |
| Cuadrática | $N^2$ | $10.000^2$ | 15-20 minutos |
| Poli nómica | $N^k$ | $10.000^K$ | Horas |
| Exponencial | $2^N$ | $2^{10.000}$ | Inmedible |

## Complejidades más comunes

1. Complejidad constante: se expresa como O(1). Se encuentra en algoritmos sin ciclos, por ejemplo en un intercambio de variables.
2. Complejidad logarítmica: Es una complejidad eficiente, la búsqueda binaria tiene esta complejidad.
3. Complejidad lineal: se encuentra en los ciclos simples.
4. Complejidad logarítmica lineal: Los mejores algoritmos de ordenamiento tienen esta complejidad.
5. Complejidad cuadrática: Aparece en el manejo de matrices de dos dimensiones, generalmente con dos ciclos anidados.
6. Complejidad cúbica: Aparece en el manejo de matrices de tres dimensiones, generalmente con tres ciclos anidados.
7. Complejidad exponencial: es la complejidad de algoritmos recursivos.

## Léxico y algoritmo

Para escribir un algoritmo deben seguirse un conjunto de pasos básicos

1. Comprender el problema
2. Identificar los elementos a incluir en el léxico: constantes, tipos, variables y acciones.
3. Encontrar la forma de secuenciar las acciones para obtener el resultado, esto es, alcanzar las poscondiciones a partir de un estado inicial que cumple con la precondición. Para establecer el orden de las acciones los lenguajes de programación proporcionan mecanismos de composición: Secuenciación, análisis de casos, iteración y recursion.
4. Al organizar las acciones en el tercer paso puede ocurrir que se detecte que faltan elementos en el léxico o que algún aspecto del problema no ha sido bien comprendido lo cual requeriría volver a los pasos anteriores.
5. Nunca la solución aparece en el primer intento, en general aparece en un proceso cíclico, entonces se debe:
6. Escribir el léxico,
    a. escribir la primera versión,
    b. incluir en el léxico nuevos elementos que faltaban,
    c. escribir la nueva versión del algoritmo y así sucesivamente

## Estructura de un algoritmo

```
LEXICO {Léxico Global del algoritmo}
        {Declaración de tipos, constantes, variables y acciones}
        Acción 1
                PRE {Precondición de la acción 1}
                POS {Poscondición de la acción 1}
                LEXICO {Léxico local, propio de la acción 1}
                        Declaraciones locales
                ALGORITMO {Implementación de la acción 1}
                        {Secuencia de instrucciones de la acción 1}
                FIN {Fin implementación algoritmo de la acción 1}
ALGORITMO
        PRE {Precondición del algoritmo principal}
        POS {Poscondición del algoritmo principal}
```

{Secuencia de instrucciones del algoritmo principal}
FIN {Fin del algoritmo principal}

## Proceso Computacional

Se refiere a un algoritmo en ejecución. La ejecución de las instrucciones origina una serie de acciones sobre elementos de memoria que representan información manejada por el algoritmo. A nivel algorítmico se asigna un nombre a cada información de modo de manejar un par nombre-valor para cada información.
Una variable representa alguna entidad del mundo real, relevante para el problema que se quiere resolver. El efecto que producen las acciones del proceso sobre las variables produce cambio de estados o de sus valores.

**Definiciones**
**Programa**: Algoritmo escrito en un lenguaje cuyas instrucciones son ejecutables por una computadora y que están almacenados en un disco.
**Tarea**: Un programa se vuelve tarea a partir del momento que se lo selecciona para su ejecución y hasta que esta termina.
**Proceso:** programa en ejecución, se ha iniciado pero aún no ha finalizado.
**Lenguajes de programación**: notación que permite escribir programas a mayor nivel de abstracción que los lenguajes de máquina. Sus instrucciones deben ser traducidas a lenguaje de máquina.
**Lenguaje de máquina**: Instrucciones que son ejecutables por el hardware de una computadora.
**Paradigmas de programación**
Paradigma: Colección de conceptos que guían el proceso de construcción de un programa. Estos conceptos controlan la forma en que se piensan y formulan los programas.
Imperativo – Procedural – Objetos.
Declarativo – Funcional – Lógico.
**Dato Información Conocimiento**
Dato: <objeto><atributo><valor> sin interpretar.
Información: añade significado al dato.
Conocimiento: Añade propósito y capacidad a la información. Potencial para generar acciones.
**Problema**
Enunciado con una incógnita, la solución es encontrar el valor de esa incógnita.
Problema computacional o algorítmico: tarea ejecutada por una computadora con una especificación precisa de los datos de entrada y de los resultados requeridos en función de estos.
**Clase de problemas**
No computables: No existe un algoritmo.
Computables
Tratables: Existe un algoritmo eficiente.
Intratable: No existe algoritmo eficiente.
**Expresiones Sentencias Léxico**
Expresiones: secuencia de operadores y operandos que se reduce a un solo valor.
Sentencias: acción produce un efecto, puede ser primitiva o no primitiva.
Léxico: Descripción del conjunto de acciones e informaciones a partir de la cual se expresa el esquema de comportamiento del algoritmo.
**Pasos para resolver un algoritmo**
Comprender el problema.
Identificar información y acciones a incluir en el léxico (constantes, tipos, variables y acciones).

Encontrar un camino de secuenciar las acciones para obtener el resultado, es decir para alcanzar la poscondición a partir del estado inicial que cumple con la precondición.

**Acciones primitivas y derivadas**

Acciones primitivas: Incorporadas por el lenguaje.

Acciones derivadas: realizadas mediante la combinación de acciones primitivas con el objeto de desarrollar una tarea en particular. Son complementarias y pueden ser desarrolladas por el programador.

**Estructura de un algoritmo**

```
LEXICO {Léxico Global del algoritmo}
        {Declaración de tipos, constantes, variables y acciones}
        Acción 1
                PRE {Precondición de la acción 1}
                POS {Poscondición de la acción 1}
                LEXICO {Léxico local, propio de la acción 1}
                        Declaraciones locales
                ALGORITMO {Implementación de la acción 1}
                        {Secuencia de instrucciones de la acción 1}
                FIN {Fin implementación algoritmo de la acción 1}

ALGORITMO
        PRE {Precondición del algoritmo principal}
        POS {Poscondición del algoritmo principal}
                {Secuencia de instrucciones del algoritmo principal}
FIN {Fin del algoritmo principal}
```

**Resumen:**

En el presente capitulo se introdujeron términos y frases de la disciplina en estudio.

Se abordo el problema desde un punto de vista conceptual definiendo con precisión términos y frases para evitar ambigüedades. Se puso especial énfasis en la necesidad de pensar las soluciones antes de escribir código. Se establecieron cuales son los pasos que deben seguirse para una solución correcta de problemas y cuales son los problemas que pueden tratarse en forma algorítmica. Se definió cuales son las características deseables de los algoritmos y se introdujo el concepto de eficiencia de modo de hacer, las cosas tan simple como se pueda.

Se introdujo, además una representación semi formal para la descripción de los algoritmos

# Introduccion a C++

# 2

**Objetivos de aprendizaje**
Dominando los temas del presente capitulo Usted podrá.

1. Acceder a las características importantes de C++
2. La estructura general de un programa
3. Utilización de objetos flujo de salida y de entrada para la creación de programas simples

Fundamentos C++

| Sin declaración using | Con declaración using |
|---|---|
| //programa para imprimir texto<br>#include <iostream><br><br><br><br><br>int main()<br>{<br>  std::cout << “Hola\n”;<br><br>  return 0;<br>} | //programa para imprimir texto<br>#include <iostream><br>using std::cout; //<br>using std::cin;<br>using std::endl;<br><br>int main()<br>{<br>  cout << “Hola” << endl;<br><br>  return 0; |

| Instruccion | Descripcion |
|---|---|
| #include | Directiva del preprocesadpr |
| <iostream> | Componente de entrada/salida (objetos cin, cout,cerr) |
| using | Declaración que elimina necesidad de repetir el prefijo std. |
| int main() | Funcion principal que retorna un entero |
| { } | Definición de un bloque de programa |
| std::cout | Uso del nombre cout del espacio de nombres std, dispositivo std de salida |
| :: | Operador binario de resolución de alcance |
| << | Operador de inserción en flujo |
| “Hola\n” | Literal Hola + salto de línea (también << std::endl; |
| ; | Finalización de una sentencia |
| return 0 | Punto de finalización correcta de la función |

Si la función, como en este caso tiene un encabezado int main() debe tener al menos un return de un valor entero. Una función void nombre() puede finalizar con la instrucción return o sin ella.

Programa que muestra la suma de dos enteros

```
# include <iostream>
int main()
{
   // declaracion de variables
   int numero1;
   int numero2;
   int suma;

   std::cout << “Escriba el primer entero”;
   std::cin >>numero1;

   std::cout << “Escriba el segundo entero”;
   std::cin >>numero2;

   suma = numero1 + numero2;

   std::cout << “La suma de “ numero1 << “ + ” <<numero2 << “ es: “ << suma << std::endl;
   return 0;
}
```

| Instrucción | Descripcion |
|---|---|
| Cin | Dispositivo std de entrada |
| >> | Operador de extracción de flujo |
| + | Operador de suma |
| - | Operador de resta |
| * | Operador multiplicativo |
| / | Operador de división |
| % | Operador de modulo o resto |
| ( ) | Operador para agrupar expresiones ej: a * (b+c) |
| == | Operador de igualdad |
| > | Mayor |
| >= | Mayor igual |
| < | Menor |
| >= | Menor igual |
| != | Operador de desigualdad |
| = | Operador de asignación |
| += | Asignación y suma x+=3; equivale a x = x + 3; |
| -= | Resta y asignación |
| *= | Multiplicación y asignación |
| /= | División y asignación |
| ++ | Operador de incremento |
| -- | Operador de decremento |

Programa que comprara dos enteros, utiliza la declaración using

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    int numero1;
    int numero2;
    cout << “Escriba dos enteros para comparar”;
    cin >>numero1 >> numero2;

    if (numero1 > numero2)
    cout <<  numero1 << “ > ” <<numero2 << std::endl;

    if (numero1 == numero2)
    cout <<  numero1 << “ == ” <<numero2 << std::endl;

    if (numero1 < numero2)
    cout <<  numero1 << “ < ” <<numero2 << std::endl;
    return 0;
}
```

**Palabras Reservadas**

**C y C++**

Auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while

**Solo C++**

and and_eq asm bitand bitor bool catch class compl const_cast delete dynamic_cast explicit export false friend inline mutable namespace new not not_eq operator or or_eq private protected public reinterpret_cast static_cast template this throw true try typeid typename using virtual wchar_t xor xor_eq

Tipos de datos fundamentals y jerarquia de promoción

| Tipos de Datos |
|---|
| long double |
| double |
| float |
| unsigned long int |
| long int |
| unsigned int |
| int |
| unsigned short int |
| short int |
| unsigned char |
| char |
| bool |

**Espacios de nombres**

# include <iostream>

Sin incluir directiva ni declaracion using

…..

std::cout << “Hola” << std::endl;

……..

#include <iostream>

Con declaracion using

//la declaración using de cada elemento solo incorpora los nombres especificados

using std::cin; usa el objeto cin del espacio std

using std::endl;

cout << “Hola” <<endl;

……

#include <iostream>

Con directiva using

//la directiva using incorpora la totalidad de los nombres del espacio de nombre

using namespace std; usa el espacio de nombre std

cout << “Hola” <<endl;

……

Espacios de nombres

Un programa incluye muchos identificadores definidos con distintos alcances. Una variable de un alcance se “traslapa” si entra en conflicto con una del mismo nombre y distinto alcance.para evitar esto se utilizan los espacios de nombres.

Para acceder a un miembro de un espacio de nombre se lo califica con el nombre del espacio, el operador de resolución de alcance y el nombre del miembro std::cout ….

Otras forma de uso de los espacios de nombre es mediante declaración using o directiva using.

**//demostración de espacios de nombre**

```
#include <iostream>
using namespace std; //usa el espacio de nombre std
int entero1 = 98 //variable global
//crea espacio de nombres ejemplo
namespace Ejemplo//declara dos constantes y una variable
{
 const doublé PI = 3.14;
 const doublé E  = 2.71;
 int entero1 = 8;
 void imprimirValores(); //prototipo
      namespace interno// espacio de nombres anidado
      {
         enum Anios {FISCAL1 = 1990, FISCAL2, FISCAL3};
       } // fin espacio de nombre interno
} //fin espacio de nombre Ejemplo

namespace //crea espacio de nombre sin nombre
{
    Doublé doubleSinNombre = 88.22;
}
```

```
Int main ()
   cout << doubleSinNombre;
   cout << entero1; //imprime la variable global
   cout << Ejemplo::entero1; // imprime entero1 del espacio de nombre Ejemplo
   Ejemplo::imprimirValores(); //invoca a la función
   Cout << Ejemplo::Interno::FISCAL1; //imprime el valor del espacio de nombre interno
```

Resumen:

1. Los comentarios de una linea comienzan con //, esta línea es omitida por el compilador.
2. Las directivas del preprocesador comienzan con #, estas líneas no terminan con ; ya que no son parte de C. Permiten incorporar archivos de encabezado. <iostream> contiene información necesaria para utilizar cin y cout-
3. Los programas en C ejecutan la función main.
4. Toda sentencia termina con ;
5. La declaración using std::cout informa al compilador que puede encontrar a cout en el espacio de nombre std y elimina la necesidad de repetir el prefijo std.

## Estructuras de control

**Objetivos de aprendizaje**
Dominando los temas del presente capitulo Usted podrá.

1. Comprender técnicas básicas de descomposición
2. Utilizar análisis de casos, o estructuras de selección
3. Utilizar repeticiones
4. Controlar repeticiones

Sentencia de selección if

```
if (expresión) {lista de sentencias};[else {lista de sentencias}]
Sin clausula else
if (nota >= 4){
   cout << “Aprobado”;
}

Con clausula else
if (nota >= 4){
   cout << “Aprobado”;
}
else{
  cout << “No aprobado”;
}

Anidados
If (nota > 9)
    cout << “Distinguido”;
else if (nota > 8)
    cout << “Sobresaliente”;
else if(nota > 6)
```

```
    cout << “Muy bueno”;
else if (nota >= 4)
    cout << “Bueno”;
else
    cout << “No aprobo”;
```

Estructura de repeticion

while

```
int i = 0;
while (I < 10 ){
    i++;
}
```

for

```
for (int I = 0; I < 10;i++)
        ;
```

Desarrollar una funcion que calcule el promedio de varios valores

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int calcularPromedio() //con cantidad conocida a priori ejemplo 10 valores
{
    Int cantidad, valor, suma;
    Suma = 0;
    cantidad = 1;
    while  (cantidad <= 10){
        cout >> “Ingrese un valor “;
        cin >> valor; cout << endl;
        suma +=valor;
        cantidad ++;
    }
    return suma/cantidad;
}
Alternative con ciclo for
    for(cantidad = 1, suma = 0; cantidad <=10; cantidad ++){
        ……..
int calcularPromedio() //con fin por valor centinela
{
    Int cantidad, valor, suma;
    Suma = 0;
    cantidad = 0;
    cout >> “Ingrese un valor “;
    cin >> valor;
    while  (valor > 0){
        suma +=valor;
        cantidad ++;
        cout >> “Ingrese un valor “;
```

```
        cin >> valor;
    }
    return (cantidad > 0?suma/cantidad: 0;
}
Alternative con ciclo for
    for(cantidad = 0, suma = 0; valor > 0; cantidad ++){
```

## Ejercicios

1. Cuál de las siguientes sentencias son correctas para la ecuación algebraica $y=ax^3 + 7$.
    a. y = a * x * x * x + 7
    b. y = a * x * x * (x + 7)
    c. y = (a * x) * x *( x + 7)
    d. y = (a * x) * x * x + 7
    e. y = a * (x * x * x) + 7
    f. y = a * (x * x * x + 7)
2. Escriba un programa que pida al ususrio dos números e informe la suma, la resta, el producto y el cociente de los mismos
3. Imprima un programa que imprima los números del 1 al 4 en una misma línea, hágalo de las formas siguientes:
    a. Utilizando un solo operador de inserción de flujo
    b. Una única sentencia con 4 operadores de inserción de flujo
    c. Utilizando cuatro sentencias
4. Escriba un programa que reciba tres números por el teclado e imprima la suma, el promedio, el producto, el mayor y el menor de esos números. Escriba un adecuado dialogo en pantalla.
5. Escriba un programa que reciba un numero que represente el radio de un circulo e imprima el diámetro, circunferencia y área.
6. Que imprime el siguiente código
    a. std::cout << "*\n**\n***\n****" <<std::endl;
    b. std::cout << 'A';
    c. std::cout << static_cast< int > 'A'; (que es static_cast? Investigue.)
7. Utilizando solo lo que hemos desarrollado en esta introduccion escriba un programa que calcule los cuadrados y los cubos de los números de 0 a 10 y los muestre por pantalla.
8. Escriba un programa que reciba un numero entero de 5 digitos, que separe el numero en sus digitoe y los muestre por pantalla, uno por línea comenzando por elmas significacivo en la primera línea.

9. Dados dos valores enteros y positivos determinar y mostrar por el dispositivo estándar de salida: la suma, la resta, el producto y la división de los mismos. Analice precondiciones adecuadas y utilice leyendas adecuadas. Resuelva teniendo en cuenta las precondiciones y da una solución alternativa sin considerarlas.
10. Responda: que pasa si el conjunto de datos es float, si es entero y sin la restricción de ser positivo.

11. Dada una terna de números naturales <dia, mes, año> que representan al día, al mes y al año de una fecha informarla como un solo número natural de 8 dígitos (AAAAMMDD). Establezca precondiciones.
12. Dada un número natural de 8 dígitos, con formato (AAAAMMDD) descompóngalo en sus elementos lógicos y muéstrelos por el dispositivo estándar de salida.
13. Dado un valor entero determinar y mostrar: la quinta parte del mismo, el resto de la división por 5 y la tercera parte del valor del primer punto. Resuelva sin precondiciones, informando por la salida estándar los resultados.
14. Dado una terna de valores determine e imprima: El mayor y el menor del conjunto. Muestre por salida estándar con las aclaraciones que considere.
15. Que cambios estratégicos produciría si el conjunto de valores ingresados fueran 20? Y si el lote fuera de una cantidad no conocida a priori?
16. Dadas dos ternas de valores que representan las fechas validas de nacimiento de dos personas, indique cual de las dos corresponde al mayor. Utilice las leyendas que crea corresponden.
17. Dado un par de valores que representa una el mes y año de una fecha valida del siglo XXI determinar e imprimir la cantidad de días de ese mes.
18. Dado una terna de valores enteros y positivos determinar si los mismos forman un triangulo.
19. Dado una terna de valores que representan los lados de un triangulo determinar si el tipo de triangulo que forman.
20. Dados dos números naturales M y N, determinar e imprimir cuantos múltiplos de M hay el el conjunto 1 a N.
21. Dados dos números enteros, M y N informar su producto por sumas sucesivas.
22. Dados un conjunto de valores enteros informar el promedio de los mayores que45 y la suma de los menores que –10.
23. Dado un valor M determinar e imprimir los M primeros múltiplos de 3 que no lo sean de 5, dentro del conjunto de los números naturales.
24. Dados un conjunto de valores enteros y positivos determinar e informar el mayor
25. Dado un conjunto de N valores informar el mayor, el menor y en qué posición del conjunto fueron ingresados.
26. Dado un conjunto de valores reales, que finaliza con un valor nulo, determinar e imprimir (si hubieron valores): el máximo de los negativos y el minimo de los positivos

# Biblioteca Standard C[1]

# 3

## Definiciones Comunes <stddef.h>

Define, entre otros elementos, el tipo **size_t** y la macro **NULL**. Ambas son definidas, también en otros encabezados, como en **<stdio.h>**.

**size_t**
Tipo entero sin signo que retorna el operador **sizeof**. Generalmente **unsigned int**.

**NULL**
Macro que se expande a un puntero nulo, definido por la implementación. Generalmente el entero cero **0** ó **0L**, ó la expresión constante **(void*)0** .

## Manejo de Caracteres <ctype.h>

**int isalnum (int);**
Determina si el carácter dado **isalpha** o **isdigit** Retorna (ok ? ≠0 : 0).

**int isalpha (int);**
Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).

**int isdigit (int);**
Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).

**int islower (int);**
Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

**int isprint (int);**
Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)

**int isspace (int);**
Determina si el carácter dado es alguno de estos: espacio (**' '**), **'\n'**, **'\t'**, **'\r'**, **'\f'**, **'\v'**. Retorna (ok ? ≠0 : 0)

**int isupper (int);**
Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

**int isxdigit (int);**
Determina si el carácter dado es un dígito hexadecimal (**'0'**..**'9'**, **'a'**..**'f'** o **'A'**..**'F'**). Retorna (ok ? ≠0 : 0)

[1] En este apartado se resumen gran cantidad de funciones de ANSI C, es al solo efecto que puedan tenerlas agrupadas. No todas se utilizan en la materia. Las de uso mas frecuentes están resaltadas en rojo.

`int tolower (int c);`
Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula.
Retorna (mayúscula ? minúscula : `c`)
`int toupper (int c);`
Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula.
Retorna (minúscula ? mayúscula : `c`)

## Manejo de Cadenas <string.h>

Define el tipo `size_t` y la macro `NULL`, ver *Definiciones Comunes*.
`unsigned strlen (const char*);`
Cuenta los caracteres que forman la cadena dada hasta el 1er carácter `'\0'`, excluido. Retorna (longitud de la cadena).

## Concatenación

`char* strcat (char* s, const char* t);`
Concatena la cadena `t` a la cadena `s` sobre `s`. Retorna (`s`).
`char* strncat (char* s, const char* t, size_t n);`
Concatena hasta `n` caracteres de `t`, previos al carácter nulo, a la cadena `s`; agrega siempre un `'\0'`. Retorna (`s`).

## Copia

`char* strncpy (char* s, const char* t, size_t n);`
Copia hasta `n` caracteres de `t` en `s`; si la longitud de la cadena `t` es < `n`, agrega caracteres nulos en `s` hasta completar `n` caracteres en total; atención: no agrega automáticamente el carácter nulo. Retorna (`s`).
`char* strcpy (char* s, const char* t);`
Copia la cadena `t` en `s` (es la asignación entre cadenas). Retorna (`s`).

## Búsqueda y Comparación

`char* strchr (const char* s, int c);`
Ubica la 1ra. aparición de `c` (convertido a `char`) en la cadena `s`; el `'\0'` es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : `NULL`)
`char* strstr (const char* s, const char* t);`
Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : `NULL`).
`int strcmp (const char*, const char*);`
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)
`int strncmp (const char* s, const char* t, size_t n);`
Compara hasta `n` caracteres de `s` y de `t`. Retorna (como `strcmp`).
`char* strtok (char*, const char*);`
Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : `NULL`).

## Manejo de Memoria

`void* memchr(const void* s, int c, size_t n);`
Localiza la primer ocurrencia de `c` (convertido a un `unsigned char`) en los `n` iniciales caracteres (cada uno interpretado como `unsigned char`) del objeto apuntado por `s`. Retorna (ok ? puntero al carácter localizado : `NULL`).
`int memcmp (const void* p, const void* q, unsigned n);`
Compara los primeros `n` bytes del objeto apuntado por `p` con los del objeto apuntado por `q`. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)
`void* memcpy (void* p, const void* q, unsigned n);`
Copia `n` bytes del objeto apuntado por `q` en el objeto apuntado por `p`; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (`p`).
`void* memmove (void* p, const void* q, unsigned n);`
Igual que `memcpy`, pero actúa correctamente si los objetos se superponen. Retorna (`p`).
`void* memset (void* p, int c, unsigned n);`
Inicializa los primeros `n` bytes del objeto apuntado por `p` con el valor de `c` (convertido a `unsigned char`). Retorna (`p`).

## Utilidades Generales <stdlib.h>

### Tips y Macros

`size_t`
`NULL`
Ver *Definiciones Comunes*.
`EXIT_FAILURE`
`EXIT_SUCCESS`
Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de `exit` ó valores de retorno de `main` para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.
`RAND_MAX`
Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función `rand`, como mínimo su valor debe ser 32767.

### Conversión

`double atof (const char*);`
Convierte una cadena que representa un real `double` a número `double`. Retorna (número obtenido, no necesariamente correcto).
`int atoi (const char*);`
Convierte una cadena que representa un entero `int` a número `int`. Retorna (número obtenido, no necesariamente correcto) .
`long atol (const char*);`
Convierte una cadena que representa un entero `long` a número `long`. Retorna (número obtenido, no necesariamente correcto).
`double strtod (const char* p, char** end);`
Convierte como `atof` y, si el 2do. argumento no es `NULL`, un puntero al primer carácter no convertible es colocado en el objeto apuntado por `end`. Retorna como `atof`.
`long strtol (const char* p, char** end, int base);`
Similar a `atol` pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver `strtod` por el parámetro `end`. Retorna como `atol`.
`unsigned long strtoul (const char* p, char** end, int base);`
Igual que `strtol` pero convierte a `unsigned long`. Retorna como `atol`, pero `unsigned long`.

## Administración de Memoria

```
void* malloc (size_t t);
```
Reserva espacio en memoria para almacenar un objeto de tamaño **t**. Retorna (ok ? puntero al espacio reservado : **NULL**)
```
void* calloc (size_t n, size_t t);
```
Reserva espacio en memoria para almacenar un objeto de **n** elementos, cada uno de tamaño **t**. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : **NULL**)
```
void free (void* p);
```
Libera el espacio de memoria apuntado por **p**. No retorna valor.
```
void* realloc (void* p, size_t t);
```
Reubica el objeto apuntado por **p** en un nuevo espacio de memoria de tamaño **t** bytes. Retorna (ok ? puntero al posible nuevo espacio : **NULL**).

## Números Pseudo-Aleatorios

```
int rand (void);
```
Determina un entero pseudo-aleatorio entre 0 y **RAND_MAX**. Retorna (entero pseudo-aleatorio).
```
void srand (unsigned x);
```
Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

## Comunicación con el Entorno

```
void exit (int estado);
```
Produce una terminación normal del programa. Todos los flujos con *buffers* con datos no escritos son escritos, y todos los flujos asociados a archivos son cerrados. Si el valor de **estado** es **EXIT_SUCCESS** se informa al ambiente de ejecución que el programa terminó exitosamente, si es **EXIT_FAILURE** se informa lo contrario. Equivalente a la sentencia **return estado**; desde la llamada inicial de **main**. Esta función *no retorna a su función llamante*.
```
void abort (void);
```
Produce una terminación anormal del programa. Se informa al ambiente de ejecución que se produjo una terminación no exitosa. Esta función *no retorna a su función llamante*.
```
int system (const char* lineadecomando);
```
Si **lineadecomando** es **NULL**, informa si el sistema posee un procesador de comandos. Si **lineadecomando** no es **NULL**, se lo pasa al procesador de comandos para que lo ejecute.
Retorna ( **lineacomando** ? valor definido por la implementación, generalmente el nivel de error del programa ejecutado : ( sistema posee procesador de comandos ? ≠0 : 0 ) ).

## Búsqueda y Ordenamiento

```
void* bsearch (
const void* k,
const void* b,
unsigned n,
unsigned t,
int (*fc) (const void*, const void*)
);
```
Realiza una búsqueda binaria del objeto ***k** en un arreglo apuntado por **b**, de **n** elementos, cada uno de tamaño **t** bytes, ordenado ascendentemente. La función de

comparación **fc** debe retornar un entero < 0, 0 o > 0 según la ubicación de ***k** con respecto al elemento del arreglo con el cual se compara. Retorna (encontrado ? puntero al objeto : **NULL**).

```
void qsort (
const void* b,
unsigned n,
unsigned t,
int (*fc) (const void*, const void*)
);
```

Ordena ascendentemente un arreglo apuntado por **b**, de **n** elementos de tamaño **t** cada uno; la función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según su 1er. argumento sea, respectivamente, menor, igual o mayor que el 2do. No retorna valor.

## Entrada / Salida <stdio.h>

### Tipos

**size_t**
Ver *Definiciones Comunes*.
**FILE**
Registra toda la información necesitada para controlar un *flujo*, incluyendo su *indicador de posición en el archivo*, puntero asociado a un *buffer* (si se utiliza), un *indicador de error* que registra sin un error de lectura/escritura ha ocurrido, y un *indicador de fin de archivo* que registra si el fin del archivo ha sido alcanzado.
**fpos_t**
Posibilita registrar la información que especifica unívocamente cada posición dentro de un archivo.

### Macros

**NULL**
Ver *Definiciones Comunes*.
**EOF**
Expresión constante entera con tipo **int** y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con **feof** y **ferror**.
**SEEK_CUR**
**SEEK_END**
**SEEK_SET**
Argumentos para la función **fseek**.
**stderr**
**stdin**
**stdout**
Expresiones del tipo **FILE*** que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

### Operaciones sobre Archivos

```
int remove(const char* nombrearchivo);
```

Elimina al archivo cuyo nombre es el apuntado por **nombrearchivo**. Retorna (ok ? 0 : ≠0)

```
int rename(const char* viejo, const char* nuevo);
```

Renombra al archivo cuyo nombre es la cadena apuntada por **viejo** con el nombre dado por la cadena apuntada por **nuevo**. Retorna (ok ? 0 : ≠0).

### Acceso

```
FILE* fopen (
              const char* nombrearchivo,
              const char* modo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** asociando un flujo con este según el **modo** de apertura. Retorna (ok ? puntero al objeto que controla el flujo : **NULL**).
```
FILE* freopen(
              const char* nombrearchivo,
              const char* modo,
              FILE* flujo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** y lo asocia con el flujo apuntado por **flujo**. La cadena apuntada por **modo** cumple la misma función que en **fopen**. Uso más común es para el redireccionamiento de **stderr**, **stdin** y **stdout** ya que estos son del tipo **FILE*** pero no necesariamente *lvalues* utilizables junto con **fopen**. Retorna (ok ? flujo : **NULL**).
```
int fflush (FILE* flujo);
```
Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por **flujo**. Su uso es imprescindible si se mezcla **scanf** con **gets** o **scanf** con **getchar**, si se usan varios **fgets**, etc. Retorna (ok ? 0 : **EOF**).
```
int fclose (FILE* flujo);
```
Vacía el *buffer* del flujo apuntado por **flujo** y cierra el archivo asociado. Retorna (ok ? 0 : **EOF**)

### Entrada / Salida Formateada

### Flujos en General

```
int fprintf (FILE* f, const char* s, ...);
```
Escritura formateada en un archivo ASCII. Retorna (ok ? cantidad de caracteres escritos : < 0).
```
int fscanf (FILE* f, const char*, ...);
```
Lectura formateada desde un archivo ASCII. Retorna (cantidad de campos almacenados) o retorna (**EOF** si detecta fin de archivo).

### Flujos stdin y stdout

```
int scanf (const char*, ...);
```
Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).
```
int printf (const char*, ...);
```
Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

### Cadenas

```
int sprintf (char* s, const char*, ...);
```
Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).
```
int sscanf (const char* s, const char*, ...);
```
Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

## Entrada / Salida de a Caracteres

```
int fgetc (FILE*); ó
int getc (FILE*);
```
Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).
```
int getchar (void);
```
Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).
```
int fputc (int c, FILE* f); ó
int putc (int c, FILE* f);
```
Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).
```
int putchar (int);
```
Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).
```
int ungetc (int c, FILE* f);
```
"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

## Entrada / Salida de a Cadenas

```
char* fgets (char* s, int n, FILE* f);
```
Lee, desde el flujo apuntado **f**, una secuencia de a lo sumo **n**-1 caracteres y la almacena en el objeto apuntado por **s**. No se leen más caracteres luego del carácter nueva línea o del fin del archivo. Un carácter nulo es escrito inmediatamente después del último carácter almacenado; de esta forma, s queda apuntando a una cadena. Importante su uso con **stdin**. Si leyó correctamente, **s** apunta a los caracteres leídos y retorna **s**. Sí leyó sólo el fin del archivo, el objeto apuntado por **s** no es modificado y retorna **NULL**. Si hubo un error, contenido del objeto es indeterminado y retorna **NULL**. Retorna ( ok ? **s** : **NULL**).
```
char* gets (char* s);
```
Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).
```
int fputs (const char* s, FILE* f);
```
Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).
```
int puts (const char* s);
```
Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

## Entrada / Salida de a Bloques

```
unsigned fread (void* p, unsigned t, unsigned n, FILE* f);
```
Lee hasta **n** bloques contiguos de **t** bytes cada uno desde el flujo **f** y los almacena en el objeto apuntado por **p**. Retorna (ok ? **n** : < **n**).
```
unsigned fwrite (void* p, unsigned t, unsigned n, FILE* f);
```
Escribe n bloques de t bytes cada uno, siendo el primero el apuntado por p y los siguientes, sus contiguos, en el flujo apuntado por f. Retorna (ok ? **n** : < **n**).

## Posicionamiento

```
int fseek (
FILE* flujo,
                long desplazamiento,
                int desde
);
```
Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR**

ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fsetpos (FILE* flujo, const fpos_t* posicion);
```

Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fgetpos (FILE* flujo, fpos_t* posicion);
```

Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).

```
long ftell (FILE* flujo);
```

Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : –1L).

```
void rewind(FILE *stream);
```

Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

## Manejo de Errores

```
int feof (FILE* flujo);
```

Chequea el *indicador de fin de archivo* del flujo apuntado por **flujo**. Contrastar con la macro **EOF** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de fin de archivo* activado ? ≠ 0 : 0).

```
int ferror (FILE* flujo);
```

Chequea el *indicador de error* del flujo apuntado por **flujo** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de error* activado ? ≠ 0 : 0).

```
void clearerr(FILE* flujo);
```

Desactiva los indicadores de fin de archivo y error del flujo apuntado por flujo. No retorna valor.

```
void perror(const char* s);
```

Escribe en el flujo estándar de error (**stderr**) la cadena apuntada por **s**, seguida de dos puntos (**:**), un espacio, un mensaje de error apropiado y por último un carácter nueva línea (**\n**). El mensaje de error está en función a la expresión **errno**. No retorna valor.

## Otros

## Hora y Fecha <time.h>

```
NULL
size_t
```

Ver *Definiciones Comunes*.

```
time_t
clock_t
```

Tipos aritméticos capaces de representar el tiempo. Generalmente **long**.

`CLOCKS_PER_SEC`
Macro que expande a una expresión constante de tipo `clock_t` que es el número por segundos del valor retornado por la función `clock`.
`clock_t clock(void);`
Determina el tiempo de procesador utilizado desde un punto relacionado con la invocación del programa. Para conocer el valor en segundos, dividir por `CLOCKS_PER_SEC`. Retorna (ok ? el tiempo transcurrido: `(clock_t)(-1)`.
`char* ctime (time_t* t);`
Convierte el tiempo de `*t` a fecha y hora en una cadena con formato fijo. Ejemplo: Mon Sep 17 04:31:52 1973\n\0. Retorna (cadena con fecha y hora).
`time_t time (time_t* t);`
Determina el tiempo transcurrido en segundos desde la hora 0 de una fecha base; por ejemplo: desde el 1/1/70. Retorna (tiempo transcurrido). Si `t` no es `NULL`, también es asignado a `*t`.

## Matemática

`int abs(int i);`
`long int labs(long int i);`
`<stdlib.h>` Calcula el valore del entero `i`. Retorna (valor absoluto de `i`).
`double ceil (double x);`
`<math.h>` Calcula el entero más próximo, no menor que `x`. Retorna (entero calculado, expresado como `double`).
`double floor (double x);`
`<math.h>` Calcula el entero más próximo, no mayor que `x`. Retorna (entero calculado, expresado como `double`).
`double pow (double x, double z);`
`<math.h>` Calcula $x^z$; hay error de dominio si $x < 0$ y $z$ no es un valor entero, o si $x$ es 0 y $z \neq 0$. Retorna (ok ? $x^z$ : error de dominio o de rango).
`double sqrt (double x);`
`<math.h>` Calcula la raíz cuadrada no negativa de `x`. Retorna ($x \geq 0.0$ ? raíz cuadrada : error de dominio).