

Carrera: Tecnicatura Superior en
Análisis de Sistemas Informáticos
(RESOL-4446-MEGC-16)

PROGRAMACIÓN I

Daniel Lombardero

Año 2018

ÍNDICE TEMÁTICO

| | |
|--|----------------|
| Presentación | 3 |
| <u>Unidad I: Introducción a la programación.</u> | 11 |
| 1.1. Concepto de algoritmo | 12 |
| 1.2. El lenguaje de programación | 15 |
| 1.3. Datos, tipos de datos y operaciones | 18 |
| 1.4. Constantes y variables | 21 |
| 1.5. Expresiones, tipos y operadores | 22 |
| 1.6. Operación de asignación | 25 |
| 1.7. Entrada y salida de datos | 25 |
| 1.8. Estructuras de selección | 26 |
| 1.9. Estructuras de repetición | 27 |
| 1.10. Estructuras anidadas | 29 |
| 1.11. Resolución del problema | 30 |
| 1.12. Diagramas de flujo | 32 |
| 1.13. Seudocódigo | 34 |
| <u>Unidad II: Algoritmos y programación estructurada.</u> | 37 |
| 2.1. Introducción | 37 |
| 2.2. Definiciones de programa y algoritmo | 38 |
| 2.3. Programas en C | 40 |
| Asignación | 42 |
| Delimitador | 46 |
| Instrucciones básicas en C | 47 |
| 2.4. Estructuras de datos | 85 |
| 2.5. Declaración de constantes | 101 |
| 2.6. Operaciones aritméticas | 102 |
| 2.7. Vectores | 106 |
| 2.8. Matrices | 107 |
| <u>Unidad III: Estructuras de datos compuestas.</u> | 113 |
| 3.1. Registros o estructuras | 113 |
| Vectores con registros | 117 |

| | | |
|-------------------------------------|---|-----|
| 3.2. | Utilización de subprogramas | 122 |
| | Introducción | 122 |
| | Funciones | 123 |
| 3.3. | Archivos | 130 |
| <u>Unidad IV: Recursividad.</u> | | 139 |
| 4.1. | Recursividad | 139 |
| <u>Unidad V: Punteros.</u> | | 145 |
| 5.1. | Concepto de puntero | 145 |
| 5.2. | Punteros y funciones | 149 |
| 5.3. | Punteros y arreglos | 150 |
| 5.4. | Arreglos de punteros | 152 |
| 5.5. | Inicialización estática de arreglos de punteros | 153 |
| 5.6. | Punteros y estructuras | 153 |
| 5.7. | Listas simplemente enlazadas | 155 |
| 5.8. | Listas doblemente enlazadas | 158 |
| Glosario | | 164 |

PRESENTACIÓN

¡ Bienvenidos a Programación I !

Séneca decía que “No hay viento favorable para el que no sabe adonde va”. La idea de esta introducción es presentarles la forma de trabajo y plantear nuestros objetivos para orientar de la mejor manera posible nuestra nave y alcanzar el destino planteado. Es muy importante por eso que estén plenamente compenetrados con ello y no pierdan de vista ese objetivo. El contenido de la materia está esencialmente desarrollado en este módulo. Nuestra clase virtual nos permitirá, alinear nuestro rumbo con el objetivo, y “acercarnos cada día un poco más a este”. Serán algunas líneas asociadas a qué debemos leer del módulo y en muchos casos algunas actividades y ejercicios adicionales que permitirán cerrar el tema. La idea es darle a la clase no sólo el contenido sino también una forma amena de comunicación. Debe ser un elemento motivador para ustedes cuando la vayan a leer y para mí al momento de escribirla.

Este curso de programación es fundamental por cuanto es el primero que ustedes tienen en su carrera, y debería dejarles alguna forma de estructurar el pensamiento para luego poder programar con cualquier lenguaje. Esto es así porque las estructuras y el modo de pensar sobre las mismas no cambian, lo que cambia es sólo la forma en que una instrucción se escribe en Pascal, en Basic, en C o en cualquier otro lenguaje. Esto es algo normalmente incomprensible para alguien ajeno al mundo de la programación, y también lo es muchas veces para personas e incluso profesionales del ambiente. Esencialmente buscamos –busco- que sepan pensar –o que aprendan a pensar- para que luego puedan llevar esa forma de trabajar a cualquier otro lenguaje.

La primera pregunta que tal vez podríamos hacernos es por qué usamos el lenguaje C para introducirnos en la programación habiendo muchísimos lenguajes que podrían servirnos para tal fin. La respuesta es que este lenguaje tiene una construcción de instrucciones muy sencilla, lo que didácticamente nos beneficia ampliamente. La forma en que las mismas se escriben es casi la misma que se utiliza para plantear el problema; esto es, la forma coloquial de la instrucción coincide casi plenamente con la forma escrita de la misma. Eso lo vamos a ir viendo a medida que utilizemos las instrucciones y será de gran utilidad para comprender su uso además de estructurar nuestro pensamiento “y ordenarlo”.

Desde el punto de vista metodológico, trabajaremos utilizando el módulo de la materia tratando de no alejarnos del eje principal que él nos provee. Haciendo una lectura del mismo podremos ir estudiando los conceptos fundamentales asociados al uso de estructuras e instrucciones además de las formas asociadas a un programa en C. A medida que los diferentes conceptos teóricos sean enunciados, ejemplos de aplicación de los mismos como así también actividades complementarias que el alumno debe ir completando conforman el material de estudio. Las respuestas a algunas actividades están incluidas en el módulo, otras en cambio no lo están y la idea es que ustedes me las envíen para poder revisarlas e ir viendo el grado de avance que van logrando a medida que va pasando el tiempo. Este tipo de interacciones son

fundamentales porque me permitirán conocer el estado de la cursada a nivel general y la de cada uno de ustedes a nivel particular. El camino que emprendemos es intenso, arduo, pero con trabajo y actitud todo es posible. Descarto que esos dos elementos estarán presentes en ustedes a lo largo de este tránsito.

¿Cuál debería ser nuestra motivación principal?

Comprender que esta materia es el punto de partida del “fascinante e infinito mundo de la programación”. Quienes encuentren en estas páginas muchas cosas que les gusten, seguramente disfrutarán de la profesión y el camino que han comenzado a transitar. La opinión de quien escribe estas líneas está cargada de subjetividad –“porque me encanta lo que hago”- pero trata de hacer saber al alumno que para llegar a la cima hay que empezar escalando por la base, y es en esta asignatura donde empieza la travesía. ¡¡Disfrutémosla!! –no la suframos-.

¿A qué área pertenece esta materia o espacio curricular?

Es parte del área de formación específica que se dedica a abordar los saberes propios de cada campo profesional –en nuestro caso la del programador y la del analista- así como también contextualizar los desarrollados en la formación de fundamento –como son la matemática y la estructura de un computador-. En esta asignatura orientamos fuertemente a la construcción de los saberes necesarios para la realización del trabajo técnico y al desarrollo de las funciones y tareas específicas del perfil del egresado. Es por eso que esta materia es importante para vuestro futuro.

¿Cuál es la utilidad de esta materia para la carrera? ¿Cuáles son las expectativas que podemos tener en el inicio?

Los temas que se desarrollarán en la asignatura son de vital importancia para la carrera. De la misma manera que para un médico el conocimiento de las distintas patologías permitirán salvarle la vida a un paciente; o para el abogado el conocimiento de las leyes es fundamental cuando debe tomar posición en un pleito; para el analista de sistemas (para el analista programador más aún) es importantísimo disponer de una estructura mental organizada en función de las formas de programar. Ese es nuestro objetivo final, que cada estructura, que cada instrucción, que cada parámetro, etc., tengan una utilización adecuada y que se vaya generando una forma de pensamiento lógico que sustente el accionar profesional. Esto se puede dar de bruces con la idea/necesidad primaria del alumno que quiere aprender “rápido” un lenguaje para programar, pero es más importante conocer las generalidades y aprender a pensar que escribir “rápidamente” un programa con un compilador y hacerlo “correr”. Esto seguramente lo harán en el futuro, pero antes tienen que saber el porqué de varias cosas para entender completamente qué es lo que están y estamos haciendo. Ustedes deben llegar a pensar y programar como profesionales. Y cuando digo que deben hacerlo así, me refiero a utilizar las mejores herramientas disponibles y las mejores características personales para que todo salga bien. Esto lo aclaro porque cuando pienso en profesionalismo, lo hago teniendo en mente la dedicación, la seriedad para encarar una tarea, la búsqueda de la eficacia y la eficiencia, la motivación y el orden con el cual se

trabaja. Si pensamos en alguien que trabaja profesionalmente, no necesariamente pensamos en alguien que tenga colgado en la pared de su oficina a un diploma –aunque pueda tenerlo- sino en alguien que trabaje con esas características que acabamos de mencionar.

¿Es necesario algún concepto o conocimiento previo para cursar la materia?

Como verán, no existe un listado o enumeración de conocimientos previos para cursar la materia, lo que sí es necesario poseer es flexibilidad mental para adaptar las formas del pensamiento lógico que necesita un programador o analista. Ya lo repetí varias veces, y me van a ver hacerlo muchas más a lo largo del curso. Lo importante es pensar. Hay ciertas estructuras que deben respetarse y con las cuales deben crearse las soluciones, de manera que es importante estar preparado para incorporar cada una de ellas y luego poder llevarlas a la práctica. El origen de las instrucciones es casi totalmente coloquial, de manera que la incorporación de las formas se hace muy naturalmente. Es evidente que en esta construcción es indispensable tener la escucha bien activa, muy abierta, estar atentos a las necesidades de nuestros interlocutores (de aquellos que plantean las necesidades y las enuncian).

¿Hay que usar algún software de C para resolver los ejercicios?

No es indispensable hacerlo, porque lo que se busca es que logren estructurar un pensamiento lógico, pero para aquellos que necesitan de un compilador para escribir sus programas y luego ejecutarlos para “verlos funcionar”, pueden bajarse alguno de los muchos que pueden encontrar libremente en la red (internet). Cualquier inquietud acerca de esto, me consultan. Trato de no imponer un compilador específico, hay muchos y muy buenos que pueden utilizarse. Trato siempre de respetar un estándar que se pueda llevar a la práctica en casi cualquier editor/compilador, por eso la indicación de una libre elección. A pesar de esto, si hay alguien que quiera usar algún compilador y no tiene idea de cuál elegir, me consultan a mi mail y ahí lo conversamos.

¿Para qué sirven los foros?

A través de ellos, normalmente se plantean situaciones problemáticas asociadas a la lógica de las instrucciones y estructuras que iremos trabajando a lo largo de la cursada. Todos podrán emitir vuestra posición/idea respecto de los planteos que serán “colgados” en los mismos. Esta dinámica es altamente provechosa en nuestro proceso de aprendizaje. Pero para que esto sea así, debemos ser ordenados y cumplir con un procedimiento que nos permita sacarle el máximo provecho a este recurso. Por eso, es indispensable que trabajemos siguiendo las recomendaciones/reglas que se detallan a continuación porque nos permite hacer el mejor uso del recurso y maximizar el provecho que podemos obtener de nuestro trabajo:

DEL USO DEL FORO

1.DEFINICIÓN

El Foro es una herramienta del campus virtual utilizada como espacio de discusión, por intermedio de mensajes, de una determinada temática. La discusión se origina mediante la presentación de un tema, pregunta o actividad

que surge de uno de los participantes, y que sirve de disparador para que los demás puedan expresar su postura acerca del tema/actividad. El debate se enriquece en la medida de que existan diferentes interlocutores que puedan expresarse. Normalmente existe un participante que toma el rol de moderador, y que es quien va conduciendo el debate. Puede haber un cierre o no dependiendo de la cuestión tratada. En caso de que el cierre exista, es usualmente llevado a cabo por el moderador.

2. TEMAS A TRATAR EN EL FORO

En el caso particular del foro del campus virtual del IRSO, cada materia tiene un foro de discusión asociado, y quien asumirá el rol de moderador será el docente de la materia a la que pertenece ese foro. Allí planteará preguntas, hará comentarios, subirá consultas, etc. que puedan generar aportes a la cursada desde diferentes puntos de vista, de modo que los cursantes puedan dar su parecer y enriquecer con sus visiones los temas tratados.

Existe también la posibilidad de que quien inicie el tema en el foro sea un alumno, porque tiene una inquietud, idea, actividad, pregunta, etc. que considera importante para lo que se está tratando en la materia por cuanto la misma puede realizar aportes que podrían ser de interés general para los cursantes de la asignatura. Entonces, deberá proceder de la siguiente forma:

- a) no podrá el alumno por sí solo iniciar una discusión en el foro;
- b) deberá comunicarle al docente mediante mail la inquietud/tema a plantear y el porqué de la misma, qué busca con ella o qué aporte puede generarle al curso;
- c) el docente deberá evaluar si la cuestión planteada puede ser de utilidad en función de los objetivos que la materia tiene y decidir si se carga o no ese disparador en el foro;
- d) en el caso de que la inquietud sea útil para los participantes de la cursada, es el docente quien sube al foro el tema planteado –haciendo explícita referencia a quien fue el generador de la cuestión tratada-;
- e) para cualquier otra consideración que tenga que ver con comunicaciones personales entre alumnos, deberá utilizarse la plataforma de e-mail a la que tienen acceso en el campus virtual, no es el foro el lugar para el intercambio de pareceres/ideas/consultas que son de interés solamente para algunos o que no tengan un trasfondo/objetivo pedagógico para la cursada.

Otro punto importante para destacar respecto de las actividades que propondré en el foro, es que cada actividad tendrá fecha de cierre, es decir que se admitirán comentarios hasta esa fecha, y luego de ello me encargaré de darle un cierre al tema con un comentario que trate de relacionar y aunar vuestros aportes. Esto es algo que creo importante por cuanto puede brindar cuestiones concretas para el aprendizaje.

La propuesta es trabajar con al menos cinco foros a lo largo de la cursada:

Presentación: En este espacio se propone a los alumnos que indiquen el porqué de su decisión para comenzar a estudiar la carrera, además de ciertas cuestiones personales básicas para que se inicie un proceso de interrelación con el docente y los demás compañeros de cursada.

Para empezar a pensar: Se trata de indagar porqué es importante pensar para programar contraponiéndolo a la actitud natural/instintiva de querer codificar.

Dispositivos de entrada/salida: ¿Por qué son importantes? Y ¿cuál es su utilidad en las aplicaciones?

Operaciones y comparaciones: ¿Por qué son necesarias y cuáles son sus equivalencias/usos?

Formas en programación: ¿Por qué hay que respetarlas?

¿Para qué sirven nuestras reuniones de chat?

Permiten una comunicación on-line para la realización de consultas asociadas con el avance de la cursada a través de una forma de respuesta instantánea. Los horarios disponibles se cargan en el campus en la primera semana de cursada, y a partir de allí ya pueden comunicarse con el docente a través de ese medio para plantearle sus dudas y para hacerle comentarios en los horarios indicados. Es un encuentro virtual pactado y que tiene una frecuencia semanal. La cartelera de chats nos permite saber siempre en qué horario estará “disponible virtualmente” el profesor para hacerle consultas. Esta herramienta es indudablemente de gran utilidad debido a que nos da la posibilidad de establecer un ida y vuelta on-line en forma instantánea que no permiten otros canales de comunicación (como el foro o el e-mail).

¿Hay alguna otra forma de comunicarse?

Sí, por medio del e-mail lombarderod@irso.edu.ar (otra opción es daniel.profeirso@gmail.com) o aguirremg@irso.edu.ar también se pueden realizar todo tipo de consultas. Este es el medio más útil para hacer/transmitir todo tipo de preguntas y comentarios personales, de manera que mi mejor consejo es que lo aprovechen. Nos conectamos todos los días al mail para leerlo y contestarlo, de manera que ténganlo en cuenta para cualquier cosa que necesiten.

¿Cuáles son los objetivos generales que se persiguen en este espacio curricular?

Es muy importante -en cualquier actividad que desarrollemos- que los objetivos que se persiguen estén definidos. No solamente eso es bueno a nivel comunicación, sino también porque nos permite entender el porqué de su existencia y cuáles son los puntos/temas de mayor trascendencia. En nuestro caso, hemos definido como objetivos los que se indican a continuación:

- Diseñar algoritmos con diferentes estructuras utilizando las instrucciones básicas de un lenguaje de programación.
- Construir los módulos necesarios en los programas utilizando subprogramas.
- Organizar y almacenar la información en archivos.
- Construir subprogramas recursivos.

- Integrar diferentes estructuras para generar aplicaciones/soluciones no sólo eficaces sino también eficientes.
- Trabajar con estructuras dinámicas y comprender las diferencias y ventajas/desventajas que las mismas tienen respecto de las estáticas.

No es un mero punteo de temas sino que representan el corazón de la materia y por ello, debemos tenerlos siempre en mente. Son la guía para realizar el recorrido de estas páginas que conforman el módulo. ¡¡No los pierdan de vista!! Además, al inicio de cada unidad tienen los objetivos correspondientes a la misma, de modo que se pueda focalizar mucho más qué es lo que se persigue específicamente en cada temática desarrollada.

¿Hay alguna bibliografía de consulta que pueda ser utilizada además de este módulo?

Sí. Los títulos de la bibliografía recomendada se encuentran indicados a continuación. Además, en el plan de trabajo pueden encontrar un detalle de qué libros son los más adecuados para cada uno de los temas/unidades tratados.

Bibliografía básica recomendada:

- Kernighan & Ritchie; (1991) El Lenguaje de programación C; México; Prentice Hall
- Brice-Arnaud Guerin; (2005) Lenguaje C++; México; Informática Técnica
- K.N.King; (2008) C Programming: A modern approach; EEUU; Norton & Co
- Ceballos; (2007) C/C++ curso de programación; España; RA-MA

¿Para qué sirven las clases y cuándo las recibiré dentro de la plataforma virtual?

La clase es una instancia sumamente importante en nuestro proceso de aprendizaje. Es uno de elementos más destacados en esta comunicación virtual que iniciamos. Cada semana (en particular los días lunes o en su defecto el primer día hábil de la semana) recibirán una clase (son treinta y dos en total a lo largo del año) que tratará el tema que corresponda en función de nuestro avance en el desarrollo de la cursada. La clase consta de ciertos comentarios sobre la temática puntual, así como algunos tips o preguntas que tratan de generar una reflexión para situarlos en el porqué de la importancia de ese tema puntual. A ese tipo de clases las llamamos teóricas.

Por otro lado, hay clases que proponen ejercicios y preguntas que tienen que ver con la puesta en práctica de esos conocimientos adquiridos. Esas clases son las que llamamos prácticas y son las que ustedes deberán completar y responder de manera que desde este lado (docente) podamos medir el avance y la comprensión sobre los temas tratados. Si bien estas clases no son obligatorias –desde el punto de vista de que no están obligados a responderlas para realizar luego las evaluaciones parciales a distancia-, representan una herramienta importante para que ustedes puedan ir monitoreando si lo que van estudiando está correctamente orientado o no en función de lo esperado.

¿Por qué hay clases optativas? ¿Para qué sirven?

Este tipo de clases son usadas para tratar algunos temas que son conexos/paralelos a lo que estudiamos en la materia, y que pueden servir de ayuda en algunos casos para contextualizar lo que hacemos en lo que tenemos como desarrollo troncal para este espacio curricular. Como su nombre lo indica no son obligatorias y el alumno puede optar por su lectura o resolución dependiendo del tipo de clase, teórica o práctica.

¿Cuáles y cuántas son las instancias evaluativas a lo largo de la cursada?

Para aprobar la cursada, el alumno deberá aprobar las tres evaluaciones parciales a distancia que se le proponen. Cada una de ellas tiene la posibilidad de una instancia de recuperación. Si el alumno no aprueba alguna de las evaluaciones parciales, deberá recurrar la materia. En el caso de no aprobar o de no entregar alguna de las evaluaciones parciales a distancia (en las dos instancias posibles: evaluación y su recuperatorio), el alumno no recibirá la siguiente/s evaluación/es a distancia y deberá recurrar la materia.

Si aprueba la cursada en 2018, el alumno tiene dos posibilidades para rendir el examen final teniendo como fecha límite para su presentación hasta las mesas de examen de Julio-Agosto de 2020.

Para poder rendir el examen final, el alumno deberá inscribirse en las mesas de examen que serán publicadas oportunamente, a través del sistema IRSO VIRTUAL.

El examen final es presencial y se realiza en el instituto.

¿Hay actividades de práctica especiales? ¿Para qué sirven?

Hay durante la cursada dos actividades especiales que hemos llamado de práctica especial/integración conceptual porque son dos ejercicios que tienen gran contacto con la tarea habitual de un analista programador/analista de sistemas. La intención de estas prácticas es resaltar ciertas cuestiones de la vida profesional para tener algún contacto con las mismas desde el aula. Tampoco son de carácter obligatorio, pero sí son muy importantes para reafirmar los conceptos que trabajamos (de manera que los invito fuertemente a que las realicen). En el plan de trabajo encontrarán las fechas de desarrollo de las mismas.

¿Cuáles son los íconos que nos ayudarán para referenciar ciertos llamados en nuestro módulo?



Actividad

Esta indicación precede a una ejercitación/tarea que debe realizar/resolver el alumno. Es importancia para poder fijar/completar el desarrollo de un tema tratado.



Atención o recuerde

El icono de "Atención o recuerde" es para aquellas definiciones o textos breves que deben recordar o son conceptos claves que se usarán a lo largo del módulo. Es fundamental prestar atención al tema que se trate.



Lea con atención

El icono de “Lea con atención” será usado cuando el texto contenga conceptos o definiciones importantes. Por eso es necesario que lea y relea esos puntos con gran detenimiento. Es mejor “perder” (léase “ganar”) un rato en la lectura que avanzar sin pensar ni fijar conceptos/definiciones importantes.



Foro

El icono de “Foro” será usado para denotar la existencia de una actividad en el foro asociado a ese tema que se está tratando. Esto permitirá la participación del alumno brindando opiniones sobre la temática abordada con un cierre por parte del docente.



Ayudante de cátedra

Nuestro colaborador aportará comentarios, tips, reflexiones y recordatorios a lo largo del módulo.

UNIDAD I: INTRODUCCIÓN A LA PROGRAMACIÓN

¿Cuáles son nuestros objetivos para esta unidad?

La idea es que esta primera parte permita al alumno interiorizarse de la forma de las instrucciones básicas recurriendo a un lenguaje coloquial que adopta la forma de un lenguaje de programación. Esto permite, a modo de transición, relacionar nuestro modo de comunicarnos cotidiano con algo que requiere de gran cantidad de formas como es la programación. Los objetivos que se persiguen en esta primera unidad son los siguientes:

- Definir correctamente qué es un algoritmo.
- Identificar las distintas instrucciones básicas que podemos utilizar y para qué sirve cada una de ellas.
- Diferenciar algunos elementos auxiliares que forman parte de cualquier lenguaje de programación: operaciones, variable, constante, estructuras anidadas, etc.
- Escribir correctamente las instrucciones básicas y lograr realizar algunas ejercitaciones usando pseudocódigo.

Antes de meternos de lleno con la temática, me gustaría comenzar este módulo con algunas reflexiones que sirvan para dar el puntapié inicial y que además sean el disparador para continuar leyendo las primeras líneas de la presente unidad.

¿Han pensado alguna vez por qué están leyendo esto?

Tal vez muchos se han hecho esta pregunta, quizás otros no. Tal vez muchos podrían contraatacar –si vale la palabra- preguntando por qué estoy yo escribiendo esto... Y mi respuesta muy seguramente, casi sin pensar y sin agregar mucho más, sería “PORQUE ME GUSTA”. Creo que ése es un motivo más que suficiente para emprender, para buscar algo nuevo, para intentarlo, y vaya que es importante... al menos es mi idea.

¿Y por qué es importante?

Porque cuando uno decide estudiar algo lo hace porque quiere –normalmente- y porque además tiene la intención de usarlo, porque quiere darle una utilidad. Algunos pensarán que las profesiones ayudan a ganar más dinero, a vivir

mejor, a servir a otros, a obtener satisfacción personal, a aumentar la autoestima y así puedo seguir enunciando un sinnúmero de posibilidades...

Hay muchos fundamentos que pueden explicar o indicar el porqué de la decisión de ponerse a estudiar. Y eso es muy propio de cada ser humano, tiene que ver con él o ella y su realidad, su contexto, aquello que lo rodea.

En ese sentido, también está bueno pensar ¿para qué voy a emprender esto?, y en particular analizar **¿por qué elegí programar?**, porque es de eso de lo que vamos a hablar durante este curso... ESA ES LA IDEA, HABLAR DE PROGRAMACIÓN, PENSAR COMO UN PROGRAMADOR, ESCRIBIR RESPETANDO LAS FORMAS COMO UN PROGRAMADOR.

Los dejo con esa pregunta/inquietud bien a flor de piel, y espero que a lo largo de estas páginas encuentren alguna satisfacción, algún guiño y tal vez alguna respuesta a tanto interrogante y a tanta necesidad. Lo que más deseo es que les guste el tránsito del camino que han elegido y que puedan disfrutarlo.

¡¡Suerte!!

Y ahora sí, empezamos con algunos conceptos que son los pilares sobre los que nos debemos apoyar para comenzar a construir este maravilloso mundo de la programación. Por eso, trataremos de comenzar definiendo qué es un algoritmo.

1.1. CONCEPTO DE ALGORITMO

Para comenzar, en esta introducción que intenta ser una primera aproximación al abordaje de la programación, podemos decir que el algoritmo trata de resolver problemas mediante programas, o quizás en una forma más sencilla, que el algoritmo como parte integrante de un programa es quien tiene la responsabilidad de generar la solución al problema planteado a través de una secuencia de instrucciones ordenadas lógicamente y que, gracias a la ejecución de las mismas en un contexto adecuado, se llega a la solución de ese problema, se obtiene una respuesta a nuestro planteo inicial. De modo que el algoritmo es el corazón de la solución.

¿Cuáles son los pasos que generalmente se siguen para realizar un algoritmo?

- A. Análisis preliminar o evaluación del problema: estudiar el problema en general y ver qué parte nos interesa, focalizar en aquello que es importante y desechar la información no útil.
- B. Definición o análisis del problema: Ver qué es lo que entra y qué es lo que sale (cuáles son los datos de partida y cuál debería ser la solución, la respuesta de nuestro cálculo, de nuestra secuencia de instrucciones), las posibles condiciones o restricciones, ...
- C. Diseño del algoritmo: Diseñar la solución propiamente dicha.
- D. El programa: Codificación del algoritmo en un lenguaje de programación. Se le da forma para que sea entendible por quien debe procesarlo.
- E. Ejecución del programa y las pruebas: Ver si el programa hace lo que queríamos, ¿resuelve correctamente el problema planteado? Si tiene errores, entonces deberíamos modificar el algoritmo/secuencia para que sirva.

En resumen, y para tener una idea que nos sirva para trabajar con pseudocódigo –que ampliaremos más adelante en el módulo-, ¿qué es un algoritmo?: Es una “fórmula” para resolver un problema. Es una “forma” de resolver un problema.

Es un conjunto de acciones o secuencia de operaciones que ejecutadas en un determinado orden resuelven el problema.

Existen muchos algoritmos para resolver un problema, normalmente hay que elegir uno, muchas veces es el más efectivo, otras veces es el que más nos gusta o el que consideramos más sencillo según nuestra visión.

Características de un algoritmo:

- Tiene que ser preciso. Responder claramente a la necesidad. No puede ser vago.
- Tiene que estar bien definido. Debe haber una secuencia estipulada.
- Tiene que ser finito. Debe tener un comienzo y un final.

La programación es adaptar el algoritmo a la computadora, al lenguaje de programación utilizado.

El algoritmo es independiente según donde lo implemente. Esto es algo fundamental, no se ve influido por el lenguaje de programación en el que será escrito/implementado. Podríamos decir que es la solución al problema, y que

como tal, debería poderse escribir de alguna manera en cualquiera de los lenguajes de programación existentes (o al menos en algunos de ellos). Por ejemplo, un algoritmo que muestre la forma en la que se hace una llamada telefónica tendría una secuencia de acciones como se ve a continuación:

INICIO

- Levante el tubo del teléfono
- Espere tono
- Marque el número del destinatario
- Espere que contesten
- Hable con el destinatario de la llamada
- Cuelgue

FIN

Otro ejemplo de algoritmo, podría ser la secuencia que se da cada vez que viajamos en colectivo desde un punto a otro.

INICIO

- Vaya a la parada del colectivo
- Espere que venga el colectivo
- Levante la mano indicando que pare
- Suba
- Indique el destino al chofer
- Acerque la tarjeta para el pago del boleto a la máquina
- Permanezca en el colectivo hasta cumplir el trayecto
- Cuando se acerque al destino presione el timbre
- Cuando pare, descienda

FIN



Secuencia: Continuidad, sucesión ordenada. Serie o sucesión de cosas que guardan entre sí cierta relación. En matemática, conjunto de cantidades u operaciones ordenadas de tal modo que cada una está determinada por las anteriores.

Secuenciar: Establecer una serie o sucesión de cosas que guardan entre sí cierta relación.

1.2. EL LENGUAJE DE PROGRAMACIÓN

Los algoritmos se escriben o implementan en una computadora para lograr luego ejecutarlos y obtener a partir de ellos el resultado/solución buscado. Para eso, existen diferentes lenguajes de programación que permiten, en diferentes formas escritas, su implementación. Vamos a detallar a continuación algunas de ellas:

Tipos de lenguajes:

-Lenguaje binario (otros lo llaman lenguaje natural, también a veces se usa el término lenguaje máquina para englobar el binario o natural con el ensamblador): Todo se programa con 1 y 0 que es lo único que entiende el procesador. Es sencillo porque solamente tiene dos valores, pero es complicado porque cada instrucción necesita para poder identificarse de varios 0 y 1.

Ventaja: No necesita ser traducido porque las máquinas entienden esos valores en forma natural.

Inconveniente: La dificultad y la confusión para corregir errores, es propia de cada máquina (de cada procesador). Hay una dependencia de muchos tipos diferentes de máquinas.

Ejemplo de codificación en lenguaje binario:

[illegible]

-De bajo nivel o ensamblador: Se utilizan mnemotécnicos (abreviaturas que representan las diferentes instrucciones –combinan los 0 y 1 para darle forma a diferentes acciones-).

Ventaja: No es tan difícil como el lenguaje máquina. Es mucho más sencillo escribir instrucciones complejas.

Inconvenientes: Cada máquina tiene su propio lenguaje (depende de cada procesador), necesitamos un proceso de traducción.

Ejemplo de lenguaje ensamblador:

```

Str:    DB "Testing...", 0
        mov eax, -2
        cmp eax, 2
        jle Label
        xor eax, eax
Label:  lea esi, Str
        push esi
        call "Response.write(string)"
        pop esi
    
```

-Lenguajes de alto nivel: son los más cercanos al lenguaje humano. Utilizan mucho el concepto de las formas coloquiales y lógicas del ser humano, es decir que la forma de pensarlos y escribirlos tiene un correlato muy directo con eso.

Ventaja: Son independientes de cada máquina (los compiladores aceptan las instrucciones estándar, pero también tienen instrucciones propias, eso les da gran flexibilidad).

Inconveniente: El proceso de traducción es muy largo y ocupa más recursos.

Aprovecha menos los recursos internos. ¡¡¡Pero es estándar!!!

Ejemplo de lenguaje de alto nivel:

```

program programa;
var num1, num2 : integer;
begin
    write('Ingresa un numero ');
    readln(num1);
    write('Ingresa otro numero ');
    readln(num2);

    if num1 > num2 then
        write(num1, ' es mayor a ', num2);
    else
        write(num1, ' es menor a ', num2);
    end.
end.
    
```



Lenguaje: Conjunto de sonidos articulados con que el hombre manifiesta lo que piensa o siente. Estilo y modo de hablar y escribir de cada persona en particular. Conjunto de signos y reglas que permite la comunicación con una computadora.

Lenguaje de alto nivel: Lenguaje que facilita la comunicación con una computadora mediante signos convencionales cercanos a los de un lenguaje natural. Acá tenemos una aproximación muy directa a lo que en el módulo llamamos lenguaje coloquial.

Instrucciones en una computadora y sus tipos:

Una instrucción es cada paso de un algoritmo ejecutado por el procesador.

Tipos de instrucciones:

- Entrada/Salida (E/S): Pasan información del exterior al interior del procesador y al revés. Son los datos que requiere el programa para ejecutarse, y los que devuelve como solución, como resultado del procesamiento.

Por ejemplo, una impresión en pantalla de un dato para que pueda ser visualizado por el usuario de la computadora. Otro caso, podría ser la que permite leer valores que el usuario ingresa por teclado para que luego puedan ser usados por el programa para hacer algo, por ejemplo, una cantidad, una fecha, un nombre, etc.

- Aritmético-lógicas: Aritméticas: +, -, *, ... ; Lógicas: or, and, <, >, ... Sirven para hacer operaciones y para manejar la interrelación entre variables, especialmente las comparaciones.

Por ejemplo, son instrucciones para hacer sumas, restas, multiplicaciones, divisiones, comparaciones de mayor, menor, igual, etc.

- Selectivas: Permiten la selección de una alternativa en función de una condición. Son llamadas también estructuras de decisión, porque a partir de opciones se consiguen tomar diferentes caminos.

Por ejemplo, si se ingresa el valor 4 entonces hacer determinadas acciones, sino hacer otras acciones. En este caso hay una dependencia del valor ingresado para hacer unas u otras acciones.

- Repetitivas: Repetición de un número de instrucciones, un número finito de veces. Se ejecuta varias veces la misma acción.

Por ejemplo, si se quieren imprimir varias líneas en blanco en la pantalla se puede recurrir a estas acciones de repetición. De esa manera repetiré la instrucción de escribir una línea en blanco tantas veces como líneas quiero que estén en blanco en la pantalla.



¿Para qué sirven las instrucciones?

Son el elemento primario con el que escribiremos los algoritmos y con estos luego los programas. Son la base de nuestro lenguaje.

Y... ¿qué es un programa?

Un programa, como primera aproximación, es un conjunto de instrucciones que ejecutadas ordenadamente resuelven un problema. Esto siempre pensándolo en el contexto de un lenguaje de programación. Un programa no existe si no hay un lenguaje de programación que lo contenga. Y un programa siempre va asociado a un algoritmo. Porque el algoritmo es esa secuencia de instrucciones que nos permiten hacer algo. De alguna manera, y muy burdamente, esta es una aproximación a aquello que queremos trabajar.

Pero, ¿cómo es que la máquina entiende lo que está escrito en el programa?


Proceso de traducción y ejecución de un programa escrito en un lenguaje de alto nivel:

Usamos un editor y escribimos el programa fuente, el compilador es el que traduce el programa al lenguaje máquina (o binario). El compilador internamente ha sido diseñado para traducir esas palabras que simbolizan una acción o una instrucción en códigos de 0 y 1.

El compilador obtiene el programa llamado objeto (que no es ni más ni menos que una secuencia de 0 y 1 que representan cada una de las instrucciones, datos y variables que maneja la computadora para obtener la solución al problema planteado). El compilador tiene –además– que buscar los errores. Normalmente no se obtiene directamente un ejecutable, sino que se necesitan otros elementos, librerías, datos adicionales, archivos, etc.

Mediante un linkador se unen el programa objeto y las librerías, y se forma un programa ejecutable.

¿Qué pasa si el programa tiene errores? ¿Cómo es posible ver dónde puede haber un error?

Para ello se utiliza un debugger que es un software que depura el programa ejecutándolo paso a paso, viendo la memoria paso a paso en busca de posibles errores.  “Para empezar a pensar”

1.3. DATOS, TIPOS DE DATOS Y OPERACIONES

- Dato: Es un objeto o elemento que se toma para realizar diversas operaciones. Generalmente son simplemente valores.

Tienen 3 características:

- Un nombre que los diferencia del resto.

- Un tipo que nos determina las operaciones que podemos hacer con ese dato (normalmente no podemos hacer las mismas operaciones con números que con letras).
- Tienen un valor que puede variar o no a lo largo de la operación.

Existen diferentes tipos de datos que tienen, según sea, las siguientes características:

- Cada tipo se representa o almacena de forma diferente en la computadora. No es lo mismo almacenar un bit, que puede tener valor 0 ó 1, que almacenar un byte que puede tomar valores de 0 a 255 (o sea 8 bits).
- Un tipo agrupa a los valores que hacen las mismas operaciones.
- Si tiene definida una relación de orden (es decir, que todo elemento salvo el primero y el último tienen un predecesor y un sucesor, como puede ocurrir en los números naturales) es un tipo escalar.
- Cardinalidad de un tipo: Número de valores distintos que puede tomar un tipo (o cantidad de diferentes valores que puede tomar una variable de ese tipo). Pueden ser finitos (caracteres), y si son infinitos el ordenador los toma como finitos porque está limitado por el tamaño de los bytes en el que la cifra es almacenada. Hay una limitación física que está dada por la cantidad de bits que cada procesador puede manejar simultáneamente, de modo que los tipos de datos (por más que sean por ejemplo, números reales –que en matemática tienen un número infinito de elementos, porque siempre entre dos números reales hay otro número real-) siempre son finitos.

Los datos pueden ser:

- Simples: Un elemento.
- Compuestos: Varios elementos relacionados.

Los tipos pueden ser:

- Estándar: Que vienen en el sistema por defecto, que son propios del lenguaje desde su nacimiento, de su misma concepción.
- No estándar: Son los que crea el usuario, son aquellos que genera el programador porque le permiten solucionar necesidades particulares, propias de su aplicación.

Los tipos estándar más importantes son:

- Numéricos:

- Entero: Subconjunto finito del conjunto matemático de los números enteros.

No tiene parte decimal. El rango de los valores depende del tamaño que se les da en memoria.

- Real: Subconjunto finito del conjunto matemático de los números reales.

Llevan signo y parte decimal. Se almacenan habitualmente en 4 Bytes (dependiendo de los modificadores y del lenguaje). Si se utilizan números reales muy grandes, se puede usar notación científica que se divide en mantisa, base y exponente (esto permite ampliar el rango de utilización de los números reales); tal que el valor se obtiene multiplicando la mantisa por la base elevada al exponente (es algo que el lenguaje de programación interpreta por sí solo, es una notación que el mismo utiliza).

- Lógicos o booleanos:

- Son aquellos tipos en los que las variables sólo pueden tomar dos valores, llamados normalmente valores de verdad, verdadero o falso (1/0).

- Caracter:

- Abarca al conjunto finito y ordenado de caracteres que reconoce la computadora (letras, dígitos, caracteres especiales, ASCII).

Tipo de cadena o string: Conjunto de caracteres reunidos bajo un mismo nombre, conforman un dato, una variable, se indica su extensión con comillas, o sea que los caracteres van a estar encerrados entre “”.

Cada lenguaje completa o modifica estos tipos dependiendo de su uso u orientación. Si el lenguaje no trae ciertos tipos que puedan ser de utilidad para el programador, será este quien los defina o cree a partir de los tipos estándar o de partida.



El tipo de datos es el conjunto de valores que puede tomar una variable durante la ejecución del programa.

Tiene básicamente tres objetivos principales:
Detectar errores en las operaciones.
Determinar cómo ejecutar estas operaciones.
Eficiencia en el uso de memoria.

1.4. CONSTANTES Y VARIABLES

- Constantes: Tienen un valor fijo que se le da cuando se la define y que ya no puede ser modificado durante la ejecución.
- Variables: El valor puede cambiar durante la ejecución del programa, pero nunca varía su nombre y su tipo.

Antes de usar una variable hay que definirla o declararla, al hacerlo hay que dar su nombre y su tipo. El nombre que le damos tiene que ser un nombre significativo, va a ser un conjunto de caracteres que dependiendo del lenguaje puede tener restricciones. Normalmente debe empezar con una letra, y el tamaño depende del lenguaje. Ese nombre o identificador debe ser una palabra que no sea propia del lenguaje, una palabra que no sea de las que se consideran reservadas al lenguaje porque significan algo en ese lenguaje, porque ya existen. Si a la variable, al declararla, no se la inicializa, en algunos lenguajes toma un valor inicial por defecto, es decir que se carga con algún valor no representativo, por ejemplo, en muchos lenguajes, cuando se crea una variable entera automáticamente toma el valor cero (0). En cualquier caso, en cualquier lenguaje, las variables pueden ser cargadas con un valor inicial y luego pueden cambiarse a lo largo de la ejecución.

-Constantes: Pueden llevar asociado un nombre o no, si no lo llevan, se llaman literales. Su valor hay que darlo al definir la constante y ya no puede cambiar a lo largo de la ejecución. En lo relacionado con el tipo, hay lenguajes en los que hay que definir las constantes con un tipo, y en otros, toma por defecto el tipo del valor con el cual se carga. La ventaja de usar constantes con nombre es que en cualquier lugar del programa donde se quiera que vaya la constante, basta con poner su nombre y luego el compilador lo sustituirá por su valor.

Además, si por alguna cuestión es necesario modificar para todo el programa el valor de la constante, con solamente modificarlo en el lugar donde fue creada alcanza, y no es necesario hacer las modificaciones del valor en todo el programa. Las constantes sin nombres son de valor constante: 5, 6, 'a', "hola".

Relación entre variables y constantes en memoria:

Al detectar una variable o una constante con nombre, automáticamente se reserva en memoria espacio para guardar esa variable o constante. El espacio reservado depende del tipo de la variable.

En esa zona de memoria es en la que se guarda el valor asociado a la variable o constante y cuando el programa use esa variable, irá a ese lugar a buscar su valor.

1.5. EXPRESIONES - TIPOS Y OPERADORES

Una expresión es una combinación de constantes, variables, signos de operación, paréntesis y nombres especiales (nombres de funciones estándar), con un sentido unívoco y definido y de cuya evaluación resulta un único valor. Toda expresión tiene asociado un tipo que se corresponde con el tipo del valor que devuelve la expresión cuando se evalúa, por lo que habrá tantos tipos de expresiones como tipos de datos. Habrá expresiones numéricas y lógicas.

Numéricas: Operadores aritméticos. Son los que se utilizan en las expresiones con números (una combinación de variables y/o constantes numéricas con operadores aritméticos y que al evaluarla devuelve otro valor numérico +, -, *, /).

Operación resto: Lo que devuelve es el resto de una división entera.

División entera: Nos devuelve el cociente de una división entera (en la que no se sacan decimales).

Potencia: a^b devuelve a elevado a la b.

Todos estos operadores son binarios (el operador se sitúa en medio), el menos también puede ser unario (lleva un único operando) y significa signo negativo.

Reglas de precedencia: El problema es que cuando se toma una expresión, según como la evalúe, puede dar como resultado diferentes valores.

La solución es aplicar prioridad entre los operadores, de modo que ante la posibilidad de usar varios, siempre aplicaremos primero el de mayor prioridad. Cada lenguaje puede establecer sus propias reglas de prioridad o precedencia de operadores. Si no nos acordamos o no las conocemos, siempre se pueden usar () para indicar qué es lo que se debe calcular primero. Normalmente, en los lenguajes, la prioridad está dada por:

1ª) ^

2ª) * / div mod

3ª) + -

Entre dos operaciones que tienen la misma precedencia para resolver la ambigüedad, hay que usar la regla de la asociatividad. La más normal es la de la asociatividad a izquierda (primero se resuelve lo de la izquierda, o sea que la resolución es de izquierda a derecha).

Expresiones lógicas: Operadores relacionales y lógicos. Una expresión lógica es aquella que sólo puede devolver dos valores (Verdadero o Falso). Los valores que pueden aparecer en una expresión lógica son de dos tipos: lógicos y relacionales.

La particularidad de las expresiones lógicas es que mientras en una expresión numérica que va a devolver un valor numérico, los operandos solo pueden ser números. En una expresión lógica los operandos no tienen porqué ser booleanos aunque se devuelva un valor booleano. Esto es lo que ocurre cuando en la expresión lógica utilizamos operadores relacionales con lo cual se obtienen valores lógicos o booleanos a partir de otros que no lo son. En cambio cuando los operadores son lógicos, los operandos obligatoriamente también tienen que ser lógicos.

Operadores relacionales:

<

>

=

<>

>=

<=

< Operando1> operador < Operando2>

5 > 3 Verdadero

¿Cómo se evalúa una expresión relacional?:

- Primero se evalúa el primer operando y se sustituye por su valor.
- Luego se evalúa el segundo operando y se sustituye por su valor.

- Finalmente se aplica el operador relacional y se devuelve el valor booleano correspondiente.

$$((3*2)+1-5^2) < (2-1)$$

-18 < 1 Verdadero

El problema del tipo real:

Desde la informática, los números reales son finitos, ya que tenemos un máximo de cifras decimales. Cuando trabajamos con un =, matemáticamente da un valor pero “informáticamente” puede no dar lo mismo (básicamente por una cuestión de redondeo/truncado del número considerado).

Soluciones:

-La que nos aporte el propio lenguaje de programación. Se considera un valor de error.

-Trabajar con números enteros siempre que sea posible.

-Utilizar las comparaciones por <> en vez de =, >= ó <= si es posible.

-Si hay que preguntar por igual, cambiar la condición utilizando valores absolutos y un valor mínimo de error (es una opción).

Por ejemplo, si la diferencia < 0.00000001 y el valor absoluto A-B < min entonces son iguales.

Operadores lógicos:

A veces queremos preguntar o evaluar por más de una condición al mismo tiempo y para esto se usan los operadores lógicos: Y (and), O (or), NO (not).

Cada uno de estos operadores lógicos tiene una tabla de verdad que relaciona los posibles valores a tomar con el resultado de la evaluación:

Y, O son operadores binarios (necesitan 2 operandos de tipo lógico):

| Operando 1 | Operando 2 | AND | OR |
|------------|------------|-----|----|
| V | V | V | V |
| V | F | F | V |
| F | V | F | V |
| F | F | F | F |

El No es unario y se coloca primero el NO, y después el operando.

| Valor | NO | Valor |
|-------|----|-------|
| V | F | |
| F | V | |

Prioridades de los operadores:

-El de mayor prioridad es el NO

-Luego el Y y el O

-<, > =,...

La tabla completa de prioridades quedaría conformada así:

^ NO

/ div mod Y

+ - O

<, >, =, <>,...

1.6. OPERACIÓN DE ASIGNACIÓN

Consiste en atribuir un valor a una variable. El valor será una expresión (constante, variable,...).

Normalmente se usa el signo igual para tal operación.

Variable a la que se le asigna el valor = al valor que le vamos a asignar

A = 5

El proceso de asignación se realiza en dos fases:

-Se evalúa la expresión de la parte derecha de la asignación obteniéndose un único valor.

-Se asigna ese valor a la variable de la parte izquierda.

¿Qué es lo que hay que tener en cuenta?:

-En la parte izquierda sólo puede haber una variable.

-La variable a la que se le asigna el valor pierde su valor anterior.

-EL TIPO DEL VALOR QUE SE OBTIENE AL EVALUAR LA PARTE DERECHA TIENE QUE SER EL MISMO QUE EL TIPO DE LA VARIABLE DE LA PARTE IZQUIERDA, ES DECIR A UNA VARIABLE SOLO SE LE PUEDEN DAR VALORES DE SU MISMO TIPO.



“Operaciones y comparaciones”

1.7. ENTRADA Y SALIDA DE DATOS

Las dos operaciones básicas de cada proceso con salida de datos son las de lectura y escritura. La lectura es equivalente a la asignación en cuanto que va a

haber una variable que recibe un valor, pero este valor no resulta de evaluar ninguna expresión sino que el valor lo vamos a leer de un dispositivo externo de entrada (por ejemplo, se lee del teclado, el operador/usuario ingresa el valor por teclado):

Leer (nombre de la variable)

El valor introducido por el dispositivo externo, tiene que ser del mismo tipo que el de la variable que se usa para asignar.



“Dispositivos de entrada y salida”

La operación de escritura lo que hace es mostrar el valor de una variable en un dispositivo externo de salida.

Escribir (variable)

La operación de escritura no es una operación destructiva de memoria. No toca ni modifica el dato de la memoria, solamente lo muestra en la pantalla.

1.8. ESTRUCTURAS DE SELECCIÓN

Se evalúa la condición y en función del resultado se ejecuta un conjunto de instrucciones u otro. Hay tres tipos de selectivas (simple, doble o múltiple):

* Simple: Evaluamos la condición y si es verdadera ejecutamos el conjunto de condiciones asociadas al entonces, y si es falso, no hacemos nada.

Si <condición>

entonces <acciones>

fin si

* Doble: Se evalúa la condición y si es verdad se ejecutan el conjunto de acciones asociadas a la parte entonces, y si es falso se ejecutan el conjunto de acciones asociadas a la parte sino.

Si <condición>

Entonces <acciones1>

Sino <acciones2>

Fin si

* Alternativa múltiple: Se evalúa una condición o expresión que puede tomar n valores. Según el valor que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor.

Según sea <expresión-variable>

<Valor1>: <acción1>

<valor2>: <acción2>

.....

[<otro>: <acciones>]

fin según

Las acciones asociadas al valor otro se ejecutan cuando la expresión no toma ninguno de los valores que aparecen antes.

El valor con el que se compara la expresión va a depender de los lenguajes, de lo que sea ese valor. En general ese valor puede ser un valor constante, un rango de valores o incluso otra condición.

Ejemplo: Para un rango de valores, leer una nota y escribir en pantalla la calificación.

Var nota: entero

Leer nota

Según sea nota

1..4: escribir "En suspenso, a reevaluar"

5..6: escribir "Aprobado"

7..8: escribir "Notable"

9: escribir "Superior"

10: escribir "Sobresaliente"

Otro: escribir "No es una nota válida"

fin según

1.9. ESTRUCTURAS DE REPETICIÓN

Son aquellas que contienen un bucle, un lazo (un conjunto de instrucciones que se repiten un número finito de veces). Cada repetición del bucle se llama

iteración. Todo bucle tiene que llevar asociada una condición, que es la que va a determinar hasta cuando se repite el bucle. Es la condición de corte.

Podemos identificar básicamente tres tipos de lazos:

1. Mientras / hacer
2. Repetir / hasta
3. Desde hasta

MIENTRAS / HACER

Mientras <condición> hacer
 <acciones>
fin mientras

En este caso, la condición del bucle se evalúa al principio, antes de entrar en él. Si la condición es verdadera, comenzamos a ejecutar las acciones del bucle y después de la última volvemos a preguntar por la condición. En el momento en el que la condición sea falsa salimos del bucle y ejecutamos la siguiente instrucción luego del bucle.

Al evaluarse la condición antes de entrar en el bucle, si la condición la primera vez es falsa, no entraremos nunca en el bucle. Puede suceder que nunca se ejecuten las acciones del bucle, por lo tanto usaremos obligatoriamente este tipo de bucle en el caso de que exista la posibilidad de que las acciones del bucle puedan ejecutarse 0 veces.

REPETIR / HASTA

Repetir
 <acciones>
hasta <condición>

En este caso, se repite el bucle hasta que la condición sea verdadera. Es decir, se repite mientras la condición sea falsa. La condición se evalúa siempre al final del bucle, si es falsa volvemos a ejecutar las acciones, si es verdadera se sale del bucle y se ejecuta la próxima instrucción después del bucle.

Como la condición se evalúa al final, incluso aunque la primera vez ya sea verdadera, habremos pasado al menos una vez por el bucle. Es decir que las acciones se ejecutarán al menos una vez.

Cuando un bucle se tenga que ejecutar como mínimo una vez, podremos usar una estructura repetir o mientras, la única diferencia que habrá entre las dos, es que para hacer lo mismo, las condiciones tienen que ser contrarias. Eso puede verse en el siguiente ejemplo, donde se desea leer tres números de teclado y realizar la suma de los mismos.

```
cont <- 0
suma <- 0
mientras cont <> 3
    suma <- suma + num
    leer num
    cont <- cont + 1
fin mientras
```

```
cont <- 0
suma <- 0
repetir
    leer num
    suma <- suma + num
    cont <- cont + 1
hasta cont = 3
```

DESDE HASTA

Este tipo de bucles se utiliza cuando se sabe de antemano el número exacto de veces que hay que ejecutarlo. Para ello se usa una variable adicional, a la que llamamos variable índice, a la que se le asigna un valor inicial y uno final. Además se va a incrementar o decrementar en cada iteración de bucle en un valor constante, pero esto se va a hacer de manera automática. El programador no se tiene que ocupar de incrementar o decrementar esta variable en cada iteración, sino que va a ser una operación implícita a la instrucción (lo hace por defecto).

Por lo tanto, en cada iteración del bucle la variable índice se actualiza automáticamente y cuando alcanza el valor que hemos puesto como final se termina la ejecución del mismo.

```
Desde <var índice>=<valor inicial> hasta <valor final>
    <acciones>
fin desde
```

1.10. ESTRUCTURAS ANIDADAS

Tanto las estructuras selectivas como los bucles se pueden anidar unos dentro de otros.

Anidación de condicionales:

```
Si <condicion1>
    Entonces <sentencia1>
```

Sino si <condicion2>

Entonces <sentencia2>

Sino si <condicion2>

Entonces <sentencia3>

Fin si

Fin si

Fin si

Bucles anidados: Al anidar bucles hay que tener en cuenta que el bucle más interno funciona como una sentencia del bloque más externo y por lo tanto, en cada iteración del bucle más externo se van a ejecutar todas las iteraciones del bucle más interno. Si el bucle más externo se repite n veces y el más interno se repite m veces, si por cada iteración del más externo se repite el más interno, entonces, el número total de iteraciones será $m \cdot n$. Los bucles que se anidan pueden ser de igual o distinto tipo.

Desde $i=1$ hasta 8

Desde $j=1$ hasta 5

Escribir "Profesor" i "¡qué buena su asignatura!" "Esto lo escribió el alumno" j

Fin desde

Fin desde

Dados todos estos lineamientos y conceptos de partida, es posible volver a la enunciación de qué son un algoritmo y un programa para ir aproximándonos a ejemplos de programación utilizando lenguaje pseudocódigo y algún que otro diagrama de flujo.

1.11. RESOLUCIÓN DEL PROBLEMA

La resolución de un problema, entonces, considerando el algoritmo, puede pensarse como:

- Análisis del problema: Comprensión.
- Diseño del algoritmo: Resolución usando los criterios que correspondan.
- Edición en la computadora: Creación del algoritmo en un lenguaje de programación.

ANÁLISIS DEL PROBLEMA

El objetivo de esta fase es comprender el problema para lo cual como resultado tenemos que obtener la especificación de las entradas y salidas, cuáles son los datos relevantes con los que contamos y qué debemos devolver luego de procesarlos, qué se espera que devuelva el programa. En definitiva: ¿Qué es lo que debe hacer el programa?

DISEÑO DEL ALGORITMO

Una vez comprendido el problema se trata de determinar qué pasos o acciones tenemos que realizar para resolverlo.

Como criterios a seguir a la hora de dar la solución algorítmica hay que tener en cuenta:

-Si el problema es bastante complicado, lo mejor es dividirlo en partes más pequeñas e intentar resolverlas por separado. Esta metodología de “divide y vencerás” también se conoce con el nombre de diseño descendente. Es algo muy apropiado también cuando se desea hacer modular a un programa, la solución general al problema puede ser suma de soluciones más simples. Las ventajas de aplicar esto son: Al dividir el problema en módulos o partes se comprende más fácilmente; al hacer modificaciones es más fácil sobre un módulo en particular que en todo el algoritmo; los resultados, se probarán mucho mejor comprobando si cada módulo da el resultado correcto que si intentamos probar de una sola corrida todo el programa porque si se produce un error sabemos en qué módulo se encuentra la falla.

Una segunda forma es utilizar un proceso de refinamiento por pasos, a partir de una idea general se va aguzando cada vez más esa idea original de modo que se aproxime a algo concreto para resolver. Se pasa de lo más complejo a lo más simple.

La representación de los algoritmos:

Una vez que tenemos la solución hay que implementarla con alguna representación. Las representaciones más usadas son los diagramas de flujo (también conocidos como flujogramas) y el pseudocódigo.

Al escribir el algoritmo hay que tener en cuenta:

-Las acciones o pasos a realizar tienen que tener un determinado orden.

- En cada momento solo se puede ejecutar una acción.
- Dentro de las sentencias del algoritmo pueden existir palabras reservadas (palabras propias del lenguaje de programación que tienen para el compilador un determinado significado).
- Si estamos utilizando pseudocódigo tenemos también que usar la escritura bien tabulada, porque eso es muestra clara de cuáles son las instrucciones que contienen a otras instrucciones y delimitan muy claramente ante la primera mirada qué código es contenido por o contiene a otro (aumenta la legibilidad del problema).

EDICIÓN EN LA COMPUTADORA

Es hacer entender nuestro algoritmo a la computadora para que lo pueda ejecutar.

1. Codificamos el algoritmo en un lenguaje de programación.
2. Ejecutamos el programa antes compilado.
3. Comprobamos los resultados, y si no funciona lo corregimos.

1.12. DIAGRAMAS DE FLUJO

Representan gráficamente paso a paso todas las instrucciones del programa a codificar reflejando la secuencia lógica de las operaciones necesarias para la resolución del problema. Muestra gráficamente el algoritmo del programa. Es una herramienta visual y lógica muy sencilla.

1. Debe tener un inicio y un fin (programa propio).
2. Deben usarse líneas rectas (no curvas) para indicar y marcar la secuencia.
3. Debe diseñarse de arriba hacia abajo y de izquierda a derecha (mayor legibilidad y comprensión).
4. Debe guardarse la mayor simetría gráfica posible, eso da un aspecto de centrado de la secuencia, mayor facilidad para la lectura.
5. Deben usarse expresiones independientes de los lenguajes de programación (formato general, es algo que implica una forma y una lógica universales, luego puede adaptarse a cada lenguaje de programación en particular).
6. Debe utilizarse el mínimo número de instrucciones posible (simplicidad del programa/algoritmo).

Entonces:

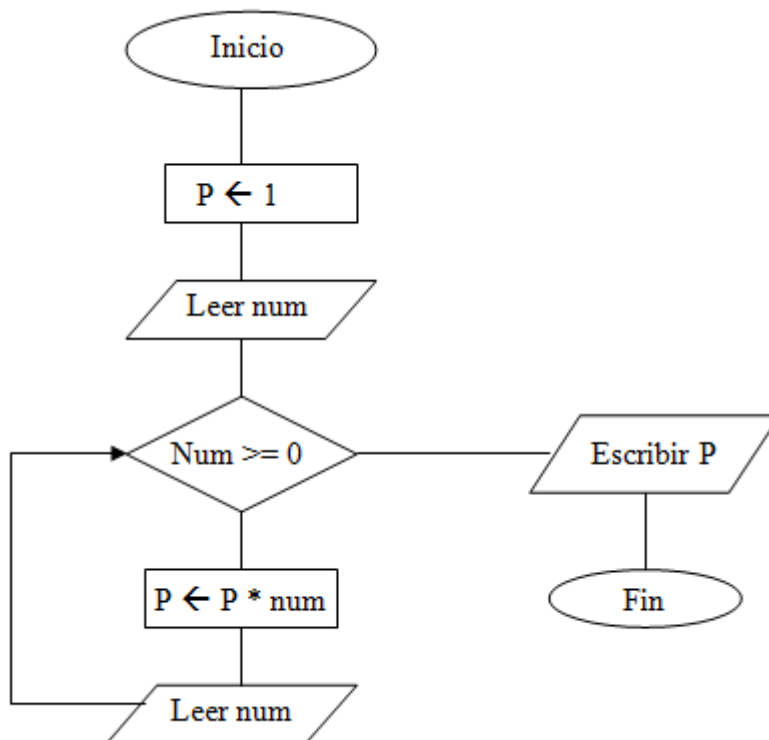
Las cajas están conectadas entre sí por líneas y eso nos indica el orden en el que tenemos que ejecutar las acciones. En todo algoritmo siempre habrá una caja de inicio y otra de fin, para el principio y final del algoritmo. Las líneas de flujo son líneas con una flecha que sirven para conectar los símbolos del diagrama. La flecha indica la secuencia en la que se van a ejecutar las acciones. El símbolo de proceso indica la acción que tiene que realizar la computadora (dentro escribimos qué es lo que debe hacer).

Ejemplo: Realizar un programa que permita multiplicar un conjunto de números positivos que se ingresan por teclado (la lectura de los números finaliza cuando se ingresa uno negativo).

Los pasos para resolver el algoritmo son los siguientes:

1. Inicializar la variable en la que se va a acumular el producto (resultado de las operaciones de multiplicar).
2. Leer de teclado el primer número.
3. Preguntar si es negativo o positivo.
4. Si es negativo termina la secuencia de ingreso de datos desde teclado y se imprime el resultado del producto.
5. Si es positivo, multiplicamos por el número ingresado y luego leemos un nuevo número, y se vuelve al paso 3, hasta que se ingrese un número negativo.

En términos gráficos, la representación es:



1.13. SEUDOCÓDIGO

Es una notación mediante la cual podemos escribir la solución a un problema dado en forma de algoritmo (pasos ordenados) utilizando palabras y frases del lenguaje natural (coloquial) aunque sujetas a determinadas reglas impuestas. Es un lenguaje de especificación de algoritmos, pero muy parecido a cualquier lenguaje de programación, por lo que luego su traducción al lenguaje es muy sencilla, pero con la ventaja de que no se rige por las normas de un lenguaje en particular. Nos centramos más en la lógica del problema.

El pseudocódigo también va a utilizar una serie de palabras clave o palabras especiales que va indicando lo que significa el algoritmo.

1. Inicio y Fin: Dónde empieza y dónde termina el algoritmo.
2. Si <condición>
Entonces <acciones1>
Sino <acciones2>
3. Mientras <condición> /hacer <acciones>
4. Repetir <acciones> / hasta
5. Desde / hasta / hacer <acciones>

6. Según sea <variable>

<Valor1 de variable>: Hacer <acciones1>

<Valor2 de variable>: Hacer <acciones2>

<Valor3 de variable>: Hacer <acciones3>

...

Algoritmo <nombre algoritmo>

Variables

<nombre>: <tipo>

Inicio

<Instrucciones>

Fin

Ejemplo 1: Realizar un programa que permita multiplicar un conjunto de números positivos que se ingresan por teclado (la lectura de los números finaliza cuando se ingresa uno negativo). Es el mismo ejemplo desarrollado para diagramas de flujo, pero ahora en pseudocódigo.

```

Algoritmo Producto
Var
  P, num: entero
Inicio
  P ← 1
  Leer num
  Mientras num >=0 hacer
    P ← p*num
    Leer num
  Fin mientras
  Escribir p
Fin
    
```

Ejemplo 2: Generar un programa que permita ingresar tres valores numéricos desde teclado y que haga la suma de los mismos. Luego imprimir el resultado en pantalla.

```

Algoritmo ejemplo
Var cont, num, sum: entero
Inicio
    Cont ← 0
    Sum ← 0
    Mientras cont <= 3
        Leer num
        Sum ← sum + num
        Cont ← cont + 1
    Fin mientras
    Escribir suma
End
    
```

Ejemplo 3: Repetir el ejercicio del ejemplo 2, pero ahora también indicar si entre esos tres números leídos hay alguno que sea negativo.

```

Algoritmo ejemplo
Var cont, num, suma: entero
    Neg: boolean
Inicio
    Cont ← 0
    Sum ← 0
    Neg ← falso
    Mientras cont <= 3
        Leer num
        Si num < 0
            Entonces neg ← verdadero
        Fin si
        Sum ← sum + num
        Cont ← cont + 1
    Fin mientras
    Si neg=verdadero
        Entonces escribir ("Se ha leído negativos")
        Sino escribir ("No negativos")
    Fin si
Fin
    
```

UNIDAD II: ALGORITMOS Y PROGRAMACIÓN

ESTRUCTURADA

¿Cuáles son nuestros objetivos para esta unidad?

La idea es que esta primera parte permita al alumno interiorizarse de la forma de las instrucciones básicas y de la estructura de un programa. Ello implica:

Definir correctamente la estructura de un programa en C.

Identificar las distintas formas de definir tipos.

Diferenciar constante y variable.

Escribir correctamente las instrucciones básicas.

2.1.- INTRODUCCIÓN

Comenzamos el estudio de esta unidad focalizando nuestros esfuerzos en la comprensión de algunos conceptos que nos acompañarán no sólo a lo largo del presente curso, sino que deben ser el punto de partida para poder ampliar nuestros conocimientos en todo el campo del saber asociado a la programación.

El lenguaje C se creó en la década de 1970 con el objetivo de disponer de un lenguaje de programación estructurado, de alto nivel y propósito general orientado hacia los nuevos conceptos de programación, especialmente para desarrollar el sistema operativo Unix, fácil de aprender y con la complejidad suficiente para permitir una correcta preparación de los programadores. Por eso existe una gran similitud entre la sintaxis del C y el lenguaje coloquial con el que planteamos el problema.

Para empezar, podemos remitirnos al siguiente ejemplo, ampliamente utilizado como primer programa por los educadores:

```
#include <stdio.h>
```

```
int main(void)
{
    printf("Hola Mundo\n");
    return 0;
}
```


Este es el ejemplo típico de “entrada” al mundo de la programación en C, representa no sólo con su mensaje, sino también con sus formas el inicio del camino. Podemos ver que hay una necesidad de formas, hay palabras que son importantes y entendibles solamente por el programa, hay un comienzo y un fin, un encabezado con un nombre; en definitiva hay un código, hay una secuencia, y con todo ello un “entendimiento” entre lo que allí está escrito y lo que la computadora debe hacer. Este programa, imprimirá en la pantalla el mensaje “Hola Mundo” cuando sea ejecutado. Es el primer paso...

2.2.- DEFINICIONES DE PROGRAMA Y ALGORITMO

Volvemos sobre esta cuestión que ya hemos estado analizando en la primera unidad de presentación de la asignatura. Las preguntas iniciales que debemos hacernos son: ¿qué es un programa?, ¿para qué programar?, ¿por qué programamos?. Para poder encontrar una base de respuesta a alguna de estas preguntas, podríamos tomar un diccionario de los que tenemos en nuestros hogares o ingresar a alguno on-line, y escribir en los renglones que tenemos disponibles a continuación, la definición de ‘*programa*’.



programa:_____

Seguramente leyendo la presente definición encontraremos muchas acepciones que tienen relación directa con lo que es un “*programa computacional*” y otras que no tienen nada que ver con la misma. El siguiente ejercicio consiste en seleccionar dentro de las distintas acepciones a la palabra ‘programa’, aquellas que pueden tener relación con un “*programa computacional*”, de modo que la definición seguramente se reducirá a:



programa:_____


Sin conocer lo que han escrito como definición de programa podría creer que la misma se aproxima mucho a la que vamos a utilizar para homogeneizar criterios en nuestro curso.

Vamos a definir *programa* como una secuencia de instrucciones ordenadas que nos permiten resolver un problema. La resolución del problema obviamente necesita de datos de partida que serán procesados por el programa y devueltos como datos de salida por el mismo. Esto indica que todo programa está asociado a un proceso. Este proceso no es ni más ni menos que el algoritmo que resuelve el problema planteado. Una vez más tenemos una palabra que no hemos usado nunca antes, la misma es ‘*algoritmo*’. Nuevamente vamos a recurrir a nuestro diccionario de cabecera para definir la palabra algoritmo:



algoritmo: _____

Seguramente tendrá mucho de parecido con la presente definición: un algoritmo es una secuencia de instrucciones ordenadas de manera lineal (esto indica que se ejecutan una detrás de la otra siguiendo un orden establecido) y pudiendo repetirse el mismo código en reiteradas oportunidades.

Basándonos en estas definiciones que aún no están completas y sólo encierran una idea de lo que queremos hacer, es que podemos comenzar a trabajar sobre los programas en C, desarrollando algoritmos que nos permitan solucionar determinados problemas. 

2.3.-PROGRAMAS EN C

Un programa en C tiene una forma dada – a grandes rasgos - por:

- un encabezado;
- una declaración de constantes, tipos y variables;
- y un desarrollo de programa.

Para un programa se debe tener una función main().

Una función tiene la forma:

```
tipo nombre_de_la_funcion (parámetros)
{
```

```
    variables locales
```

```
    instrucciones C
```

```
}
```

Si la definición del tipo es omitida, C asume que la función regresa un tipo entero. A continuación se muestra un ejemplo de programa:

```
/* Programa ejemplo */
```

```
main()
{
    printf( "Hola C\n" );
    exit (0);
}
```

Algunos comentarios sobre este programa:

- C requiere un punto y coma al final de cada línea.
- printf es una función estándar de C, la cual es llamada desde la función main().
- \n significa salto de línea.
- exit() es también una función estándar que hace que el programa termine. En el sentido estricto no es necesario usarla ya que es la última línea de main() y de cualquier forma terminará el programa.

De manera que el siguiente programa hace lo mismo que el anterior:

```
main()
{
    printf( "Hola C\n" );
}
```

En caso de que se hubiera llamado a la función printf de la siguiente forma:

```
printf("0\n.1\n..2\n...3\n");
```

La salida tendría la siguiente forma:

0
.1
..2
...3

🧐 Todos los programas requieren de una función main()



Como pueden ver, el programa tiene una forma que debemos respetar. El lenguaje tiene formas y reglas que debemos aceptar.

ASIGNACIÓN

La asignación es la acción de cargarle un valor a una variable (definimos variable como aquel elemento en el cual es posible almacenar datos en forma transitoria, es decir que el valor contenido en la misma puede variar a través del tiempo). La misma se denota a través de "=", del lado izquierdo del signo se coloca la variable destino a la cual se le carga el valor, mientras que del lado derecho se coloca el valor que le queremos cargar a la variable destino. Podemos utilizar los siguientes ejemplos para comprenderlo:

z=8 donde **z** es una variable entera (int)
p=2.11 donde **p** es una variable real (double)
k='m' donde **m** es una variable caracter (char)

🧐 La asignación se realiza a través de =

Para declarar una variable en C, se debe seguir el siguiente formato:

tipo lista_variables;

tipo es un tipo válido de C y *lista_variables* puede consistir en uno o más identificadores (nombres de variables) separados por una coma. Un identificador debe comenzar con una letra o un guión bajo.

Ejemplos:

```
int i, j, k;  
float x,y,z;  
char ch;  
int _k,_h,otras,var2;
```

Una **variable global** se declara fuera de todas las funciones, incluyendo a la función main(). *Una variable global puede ser utilizada en cualquier parte del programa.*

Por ejemplo:

```
short num, sum;  
int num2, sum2;  
char letra1;
```

```
main()  
{  
...  
}
```

Las variables num, sum, num2, sum2 y letra1 son globales por la forma en que fueron definidas, ello implica que podrán utilizarse en cualquier parte del programa.

Es también posible inicializar variables globales usando el operador de asignación =, por ejemplo:

```
float sum= 0.0;  
int sum2= 0;  
char letra1= 'A';
```

```
main()  
{  
...  
}
```

En este caso, cuando se inicia la ejecución del programa, la variable sum contendrá el valor 0.0; la variable sum2 el valor 0 y la variable letra1 tendrá cargado el valor 'A'. Aclaración: cargar y asignar son sinónimos para el contenido del presente módulo.



Recordar: Las variables globales se pueden usar en cualquier parte del programa.

A propósito, es importante definir qué es ejecutar un programa. La ejecución es el proceso mediante el cual una computadora lleva a cabo las instrucciones de un programa informático. Ejecutar un programa implica que éste estará en estado de ejecución y, por ende, en memoria, hasta que se finalice. Hablar de ejecutar un programa o de correr un programa es hablar de lo mismo para el presente módulo (esta noción se desprende de las palabras *execute* y *run* que se utilizan indistintamente para referirse a la acción de “hacer” lo que indican cada una de las instrucciones y en la secuencia lógica que las mismas indican, es decir, las acciones se van ejecutando una detrás de la otra en la forma que fueron escritas por el programador). Hasta la mismísima Real Academia Española, en la sexta acepción para el vocablo ejecutar se refiere a algo relacionado con la Informática y dice: “6. tr. *Inform.* Realizar las operaciones especificadas por un programa de un ordenador.”, lo que refuerza aún más el concepto que estamos explicando.

Volviendo a la inicialización de las variables, para el caso que tomamos como ejemplo, también puede escribirse de la siguiente forma:

```
float sum;  
int sum2;  
char letra1;  
  
main()  
{  
    sum = 0.0;  
    sum2 = 0;  
    letra1 = 'A';  
  
    ...  
}
```

En C también se permite la asignación múltiple usando el operador =, por ejemplo:

```
a = b = c = 12;
```

En este caso, la asignación se hace de derecha a izquierda, permitiendo simplificar bastante la escritura de las acciones en una sola línea en lugar de tres. La forma en la que se ejecuta esta línea de código es la siguiente: primero se asigna el valor 12 a la variable *c*, luego el valor que tiene cargado *c* (que es el 12) se asigna a la variable *b*, que queda entonces con el valor 12, y por último, el valor cargado en *b* se copia en *a*, o sea que *a* recibe también el valor 12 que está alojado en *b*.

Esta forma es obviamente muy útil porque permite simplificar código cuando muchas variables deben recibir el mismo valor.

El código equivalente para una carga simple (cada variable por separado) sería:

```
c = 12;  
b = 12;  
a = 12;
```

En este caso, se ha respetado inclusive el orden temporal en el que cada variable se carga del valor 12, primero *c*, luego *b* y por último *a*.

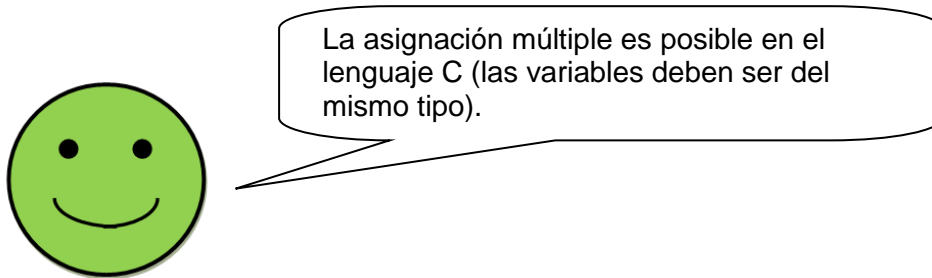
También es posible efectuar algunos cálculos antes de asignar el resultado a un grupo de variables. Por ejemplo,

```
a=1;  
b=c=2;  
a = b = c = a + b + c;
```

en este caso la suma de los valores de las variables *a*, *b* y *c* se guarda en *c*; o sea que como *a* vale 1, *b* es 2 y *c* es 2, tenemos que la suma de las tres variables es $1+2+2=5$ que será el valor que se cargue en la variable *c*.

Entonces tenemos que *c* se carga con 5, luego *b* se carga con el valor de *c*, por eso también queda cargada con 5 y por último *a* se carga con el valor de *b* (que es 5). Como consecuencia, las variables *a*, *b* y *c* quedan cargadas al final de la línea de asignación múltiple $a = b = c = a + b + c$; con el valor 5.

Algo muy importante que se debe tener en cuenta es que la asignación múltiple se puede llevar a cabo si todas las variables son del mismo tipo. En el caso que hemos estado trabajando, *a*, *b* y *c* deberían ser del mismo tipo (por ejemplo, podrían ser todas enteras – tipo `int`).



Este último comentario permite vislumbrar una noción básica del lenguaje C. La definición de variables debe hacerse siempre antes que comencemos a escribir instrucciones en un bloque de programa. Esto es algo que enunciamos aquí pero que trabajaremos muy profundamente un poco más adelante.

Algunos ejemplos para terminar con la asignación:

`a = 3;`
a queda cargada con el valor 3

`b = 1;`
b queda cargada con el valor 1

`c = d = 2;`
d y *c* quedan cargadas con el valor 2

`a = b = c = d = a + b - c * 2 + d * 3;`
d, *c*, *b* y *a* quedan cargadas con el valor que devuelve la operación $a + b - c * 2 + d * 3$

DELIMITADOR

Para indicar el final de una instrucción se utiliza el delimitador punto y coma. De esta forma, cada vez que una línea de instrucción esté completa deberá colocarse este “;” indicando el final de la misma, y que a partir de ahí comienza una nueva instrucción.

Por ejemplo, si se asigna el valor 7 a la variable *y*, la instrucción completa incluyendo el delimitador será:

y=7;

El punto y coma al final de la línea indica que esa instrucción está completa y, que a partir de allí comienza una nueva instrucción.



El delimitador es el ;

INSTRUCCIONES BASICAS EN C

a.- Estructuras condicionales – la sentencia IF

Esta instrucción especifica condiciones para que una acción sea ejecutada.

La forma básica que tiene esta instrucción es la siguiente:

```
if <condición>
    <acción/acciones1>
else
    <acción/acciones2>
```

Si la condición después del “if” es verdadera, las acciones que están después se ejecutarán. Si en cambio la condición después del “if” es falsa, las acciones que se ejecutarán son las que están después del “else”.

La parte asociada al “e/se” puede no existir, haciendo que la instrucción se reduzca a la forma más usualmente utilizada:

```
if <condición>
    <acción/acciones>
```

En esta, si la condición es verdadera las acciones que están después se ejecutarán. Si la condición es falsa, no ejecutará ninguna acción.

Ejemplo: Para poder visualizar más concretamente esta instrucción, podemos utilizar el siguiente ejemplo,

```
if (p==3)
```

$z=8;$

La condición es: $p=3$. La condición será verdadera si la variable p vale 3.


La acción es: $z=8$. La acción que se ejecutará es asignarle 8 a la variable z .

Por lo tanto, lo que aquí dice es que si el valor de la variable p es igual a 3 entonces asigna el valor 8 a la variable z . Debemos con estas instrucciones comenzar a trabajar escuchando lo que nuestro interlocutor quiere, cada vez que nos solicita una solución de un problema utilizando un programa. En este caso él nos dirá “si el valor de p es 3 entonces quiero que se le asigne el valor 8 a la variable z ”.



Relea el párrafo anterior y reflexione acerca de la forma coloquial de la instrucción y de su forma en el lenguaje C. ¿La encuentra parecida?

Esto es lo que nosotros expresamos coloquialmente cada vez que queremos hacer algo. De esta forma es más que importante que comencemos a saber escuchar qué es lo que nuestros interlocutores desean que realice un programa, para utilizar específicamente la instrucción que corresponde. Este, es quizás el punto clave sobre el que todo programador debe poner especial énfasis. La tarea de comprensión de la inquietud de nuestro interlocutor será lo que a través de nuestro trabajo, garantice el correcto funcionamiento de nuestro programa, y que el mismo alcance el fin para el cual está siendo creado.

 Debemos comprender cuál es nuestro problema para encontrarle una solución y luego *programar*. La programación debe ser la última tarea en nuestro esquema de trabajo.

Ejemplo: Si utilizamos la siguiente instrucción,

```
if (p==3)
    z=8
else
    z=11;
```

La condición es: $p=3$. La condición será verdadera si la variable p vale 3.

La acción es: $z=8$. La acción que se ejecutará es asignarle 8 a la variable z *si la condición $p=3$ es verdadera*.

La acción es: $z=11$. La acción que se ejecutará es asignarle 11 a la variable z *si la condición $p=3$ es falsa*.

Por lo tanto, resumiendo y volviendo al lenguaje coloquial, tenemos que “si p es igual a 3 entonces z quedará cargado con el valor 8; si en cambio p tiene cualquier otro valor, z quedará cargado con el valor 11”. Esto es lo que dice nuestra instrucción “if” precedente.

Obviamente, es posible colocar más de una acción a cada condición. En el caso que esto suceda, deberán utilizarse los delimitadores “{” y “}” (son los delimitadores de inicio y fin) para indicar las acciones que tiene comprendidas esa condición.

Ejemplo: “Si p es igual a 3 quiero asignarle el valor 8 a la variable z y el valor 1 a la variable m , en cambio si p es distinto de 3 quiero asignarle el valor 11 a z y 2 a m ”.

Nótese que este enunciado debe escribirse de la siguiente manera siguiendo las reglas que hemos ido definiendo en la presente unidad:

```
if (p==3)
{
    z=8;
    m=1;
}
else
```



```
{
    z=11;
    m=2;
};
```



Otra vez, la forma coloquial coincide con la del lenguaje C. Cada día que pasa, “son más parecidas”.

Veamos que las acciones están comprendidas entre los “{” y “}” correspondientes a la rama del entonces y a la rama del “e/se”. En cada una hay dos acciones. Al final de cada acción de las que se encuentran en la primera posición tenemos un delimitador que indica que la instrucción terminó y que luego viene otra instrucción. Al final de la segunda instrucción tenemos “,” indicando que terminó la instrucción pero en muchas versiones de C no es necesario colocar este segundo delimitador “,” porque “}” actúa por sí sola como delimitador. De esta manera, en aquellos compiladores que lo admiten, podemos no escribir “,” antes de un delimitador “}”. Luego del “}” asociado al “e/se” se observa que hemos colocado un “,”, el mismo indica que la instrucción “if” termina en ese “,”.



Las tres formas básicas que admite la instrucción “if” son las siguientes:

```
if (condición)
    acción;
```

...0

```
if (condición)
    acción 1;
else
    acción2;
```

...0

```

if (condición 1)
acción 1;
else if (condición2)
acción2;
...
else
acción n;

```

El flujo lógico de esta estructura es de arriba hacia abajo. La primera sentencia se ejecutará y se saldrá de la estructura if si la primera condición es verdadera. Si la primera condición fue falsa, y existe otra condición, se evalúa, y si la condición es verdadera, entonces se ejecuta la sentencia asociada. Si existen más condiciones dentro de la estructura if, se van evaluando éstas, siempre y cuando las condiciones que le precedan sean falsas.

La sentencia que está asociada a la palabra reservada else, se ejecuta si todas las condiciones de la estructura if fueron falsas.

Ejemplo de estructura if en un programa:

```

main()
{
    int x, y, w;

    .....

    if (x>0)
    {
        z=w;
        .....
    }
    else
    {
        z=y;
        .....
    }
}

```

Con esto completamos básicamente los primeros conceptos asociados a la instrucción "if". De esta manera podremos muy pronto comenzar a escribir nuestros primeros programas utilizando este lenguaje. Es obvio que el if por su relación condición/decisión es una de las instrucciones más usadas en cualquier lenguaje, siempre es necesario tomar decisiones basadas en condiciones, y eso es exactamente lo que el if permite hacer.

Entonces, con la instrucción **if** tenemos el primer ejemplo de un enunciado condicional.

b.- El operador ?

El operador condicional “?” es más eficiente que la sentencia **if**. El operador “?” tiene el siguiente formato:

expresión 1 ? expresión2 : expresión3;

Que es equivalente a:

```
if (expresión 1)
    expresión2
else
    expresión3;
```

Por ejemplo, para asignar el máximo de *a* y *b* a la variable *z*, usando “?”, tendríamos:

z = (*a*>*b*) ? *a* : *b*;

que es lo mismo que:

```
if (a > b)
    z = a;
else
    z = b;
```

Esto implica que su forma es muy sencilla, por ello hay que tenerla en cuenta porque permite reducir fuertemente cantidad de líneas de código. Por eso también hablamos de mayor eficiencia.

c – scanf y printf

Estas dos instrucciones permiten la lectura y la escritura de variables sobre un dispositivo determinado. El dispositivo podría ser por ejemplo el teclado, una pantalla, una impresora, un archivo o cualquier otro de los que conocemos como de salida.



scanf y printf nos permiten comunicarnos con el entorno, con el usuario.

La función printf tiene un caracter especial para formatear (%) – al que le sigue un caracter que define un cierto tipo de dato (formato) para una variable.

%c caracteres

%s cadena de caracteres

%d enteros

%f flotantes

Por ejemplo:

```
printf("%c %d %f",ch,i,x);
```

La sentencia de formato se encierra entre " ", y luego se enumeran cada una de las variables que contienen los valores que van a imprimirse. Hay que asegurarse de que el orden de los formatos coincida con el orden de las variables de cada uno de los tipos de formato enumerados. Esto es, si una variable es caracter, hay colocar el formato %c y luego enumerar una variable de tipo caracter y no una de tipo entero en el lugar donde está el formato %c sino tendremos inevitablemente alguna diferencia o error. En pocas palabras, esto es algo muy importante, ya que de otro modo existirán incompatibilidades.

scanf() es la función para ingresar valores a variables. Su formato es similar a printf. Por ejemplo:

```
scanf("%c %d %f %s", &ch, &i, &x, cad);
```

Observar que se antepone & a los nombres de las variables, excepto a la cadena de caracteres. Esto tiene que ver con una cuestión de los lugares de memoria donde se guardan los valores de las variables. Ya lo veremos cuando trabajemos punteros hacia el final del curso, por ahora solamente acepten que esto es una forma que debe ser respetada. Entonces, debemos considerar siempre que para la lectura de variables utilizando la instrucción scanf, debe colocarse el signo & antes del nombre de las variables que queremos leer de teclado excepto cuando se trate de una cadena de caracteres, que es el único caso en el que no lo necesita.

El parámetro que tiene `printf` es justamente la cadena de caracteres que nosotros deseamos sea enviada al dispositivo considerado.

Por ejemplo, si escribimos **`printf`** (“hola”); lo que aparecerá en pantalla (que es el dispositivo tomado por defecto cuando no se nombra ningún otro) será lo siguiente:



hola

Sobre el margen izquierdo aparecerá la palabra “hola” que es justamente lo que nosotros enviamos a escribir sobre la pantalla.

En este caso, la utilización del “*printf*” hace que el cursor se quede preparado en la última posición válida una vez que escribió sobre la pantalla. De este modo si enviamos otra palabra a pantalla utilizando la instrucción “*printf*”, lo que conseguimos es que la segunda palabra aparezca al lado de la primera. Esto es, si la secuencia de instrucciones es la siguiente:

`printf` (“hola”);

`printf` (“chau”);

lo que aparecerá en la pantalla es lo que se ve a continuación:



holachau

Lo que vemos es que la palabra “*chau*” se escribió a continuación de la palabra “hola”.

Si escribiéramos una tercera palabra a continuación:

```
printf ("hola");
```

```
printf ("chau");
```

```
printf ("jaja");
```

lo que lograríamos sería:



```
holachaujaja
```

Aquí se ve muy claramente que el *“printf”* va permitiendo la escritura sobre el dispositivo seleccionado (en este caso la pantalla) sin ningún tipo de control y caracter por caracter uno a continuación del otro.

Muchísimas veces a nosotros nos interesa mostrar la información de una manera más ordenada. Para ello podemos recurrir a una salida con formato, la que corresponde es la *“\n”*. Con el *“\n”* lo que logramos es escribir la cadena de caracteres correspondiente que tiene asociada instrucción y, automáticamente pasar a la primera posición de la siguiente línea. De esta forma, cuando volvamos a escribir, estaremos escribiendo sobre la siguiente línea.


Por ejemplo, si la secuencia de instrucciones que tenemos es:

```
printf ("hola\n");
```

```
printf ("chau\n");
```

```
printf ("jaja\n");
```

Esto se verá representado en la pantalla como:



```
hola  
chau  
jaja
```


Podemos llegar también a combinar estas instrucciones como por ejemplo:

```
printf ("hola");
```

```
printf ("chau\n");
```

```
printf ("jaja\n");
```

Esto generará lo siguiente en la pantalla:

```
holachau  
jaja
```

Conociendo las instrucciones “if”, “scanf” y “printf” es posible comenzar a desarrollar aplicaciones. Para ello debemos trabajar sobre la forma de un programa en principio muy básico, para que con el transcurso del tiempo vayamos agregando estructuras más complicadas.

Nuestro objetivo es lograr generar un programa a partir de un pensamiento lógico muy profundo por más sencillo que sea de resolver el problema de modo de poder ir creciendo en cuanto a aplicaciones y cantidad de instrucciones, tratando de establecer una forma de trabajar, una forma de pensar...

La función **scanf** de la biblioteca estándar del lenguaje C permite asignar a una o más variables, uno o más valores (datos) recibidos desde la entrada estándar (el teclado). La sintaxis de su llamada es:

```
scanf( <cadena_de_control> [, <lista_de_argumentos> ] )
```

En la <cadena_de_control>, el programador debe indicar el **formato** de entrada de los datos que se van a recoger por teclado. Para ello, se puede hacer uso de:

- Especificadores de formato.
- Otros caracteres

En **formato** se especifica qué tipo de datos se quieren leer. Se utiliza la misma descripción de formato que en **printf** (o sea % seguido del tipo de datos).

Veamos ahora un ejemplo de programa completo con la instrucción printf:

```
#include <stdio.h>
main()
{
    printf ("Esta es una línea de texto.");
    return 0;
}
```

Analizando el código de nuestro programa tenemos lo siguiente:

-printf ("Esta es una línea de texto."); es la instrucción que imprime en pantalla el texto “Esta es una línea de texto.” Esta instrucción llamada printf() está definida en un archivo que contiene el conjunto de comandos estándar del lenguaje (ese conjunto de instrucciones también se conoce como librería estándar). El archivo que contiene la librería estándar de instrucciones es el stdio.h que aparece en la primera línea del programa.

-#include <stdio.h> es entonces la línea que indica que están disponibles para su uso en este programa de las instrucciones estándar que el lenguaje tiene. La palabra reservada *include* justamente denota la inclusión para su uso en este programa de todas las instrucciones estándar que se encuentran definidas dentro del archivo stdio.h

-La instrucción return <expresión>; es utilizada en este lenguaje para indicar el valor de retorno de una función o de un programa. En el caso de este programa, como la idea es solamente imprimir, lo que devuelve no es importante, entonces podría finalizarlo con cualquier valor en el return, en este caso se devuelve el valor 0 (cero).

-Recordemos también que el lenguaje C utiliza como delimitador el “;” luego de cada instrucción y se usan los “{” y “}” para indicar los límites de inicio y fin del código del programa.

La pantalla, una vez ejecutado el programa indicará lo siguiente:

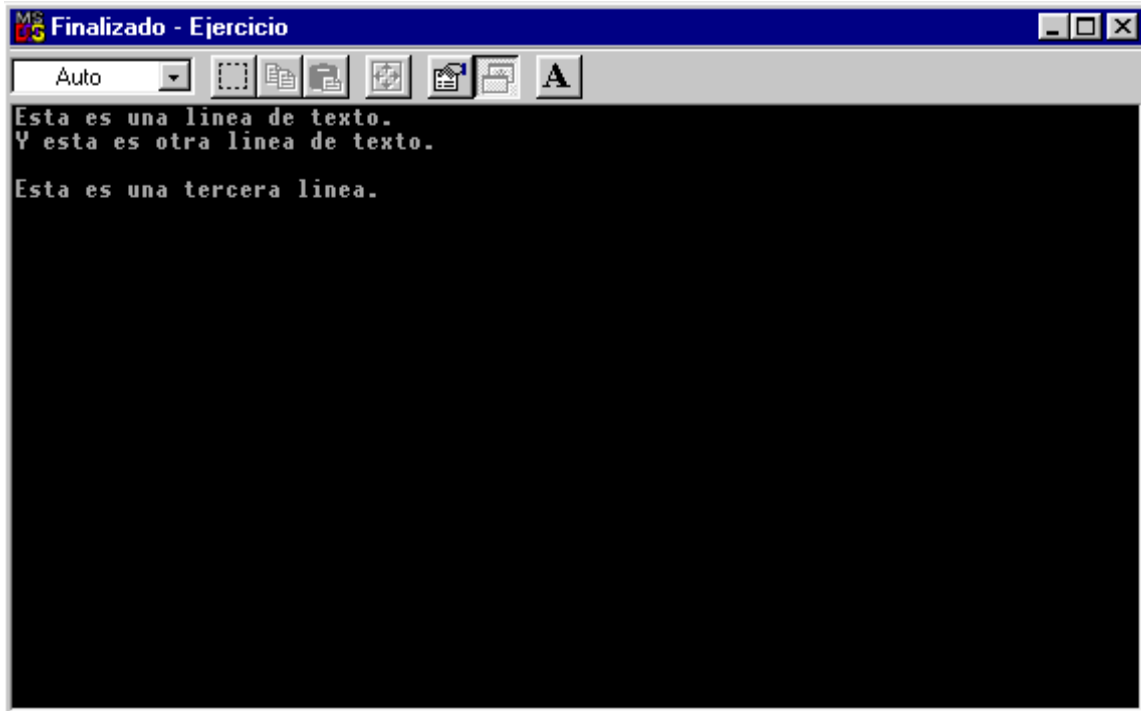


Otro ejemplo:

```
#include <stdio.h>
main ()
{
    printf ("Esta es una línea de texto.\n");
    printf ("Y esta es otra ");
    printf ("línea de texto.\n\n");
    printf ("Esta es una tercera línea.\n");
    return 0;
}
```

Obsérvese que ahora este programa tiene cuatro líneas con **printf ()**, la línea superior será ejecutada en primer lugar seguida de las otras tres líneas en el orden en que aparecen. El caracter de barra invertida (\) es utilizado en la función **printf ()** para indicar que luego de él sigue un caracter especial de control. En este caso, la "n" indica la petición de una nueva línea de texto, es una indicación para regresar el cursor al lado izquierdo de la pantalla y a la vez moverlo una línea hacia abajo. Se puede colocar un comando "\n" en cualquier parte e iniciar una nueva línea, incluso en la mitad de una palabra y de esta manera dividir la palabra entre dos líneas. La primera función **printf ()** despliega una línea de texto y regresa el cursor. La segunda **printf ()** despliega otra línea de texto pero sin regresar el cursor, de tal manera que la

tercera línea aparece al final de la segunda, entonces, le siguen dos retornos de cursor dando como resultado un espacio en blanco. Finalmente la cuarta instrucción **printf ()** despliega una nueva línea de texto seguida por el retorno del cursor, finalizando el programa. Esta sería la forma en la que se presenta la pantalla una vez que se terminó de ejecutar el programa.

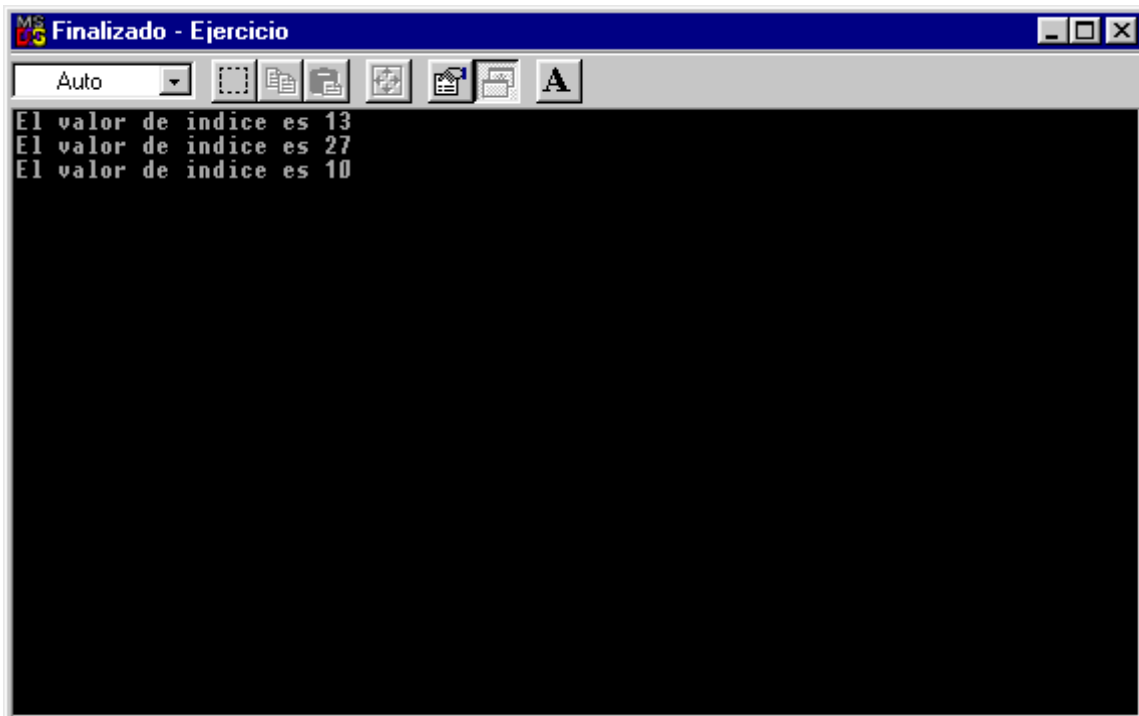


Vayamos ahora a un nuevo ejemplo:

```
# include <stdio.h>
main()
{
    int indice;
    indice = 13;
    printf("El valor de indice es %d\n", indice);
    indice = 27;
    printf("El valor de indice es %d\n", indice);
    indice = 10;
    printf("El valor de indice es %d\n", indice);
    return 0;
}
```

En este caso, la primera línea, indica la definición de una variable de tipo entera llamada índice (**int indice;**). La palabra **int** es una palabra reservada de C y no puede ser utilizada con otros fines, define una variable que almacena un número entero dentro de un rango predefinido de valores, definiremos el

actual rango posteriormente. El nombre de la variable será **índice**. El punto y coma al final de la línea es un delimitador de enunciado como se explicó anteriormente. Obsérvese que, aunque hemos definido una variable, no le asignamos a ésta un valor, por ello se dice que contiene un valor indefinido. Siguiendo con el programa, vemos que hay tres enunciados que asignan un valor a la variable **índice**, pero solo uno por vez. Así pues, primero se le asigna a **índice** el valor de 13, imprimiéndose en la línea siguiente. Después se le asigna el valor 27 imprimiéndose en la línea siguiente, y finalmente se le carga el valor de 10 imprimiéndose luego. Está claro que **índice** es una variable que puede almacenar muchos valores diferentes pero solo uno a la vez. Luego de la ejecución del programa, la pantalla mostrará:



```

MS-DOS Finalizado - Ejercicio
Auto
El valor de indice es 13
El valor de indice es 27
El valor de indice es 10
  
```

Si ahora nos concentramos en las instrucciones **printf ()**, tenemos que todas son idénticas. Algo que encontramos es el carácter %, que se utiliza para indicar que se va a mostrar un valor. El carácter que le sigue al símbolo % señala cuál es el tipo de dato que se va a mostrar, en este caso es una d, lo que indica que el valor es de tipo entero decimal. Después de la d encontramos la \n para marcar que allí se produce el retorno del cursor.

El formato puede ser básicamente:

%d Entero decimal

%u Entero decimal con signo

%x Entero hexadecimal

%c Carácter

%f Coma flotante (**float**)

%lf Coma flotante (**double**)

La estructura básica de un programa –como ya vimos- está formada por un encabezado, una definición de constantes, tipos de datos y variables, y la parte de lógica y algoritmo del programa propiamente dicha donde se desarrolla la solución a la aplicación.

Un programa escrito en lenguaje C está compuesto por una o más funciones. Como ya es sabido, una función es un programa que sirve para realizar una tarea determinada, por ejemplo, la función **scanf** sirve para leer datos desde el teclado.

Existe una función que está presente en todos los programas escritos en lenguaje C, su misión es marcar el inicio y fin de la ejecución de cada uno de ellos; es la función principal, la primera que se ejecuta; es la función **main**. Su sintaxis "básica" es:

```
<tipo_de_datos> main()  
{  
    <bloque_de_instrucciones>  
}
```

La función **main** contiene al **bloque de instrucciones principal de un programa**, dentro de los caracteres *abrir llave* (**{**) y *cerrar llave* (**}**).

Los paréntesis "**()**" escritos después de **main** sirven para indicar que el identificador **main** es una función. Es importante comprender que **main** no es una palabra reservada de C. Ningún identificador de función lo es.

Finalmente, no podemos pasar por alto que delante de **main** se debe escribir el

nombre de un tipo de datos, que será el tipo del dato que se devolverá mediante el **return** que se coloca dentro de la función (o programa). Por ejemplo, véase lo siguiente:

```
int main ()
{
<bloque_de_instrucciones>
}
```

En este caso, se ha escrito la palabra reservada **int**; que indica que esa función devolverá –en el caso de requerirlo- un valor entero, naturalmente la función main es una función entera, puede obviarse la palabra int, y a pesar de ello, la función main seguirá siendo entera, porque naturalmente lo es, está definida como tal en la librería de funciones correspondiente. O sea que también podría escribirse de la siguiente manera:

```
main()
{
    <bloque_de_instrucciones>
}
```

Con esto es posible comenzar a trabajar en la generación de algunos programas sencillos utilizando las estructuras que hemos ido describiendo a lo largo del módulo.



Como vemos, y como lo hemos dicho en algunas oportunidades, las formas son muy importantes.

Ejemplo:

Generar un programa que permita imprimir “Programación I” en la pantalla.

Para ello vamos a utilizar la instrucción `printf` de manera de escribir sobre el dispositivo de salida.



Usted está en condiciones de realizarlo solo. Para ello recurra a los conceptos desarrollados con anterioridad y, utilizando la estructura básica de un programa trate de hacer este ejemplo a continuación. Luego coteje con el texto.

This image shows a blank sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

El programa será el siguiente:

```
#include <stdio.h>

int main ()
{
    printf ("Programación I");
}
```



En este programa, podemos observar lo siguiente:

El estándar de C define un conjunto de **bibliotecas** de funciones, que necesariamente vienen con todo entorno de compilación de C y que incluyen una gran cantidad de funciones preprogramadas.

Las interfases de estos elementos vienen definidas en unos archivos (*header files* – *archivos cabecera*) que deben ser definidos al inicio del programa para que puedan ser utilizados en el mismo. El nombre de estos archivos suele terminar en **.h**

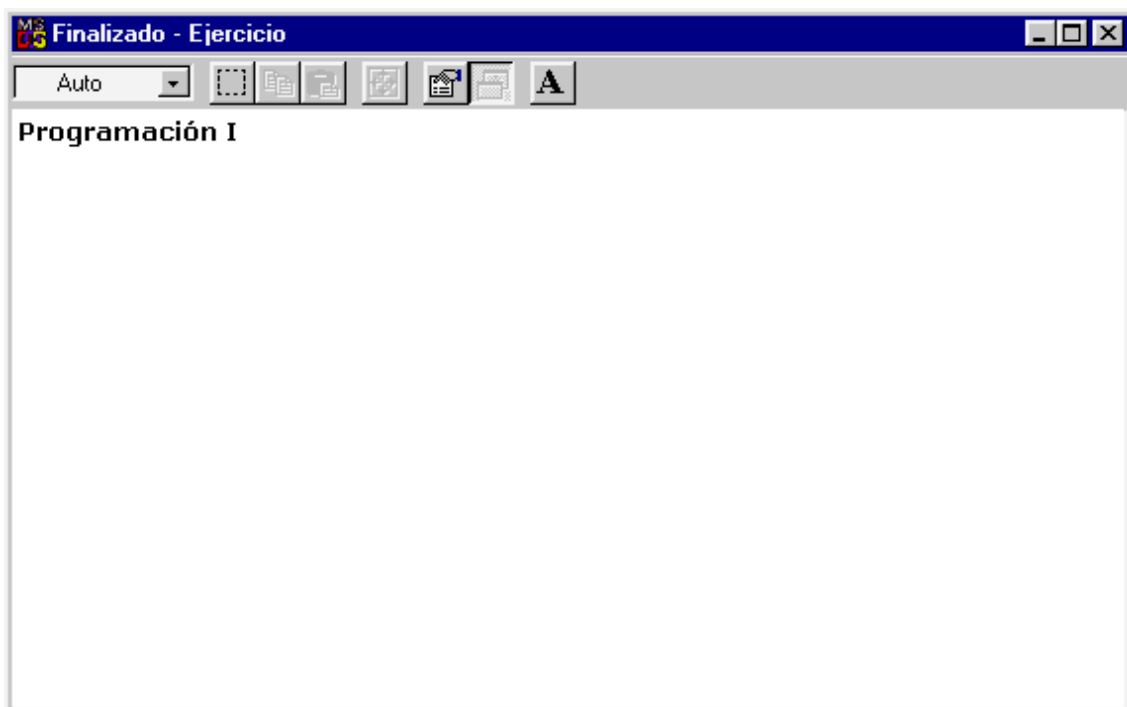
Algunas bibliotecas estándar son:

- entrada y salida de datos (stdio.h) (scanf y printf por ejemplo)
- manejo de cadenas de caracteres (string.h)
- memoria dinámica (stdlib.h)
- rutinas matemáticas (math.h)

Con stdio y stdlib alcanza normalmente para todas aquellas aplicaciones básicas en las que usemos algunas funciones estándar.

Este sencillo programa realiza la escritura sobre pantalla de la palabra “Programación I” (printf ("Programación I");).

De manera que luego de ejecutado el programa, la pantalla se verá como sigue:



Si tuvo dificultades para realizar el ejemplo, le sugiero que relea el texto desde el comienzo del módulo.

d.- Comentarios

En el C original, tienen la forma ***/* cualquier texto */***

Los comentarios se pueden extender varias líneas

No se pueden anidar comentarios (comentarios dentro de otros)

En C++ se usan también comentarios de una sola línea. La sintaxis es

// cualquier texto

Todo lo que se escriba a partir de las dos barras es un comentario. El comentario termina con el final de la línea.

Ejemplos:

```
{
    /*      Esto es un comentario
           que ocupa varias líneas
    */

    // esto es un comentario de C++
    // y esto es otro comentario
}
```

Agregamos comentarios al programa para hacerlo más entendible para el programador o para quien debe modificar luego el código que programó otro. Lo importante es que el texto del comentario no tiene ningún significado para el compilador, por lo tanto, este ignora completamente los comentarios que viene encerrados en caracteres especiales. Entonces, la combinación de barra y asterisco se usa en C para delimitar comentarios como podemos ver en el siguiente código, observe que este programa ilustra una mala técnica al hacer comentarios en demasía pero muestra la forma en que deben realizarse.

```
# include <stdio.h>
/* Este es un comentario que el compilador ignora */
int main()      /* Este es otro comentario ignorado por el compilador*/
{
    printf("Utilizando comentarios "); /* Un comentario
                                       se puede continuar
                                       en otra línea */

    printf ("en C.\n");
```

```

return 0;
}
/* Agregamos aquí un comentario más... */

```

Obsérvese que pueden incluso ubicarse comentarios antes del principio del programa, lo que ilustra que un comentario puede preceder al programa en sí. Una buena práctica de programación es incluir un comentario antes del inicio del programa con una breve descripción del mismo: para qué sirve, qué hace, dónde se va a usar, quién fue su creador, cuál es el departamento en el que se va a usar, etc. Hay que recordar que el comentario debe iniciarse con barra y asterisco, y debe terminarse con asterisco y barra (/* Texto del comentario */). Es muy importante que no se deje espacio alguno entre el asterisco y la barra pues de lo contrario el compilador no sabrá que se trata de un comentario y por ende, se generarán mensajes de error.



Los comentarios son importantes para que quienes deban hacer mantenimiento o modificaciones al sistema puedan tener mayor información de lo que se quiso hacer con esas líneas de código.

Volvamos ahora a la función `scanf()`. Recordemos que con ella se realiza la entrada de datos con formato -desde por ejemplo- el teclado, y se carga lo leído en los argumentos que se le pasan a dicha función.

Vamos a un ejemplo que permite visualizar su uso:

Ejemplo: Generar un programa que imprima la palabra “uno” si se ingresa el valor 1 desde teclado. Si se ingresa cualquier otro valor que imprima la palabra “otro”.

```

#include <stdio.h>
#include <stdlib.h>

```

```

int x;

```

```

main ()
{

```

```

printf ("Ingrese un valor\n");
scanf ("%d", &x);
if (x==1)

```

```
printf ("uno\n");
else
    printf ("otro\n");
}
```



La definición de las variables se realiza utilizando el nombre del tipo, el nombre de la variable y un delimitador “;” ej.: int x;



Si hay más de una variable declarada del mismo tipo, pueden separarse por comas ej.: int x,y,z;

Otro ejemplo con **scanf()**, **if** y **printf()**:

Ejemplo: Generar un programa que imprima la palabra “uno” si se ingresa el valor 1 desde teclado. Si se ingresa cualquier otro valor que imprima el valor ingresado.

```
#include <stdio.h>
#include <stdlib.h>

int x;

main ()
{

    printf ("Ingrese un valor\n");
    scanf ("%d", &x);
    if (x==1)
        printf ("uno\n");
    else
        printf ("%d", &x);
}
```

En este programa podemos observar lo siguiente:

Lo único que se ha modificado respecto del programa anterior es que en el segundo “*printf*” donde antes escribíamos la palabra “otro” en pantalla, ahora directamente se escribe el valor diferente de 1 que ingrese el operador. Ese valor está guardado en la variable x.

Valor ingresado diferente de 1 y el valor leído de teclado.



trate de hacer este ejemplo a continuación. Luego coteje con el texto.

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

```
#include <stdio.h>
#include <stdlib.h>

int x;

main ()
{
    printf ("Ingrese un valor\n");
    scanf ("%d", &x);
    if (x==1)
        printf ("uno\n");
    else
    {
        printf ("%d", &x);
        printf ("Valor ingresado diferente de 1\n", &x);
    }
}
```

En este programa podemos observar lo siguiente:

- Lo único que se ha modificado respecto del programa anterior es que en la rama del “e/se” tenemos dos instrucciones. Esa es la causa por la que hubo que agregar en la misma el “{” y el “}”.
- Las instrucciones asociadas al “e/se” están separadas por “,”.

Si tuvo dificultades para realizar el ejemplo, le sugiero que relea el texto desde el inicio.

e.- Instrucción FOR

La forma que tiene la instrucción “for” es la siguiente:

for (**expresión 1**; **expresión2**; **expresión3**)

acción;

o { **bloque de acciones** }

En donde *expresión 1* se usa para realizar la *inicialización* de variables, usando una o varias instrucciones, si se usan varias instrucciones deberá usarse el operador “,” para separarlas. Por lo general, establece el valor de la variable de control del ciclo. La *expresión2* se usa para la condición de terminación del ciclo y la *expresión3* es el modificador a la variable de control del ciclo cada vez que la computadora lo repite, pero también puede ser más que un incremento.

Por ejemplo:

int X;

```
main()
{
    for( X=3; X>0; X--)
    {
        printf("X=%d\n",X);
    }
}
```

genera la siguiente salida a pantalla ...

```
X=3
X=2
X=1
```

Las siguientes instrucciones “for” son válidas en C. Las aplicaciones prácticas de tales instrucciones no son importantes aquí, ya que tan sólo se intentan ilustrar algunas características/formas que pueden ser de utilidad:

```
for ( x=0; ( (x>3) && (x<9) ); x++ )
```

```
for ( x=0, y=4; ( (x>3) && (x<9) ); x++, y+=2)
```

En el segundo ejemplo se muestra una forma en la que múltiples expresiones pueden aparecer, respetando siempre que estén separadas por una coma “,”

La instrucción “*for*” hace que las instrucciones que contiene sean ejecutadas una vez para cada valor en el rango [valor inicial al valor final].

La variable de control y los valores inicial y final deben ser de un tipo ordinal (discretos con orden) .

Ejemplo: Dado el siguiente “*for*”,

```
for( X=1; X<5; X++)
```

```
    printf("Hoy\n");
```

lo que se genera sobre la pantalla es lo que a continuación se muestra

```
Hoy
Hoy
Hoy
Hoy
```

De esta manera se ha ejecutado tal como se preveía, cuatro veces el lazo que contiene el “*for*”, en este caso constituido sólo por una instrucción “*printf*” que escribe en pantalla la palabra “Hoy”.



La forma de la instrucción “*for*” es la siguiente:

```
for ( expresión 1; expresión2; expresión3)
```

```
    acciones;
```

```
    o { bloque de acciones }
```

Otro ejemplo de ciclo *for*:

```
#include <stdio.h>
```

```
int main()
```

```
{
```

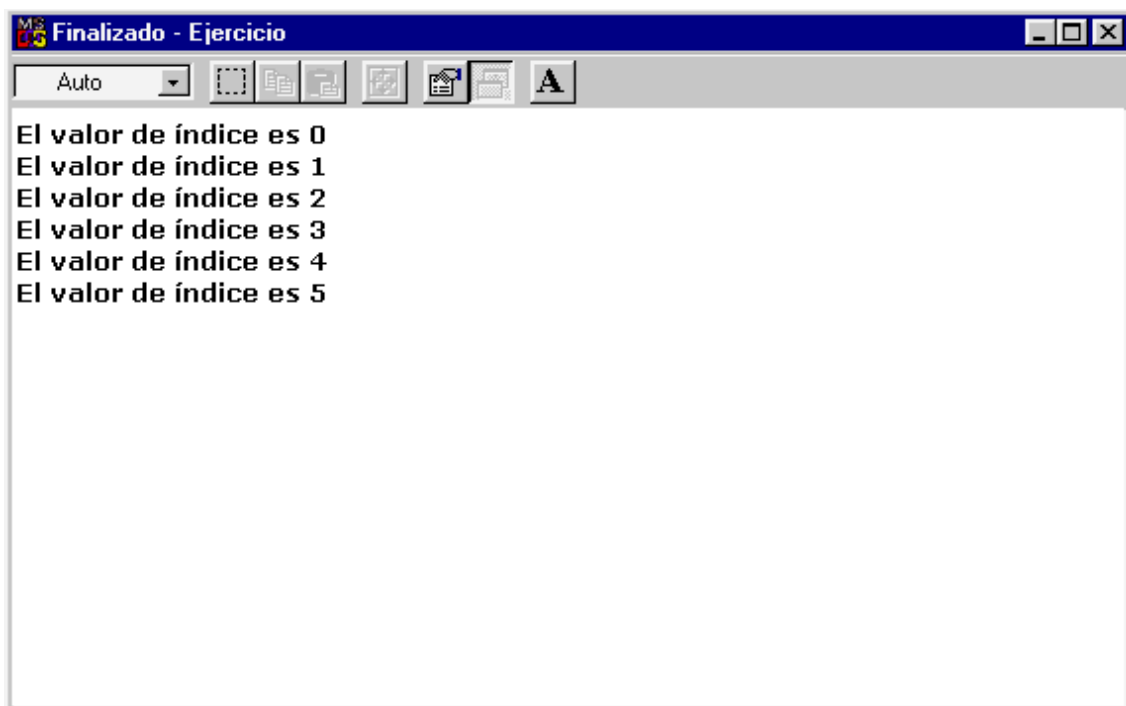
```
    int indice;
```

```
    for(indice = 0 ; indice < 6 ; indice = indice + 1)
```

```
printf ( "El valor de índice es %d\n", indice);  
return 0;  
}
```

Como vimos, el ciclo **for** consiste de la palabra reservada **for** seguida de una expresión entre paréntesis. Esta expresión se compone de tres elementos, cada uno separado por un punto y coma. El primero contiene la expresión "**índice = 0**" y se le llama campo de inicialización. Cualquier expresión en este lugar se ejecuta antes del inicio del ciclo. El segundo lugar, que contiene la expresión "**índice < 6**", corresponde a la evaluación que se hace al principio de cada ciclo y puede ser cualquier expresión que pueda evaluarse como verdadero o falso. Sirve para saber si se ha completado el ciclo del **for** o si todavía debe continuarse ejecutando. La expresión del tercer lugar se ejecuta en cada ciclo luego de que se ejecutan las acciones contenidas en el cuerpo del **for**. Hay que destacar que esto sucede hasta que se alcance la condición de corte del ciclo. Esas acciones son lo que conocemos como cuerpo ejecutable del ciclo **for()**. Esas acciones pueden ser una o varias. Si son varias, deben incluirse las llaves "{" y "}" que denotarán el inicio y fin de las instrucciones que se ejecutarán en cada ciclo del **for()**.

Luego de la ejecución del programa, la pantalla mostrará:





Preste atención a las diferencias entre los ciclos for y while.

f.- WHILE

La instrucción “while” tiene la siguiente forma:

```
while <condición>
```

```
<acciones>
```

Esta estructura tiene una condición que controla la ejecución repetida de las acciones. En la misma, las acciones se ejecutan mientras que la condición sea verdadera. En el caso de que la condición sea falsa, la ejecución del ciclo finaliza. La condición es evaluada siempre antes de que la acción se ejecute, de modo que si la condición al inicio del ciclo es falsa, la acción no será ejecutada nunca.

Desde el punto de vista coloquial, la instrucción “while” puede ser pensada como: *“mientras se cumpla la condición ejecuto las acciones”*.



La forma de la instrucción “while” es:

```
while <condición>
```

```
<acciones>
```

Un ejemplo del ciclo “while” es el siguiente, donde se imprimen los números de líneas desde el 1 al 99, uno debajo del otro.

```
main()
{
    int x=1;
    while ( x < 100 )
    {
        printf("Línea número %d\n",x);
        x++;
    }
}
```

}

A continuación se muestra otro ejemplo de ciclo while:

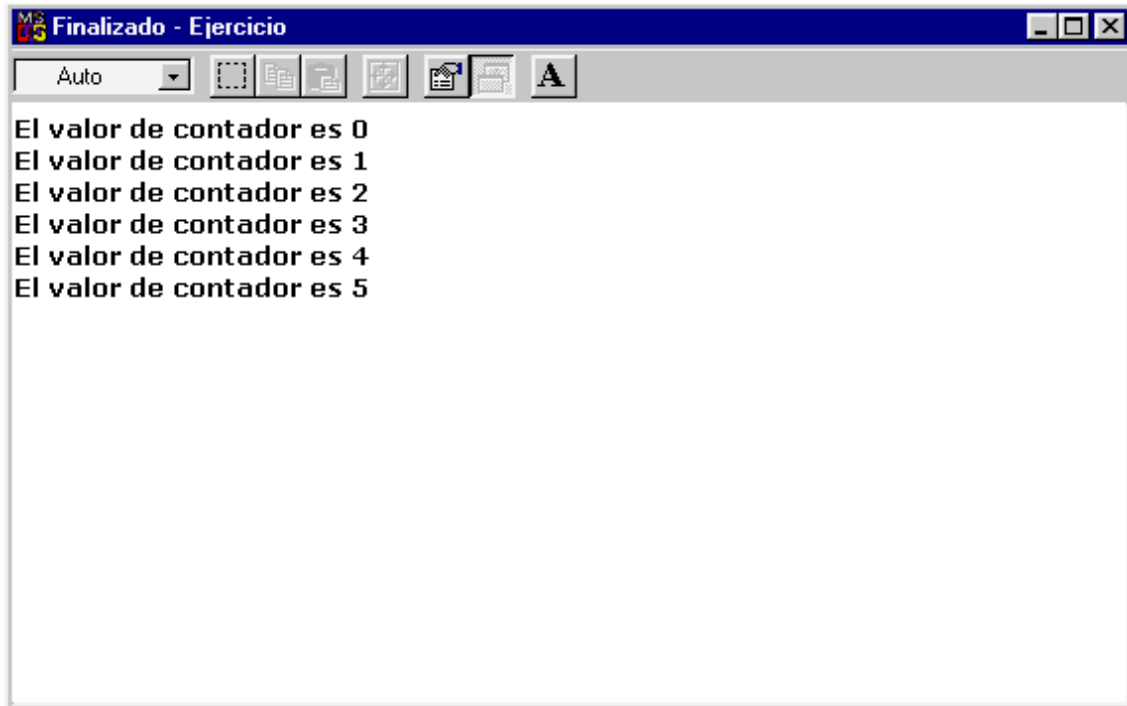
```
/* Este es otro ejemplo del ciclo while. */
#include <stdio.h>
main()
{
    int contador;

    contador = 0;
    while (contador < 6)
    {
        printf ("El valor de contador es %d\n", contador);
        contador = contador + 1;
    }
    return 0 ;
}
```

Este programa empieza con un comentario (`/* Este es otro ejemplo del ciclo while. */`) y la entrada **main ()**. Luego se define una variable de tipo entero a la que llamamos **contador** dentro del cuerpo del programa, esta variable es inicializada a cero para después entrar en el ciclo **while**. La sintaxis del ciclo **while** es justamente como se muestra en el programa. A la palabra reservada **while** le sigue una expresión entre paréntesis que es la condición de entrada en el ciclo y luego una serie de instrucciones encerradas entre llaves. Mientras la expresión entre paréntesis –condición de corte del ciclo- sea verdadera, todos los enunciados entre las llaves se ejecutarán repetidamente. En este caso, debido a que la variable **contador** es incrementada en 1 cada vez que los enunciados entre las llaves son ejecutados, en algún momento se alcanzará el valor 6. Es a partir de ese momento que **contador** ya no es menor a 6 finalizando así el ciclo. El programa continuará entonces ejecutando las instrucciones que siguen a las llaves. Hay algunos comentarios para destacar y que pueden ser de gran utilidad para terminar de comprender el funcionamiento del ciclo: Primero, si la variable **contador** fuera inicializada con un valor mayor a 5, los enunciados dentro de las llaves podrían no ejecutarse por lo que es posible tener un ciclo while que jamás se ejecute. Segundo, si la variable de corte o control (en este caso **contador**) no se incrementa dentro del ciclo, este jamás terminaría y por ende el programa tampoco. Finalmente, si el while

contiene una sola instrucción a ejecutar, entonces no es necesario el uso de llaves.

Para el caso del ejemplo, la pantalla –una vez ejecutado el programa- será:



g.- DO ... WHILE

Al contrario de los ciclos for y while que comprueban la condición al inicio del ciclo, el do ... while la examina al final del mismo. Esta característica provoca que un ciclo do ... while siempre se ejecute al menos una vez. La forma general del ciclo es:

```
do {
    acciones;
} while (condición);
```

Aunque no son necesarias las llaves cuando sólo está presente una instrucción, se usan normalmente para hacer más claros los límites de las acciones.

En el siguiente programa se usa un ciclo do ... while para leer números desde el teclado hasta que uno de ellos sea menor o igual a 100:

```
main()
{
    int num;

    do
    {
        scanf("%d", &num);
    } while ( num>100 );
}
```

Esta estructura tiene una condición que controla la ejecución de las acciones hasta que se cumpla la condición. Esto es, las acciones se ejecutarán mientras la condición no sea falsa. Cuando la condición sea falsa, el ciclo finalizará. La condición es evaluada siempre después de que la acción se ejecute de modo que las acciones se ejecutarán al menos una vez.

Desde el punto de vista coloquial, la instrucción “do...while” puede ser pensada como: *“repetir la ejecución de las acciones hasta que no se cumpla la condición”*.



La forma de la instrucción “do...while” es la siguiente:

```
do {
    acciones;
} while (condición);
```

Vamos a un ejemplo del ciclo do...while:

```
/* Este es un ejemplo del ciclo do-while */
# include <stdio.h>

main()
{
    int i;

    i = 0;
    do
    {
        printf ( "El valor de i es ahora %d\n", i );
        i = i + 1;
    }
```

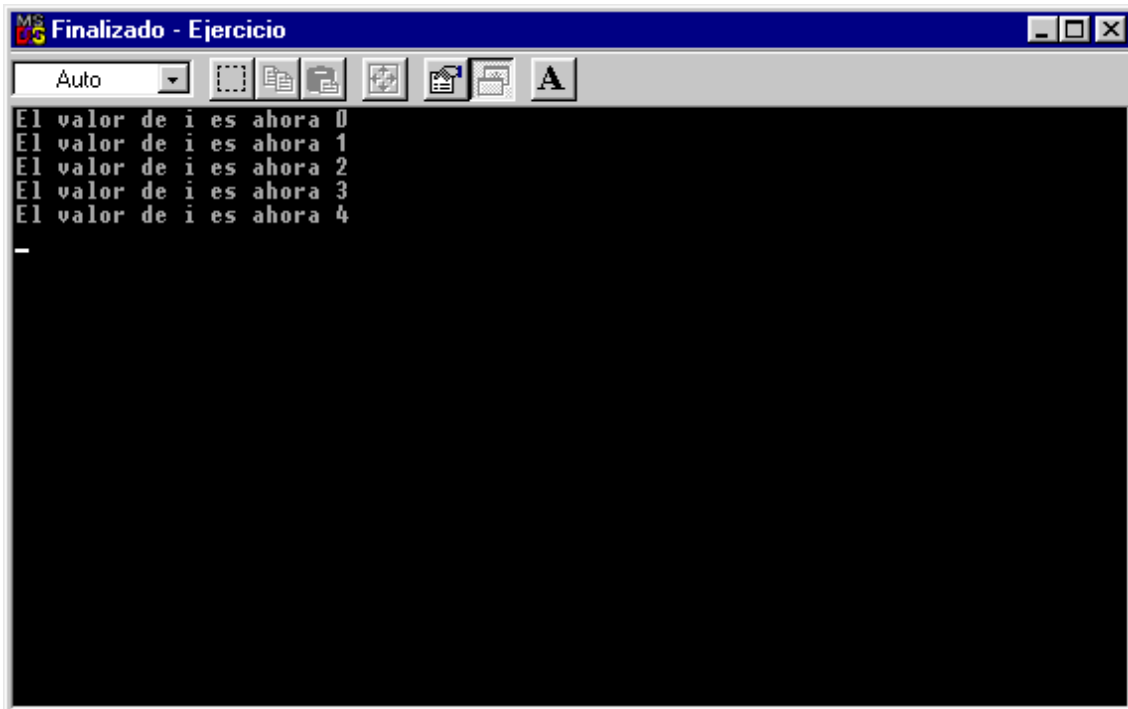


```

    }
    while (i < 5);
    return 0;
}
    
```

Los enunciados entre llaves se ejecutan repetidamente siempre que la expresión entre paréntesis sea verdadera. Cuando la expresión sea falsa, la ejecución del ciclo termina y el control del programa pasa a las instrucciones siguientes. Respecto al ciclo **do-while** cabe repetir lo siguiente: En primer lugar, debido a que la evaluación de verdadero o falso de la condición del ciclo se hace al final del bucle, los enunciados dentro de las llaves se ejecutan al menos una vez. Segundo: si la variable **i** no cambia dentro del ciclo, entonces el programa jamás terminará de ejecutarse, es importante que exista una condición de corte o de salida del bucle y que además pueda cumplirse en algún momento.

Luego de la ejecución de este programa, la pantalla quedará como se muestra:



```

MS-DOS Finalizado - Ejercicio
Auto
El valor de i es ahora 0
El valor de i es ahora 1
El valor de i es ahora 2
El valor de i es ahora 3
El valor de i es ahora 4
_
    
```

h.- switch/case

La instrucción switch/case permite a través del valor que puede tomar una variable, realizar distintas acciones.

Aunque con la estructura if ... else if se pueden realizar comprobaciones múltiples, en ocasiones no es muy elegante, ya que el código puede ser difícil

de seguir y puede confundir incluso al autor transcurrido un tiempo. Por lo anterior, C tiene incorporada una sentencia de bifurcación múltiple llamada switch. Con esta sentencia, la computadora comprueba una variable sucesivamente frente a una lista de constantes enteras o de carácter. Después de encontrar una coincidencia, la computadora ejecuta la instrucción o bloque de instrucciones que se asocian con la constante. La forma general de la instrucción switch es:

```
switch (variable) {
case constante1:
    secuencia de acciones
break;
case constante2:
    secuencia de acciones
break;
case constante3:
    secuencia de acciones
break;
...
default:
    secuencia de acciones
}
```

donde la computadora ejecuta la secuencia default si no coincide ninguna constante con la variable, esta última es opcional. Cuando se encuentra una coincidencia, se ejecutan las acciones asociadas con el case hasta encontrar la sentencia break con lo que sale de la estructura switch.

Las limitaciones que tiene la sentencia switch ... case respecto a la estructura if son:

- Sólo se tiene posibilidad de revisar una sola variable.
- Con switch sólo se puede comprobar por igualdad, mientras que con if puede ser con cualquier operador relacional.

- No se puede probar más de una constante por case. Esto en términos estándar.

Un ejemplo que combina el switch/case con el do ... while es una rutina de selección en un menú como se muestra a continuación:

```
main()
{
    int opc;

    printf("1. Derivadas\n");
    printf("2. Limites\n");
    printf("3. Integrales\n");

    do
    {
        printf("   Teclear una opcion: ");
        scanf("%d", &opc);

        switch(opc)
        {
            case 1:
                printf("\tOpcion 1 seleccionada\n\n");
                break;
            case 2:
                printf("\tOpcion 2 seleccionada\n\n");
                break;
            case 3:
                printf("\tOpcion 3 seleccionada\n\n");
                break;
            default:
                printf("\tOpcion no disponible\n\n");
                break;
        }
    } while( opc != 1 && opc != 2 && opc != 3);
}
```

Aquí otro ejemplo:

```
# include <stdio.h>
main()
{
    int pato;
    for (pato = 3 ; pato < 13 ; pato = pato + 1)
    {
        switch (pato)
        {
            case 3 : printf("pato vale tres\n"); break;
            case 4 : printf("pato vale cuatro\n"); break;
```

```

        case 5 :
        case 6 :
        case 7 :
        case 8 :
        printf("El valor de pato está entre 5 y 8\n");
        break;
        case 12 :
        printf("pato vale doce\n");
        break;
        default : printf("Valor indefinido en una instrucción"
            "case\n"); break;
    } /* Fin de la instrucción switch */
} /* Fin del ciclo */
return 0;
}

```

La mejor manera de entender el funcionamiento de la instrucción **switch** –y de muchas otras también- es realizando el seguimiento del programa. En este caso, cuando la variable **pato** vale 3 la instrucción **switch** hace que el control del programa vaya directamente a la línea **case 3 : printf("pato vale tres\n"); break;** donde **printf ()** vuelca en pantalla el texto "pato vale tres" y el enunciado **break** hace salir al control del programa fuera del ciclo de instrucciones de **switch**.

Cuando el valor de la variable **pato** está especificado en una instrucción **case** dentro del ciclo de instrucciones de **switch**, los enunciados del programa serán ejecutados en orden hasta encontrar una instrucción **break**. Cuando **pato** vale 5, como no hay ninguna instrucción para ejecutar, el programa continúa hasta encontrar una instrucción ejecutable, que en el ejemplo es el **printf ()** que está a continuación de la línea de case 8 como sigue:

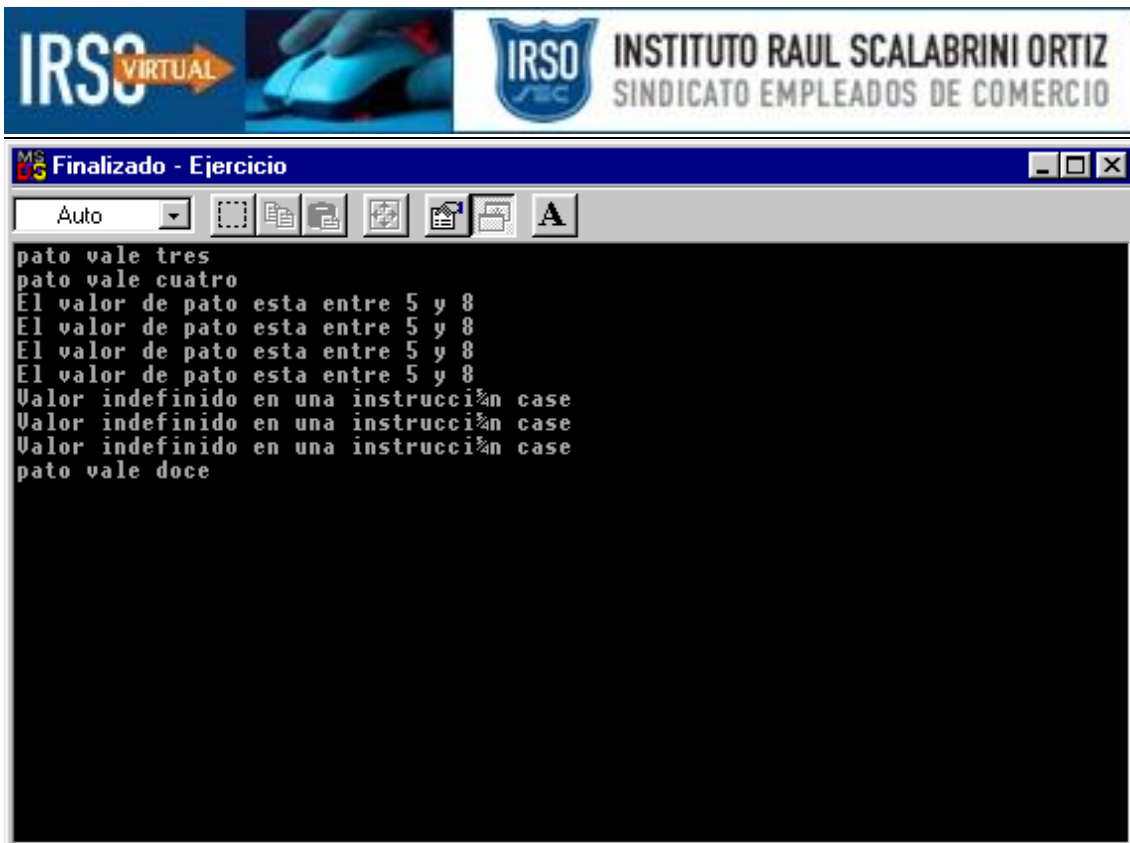
```

case 5 :
case 6 :
case 7 :
case 8 :
printf("El valor de pato está entre 5 y 8\n");

```

Si el valor de **pato** no está asociado con una instrucción **case**, se ejecuta el enunciado especificado en la instrucción **default**.

Una vez ejecutada la corrida del programa, la pantalla mostrará lo siguiente:



El switch/case es un muy buen recurso para reemplazar secuencias de instrucciones con varios if. Pero no siempre que utilizamos if podemos reemplazarlos “eficientemente” por instrucciones switch/case.

Uso de break y continue

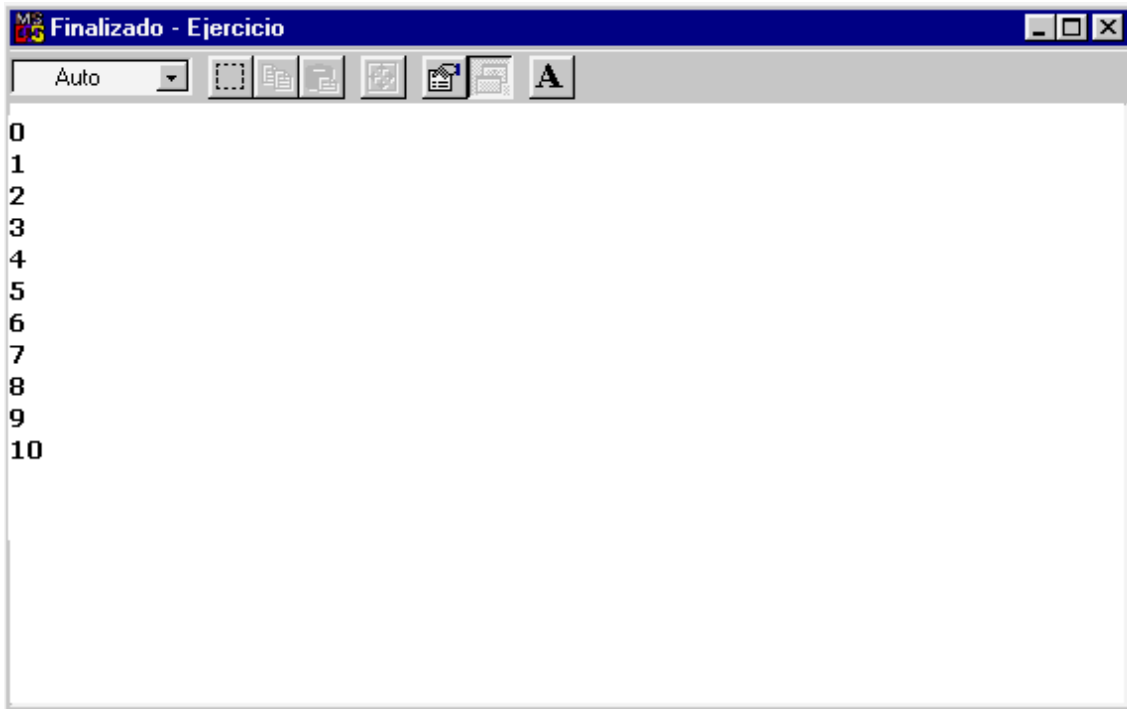
Como se comentó uno de los usos de la instrucción break es terminar un case en la sentencia switch. Otro uso es forzar la terminación inmediata de un ciclo, saltando la prueba condicional del bucle en el que esté incluido el break.

Cuando se encuentra la sentencia break en un bucle, se termina inmediatamente el ciclo y el control del programa pasa a la siguiente acción del ciclo. Por ejemplo:

```
main()
{
    int t;
    for(t=0; t<100; t++)
    {
```

```

        printf("%d ", t);
        if (t==10) break;
    }
}
    
```



Este programa muestra en pantalla los números del 0 al 10, cuando alcanza el valor 10 se cumple la condición del if, se ejecuta el break y sale del ciclo.

Por otro lado, continue funciona de manera similar al break. Sin embargo, en vez de forzar la salida, continue fuerza la siguiente iteración, por lo que salta el código que falta para llegar a probar la condición. Por ejemplo, el siguiente programa visualizará sólo los números pares:

```

main()
{
    int x;

    for( x=0; x<20; x++)
    {
        if (x%2)
            continue;
        printf("%d ",x);
    }
}
    
```



En el siguiente ejemplo, se leen valores enteros y se procesan de acuerdo a las siguientes condiciones: Si el valor que se lee es negativo, se desea imprimir un mensaje de error y se abandona el ciclo. Si el valor es mayor a 100, se ignora y se continúa leyendo, y si el valor es cero, se desea terminar el ciclo.

```
main()
{
    int valor;

    while( scanf("%d", &valor) == 1 && valor != 0)
    {
        if ( valor<0 )
        {
            printf("Valor no valido\n");
            break;
            /* Salir del ciclo */
        }

        if ( valor>100)
        {
            printf("Valor no valido\n");
            continue;
            /* Pasar al principio del ciclo nuevamente */
        }

        printf("Se garantiza que el valor leído está entre 1 y 100");
    }
}
```

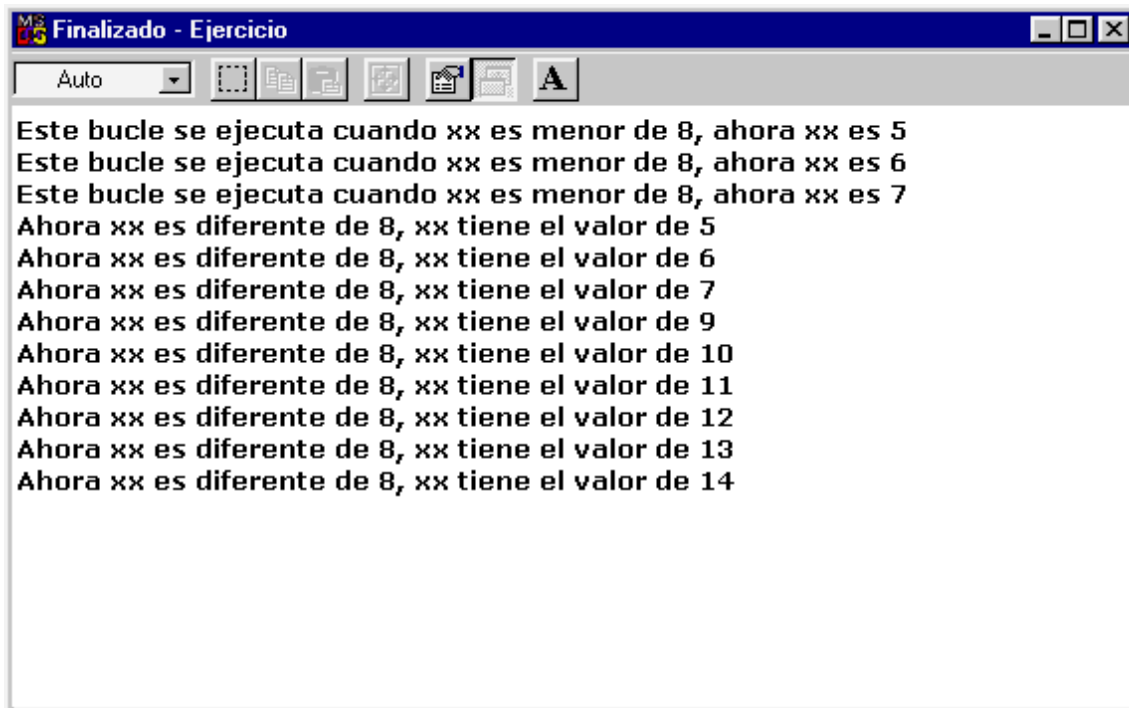


```
}
}
```

Ahora otro ejemplo combinando ambos conceptos:

```
#include <stdio.h>
main()
{
    int xx;

    for(xx = 5 ; xx < 15 ; xx = xx + 1)
    {
        if(xx == 8)
            break;
        printf("Este bucle se ejecuta cuando xx es menor de 8,"
            "ahora xx es %d\n", xx);
    }
    for(xx = 5 ; xx < 15 ; xx = xx + 1)
    {
        if(xx == 8)
            continue;
        printf("Ahora xx es diferente de 8, xx tiene el valor de %d\n", xx);
    }
    return 0;
}
```



Si vemos el código del programa, tenemos que en el primer bucle **for**, existe una instrucción **if** que llama a un **break** si **xx** es igual a 8. La instrucción **break**

lleva al programa a salir del bucle que se estaba ejecutando para continuar con las instrucciones siguientes inmediatas al bucle, terminando este en forma efectiva. Se trata de una instrucción muy útil cuando se desea salir del bucle dependiendo de los resultados (se evalúan valores de variables para tomar decisiones) que se obtengan dentro del bucle. En este caso, cuando **xx** alcanza el valor 8, el bucle termina imprimiendo el último valor válido, 7. La instrucción **break** salta inmediatamente después de la llave que cierra el bucle. En el siguiente bucle **for** hay una instrucción **continue** la que no finaliza el bucle pero suspende la ejecución del presente ciclo y continúa con el siguiente. Cuando el valor de **xx** alcanza como en este caso, el valor 8, el control del programa saltará al final del ciclo para continuar la ejecución del mismo, haciendo que la instrucción **printf ()** no se ejecute cuando en el bucle **xx** alcanza el valor de 8. El enunciado **continue** siempre salta al final del bucle, justo antes de la llave que indica el fin del bucle.



“Formas en la programación”

2.4. ESTRUCTURAS DE DATOS

INTRODUCCIÓN:

Espero que hayan comprendido los conceptos fundamentales tratados en los puntos anteriores como **qué es un algoritmo, cómo se estructura un programa en C, cuáles son las instrucciones más importantes**

Si alguna de estas preguntas no tiene respuesta en este momento, por favor releer los puntos correspondientes nuevamente de modo de fijar concretamente estos conceptos porque sin ellos, es imposible continuar con el desarrollo de la presente unidad.

Si todo fue comprendido, continuamos.

Aquí vamos a tratar de entender básicamente cuáles son los tipos de datos que permite manejar este lenguaje, la forma de definirlos y el rango que cada uno de ellos abarca. Para esto vamos a presentarlos uno por uno y, luego hacer

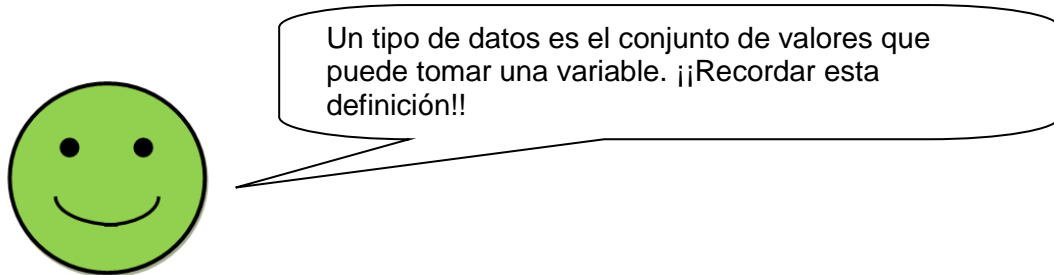
unas ejercitaciones de modo que se internalicen sus diferencias, para utilizarlos correctamente en la generación de programas.

Vayamos ahora al desarrollo de este tema tan importante recordando cuál es uno de los objetivos de la presente unidad: *Identificar los tipos de datos básicos que maneja C, tanto aquellos que vienen predefinidos como los que define en función de su necesidad el programador.*

DEFINICIÓN DE TIPO DE DATOS:

Un tipo de datos, **es el conjunto de valores que puede tomar una variable.**

Me gustaría ahora que nos concentremos un poco en esta definición de modo de ampliarla y comentarla.



Como hemos dicho en la unidad anterior, dentro de la estructura de nuestro programa hay diferentes elementos que coexisten de modo de satisfacer nuestra necesidad básica de procesamiento; sólo podremos procesar si tenemos qué procesar. Eso que vamos a procesar no son ni más ni menos que datos que seguramente manejaremos con una variable que los aloja. Esos datos podrán ser, por ejemplo: enteros, reales, caracteres, etc. *Cada uno de estos posibles conjuntos de valores que puede tomar una variable es lo que nosotros denominamos tipo.* Como vemos algunos de estos conjuntos tienen una cantidad limitada de posibles valores, tal es el caso de los caracteres, de los que sólo existen 256 valores diferentes. Otros tienen una cantidad “casi ilimitada” como por ejemplo los reales. Las comillas en este caso indican algo que por más lógico que sea vale la pena aclarar: el conjunto de los reales en una computadora no es un conjunto infinito de datos como lo es en la

matemática. Aquí las limitaciones físicas en cuanto a hardware se refiere, hacen imposible que la cantidad de datos con los que se trabaje sea infinito. De este modo siempre que trabajamos con un sistema estamos trabajando con una cantidad finita de datos aunque en la mayoría de las aplicaciones es muchísimo más que suficiente en función de los problemas que se nos plantean.

Existe una clasificación que permite identificar dos formas diferentes en las que se presentan los tipos:

Tipos de datos predefinidos por C

Tipos de datos definidos por el programador

TIPOS PREDEFINIDOS

¿Qué son los tipos predefinidos?

Son aquellos que son propios del lenguaje y ya vienen definidos implícitamente en el mismo.

¿Cuáles son los tipos predefinidos?

Estos tipos de datos son principalmente los que se enumeran a continuación:

| Tipo | Tamaño (bytes) | Límite inferior | Límite superior |
|--------------------|-----------------------|--|--|
| char | 1 | -- | -- |
| unsigned char | 1 | <u>0</u> | <u>255</u> |
| short int | 2 | <u>-32768</u> | <u>+32767</u> |
| unsigned short int | 2 | <u>0</u> | <u>65536</u> |
| (long) int | 4 | <u>-2^{31}</u> | <u>$+2^{31} - 1$</u> |
| float | 4 | <u>$-3.2 \times 10^{\pm 38}$</u> | <u>$+3.2 \times 10^{\pm 38}$</u> |
| double | 8 | <u>$-1.7 \times 10^{\pm 308}$</u> | <u>$+1.7 \times 10^{\pm 308}$</u> |

Los tipos de datos básicos tienen varios *modificadores* que les preceden. Se usa un modificador para alterar el significado de un tipo base para que encaje con las diversas necesidades o situaciones. Los modificadores son: signed, unsigned, long y short.

Normalmente, los tipos int son long int, a menos que se especifique explícitamente short int.

Nota: no hay un tipo booleano en C. Se deberá usar char, int o aún mejor unsigned char.

signed, unsigned, long y short pueden ser usados con los tipos char e int.

Aunque es permitido el uso de signed en enteros, es redundante porque la declaración de entero por defecto asume un número con signo.

Para declarar una variable en C, como ya hemos indicado en su momento, se debe seguir el siguiente formato:

tipo lista_variables;

tipo es un tipo válido de C y *lista_variables* puede consistir en uno o más indentificadores separados por una coma que denotan las variables. Un identificador debe comenzar con una letra o un guión bajo.

Ejemplo:

```
int i, j, k;
```

```
float x,y,z;
```

```
char ch;
```

Existen muchos otros tipos de datos definidos por C que iremos ampliando con el correr de este curso, pero para profundizar necesitamos conocer muchas estructuras de datos que aún no manejamos.

Se puede cambiar el tipo de una expresión de esta forma:

(nuevo_tipo) expresión

Por ejemplo, para forzar a que una división de enteros se realice en coma flotante, podemos escribir:

```
int x=5,y=2;
```

```
float f;
```

```
f = (float)x/y;
```

En este ejemplo, el valor de x, que es entero, se transforma a **float**. Así la división se realizará en coma flotante. El resultado que quedará guardado en f luego de la división será 2.5

Otros ejemplos:

A.

```
float a = 5.25;
int b;
```

```
main()
{
b = (int)a; /*En este caso, la variable b queda cargada con el valor 5*/
return 0;
}
```

B.

```
int x=7, y=5 ;
float z;
```

```
main()
{
z=x/y; /*En muchos compiladores, z queda cargada con el valor 1*/
return 0;
}
```

Por eso, es necesario hacer un cast si se desea obtener el resultado de tipo float. Lo que lleva a generar un código que sea:

```
int x=7, y=5;
float z;
```

```
main()
{
z = (float)x/(float)y; /*En este caso, z queda cargada con el valor 1.4*/
return 0;
}
```

TIPOS DEFINIDOS POR EL PROGRAMADOR

¿Qué son los tipos definidos por el programador?

Son aquellos generados por el programador en función de las necesidades que pueda tener en un programa determinado.

Uno de los tipos definidos por el programador es la enumeración.

Una *enumeración* es un conjunto de constantes enteras con nombre y especifica todos los valores legales que puede tener una variable del tipo *enum*.

La forma como se define una enumeración es con la palabra clave `enum` para el comienzo del tipo. Su formato es:

`enum nombre_enum { lista_de_enumeración } lista_de_variables;`

Es opcional **nombre_enum** y **lista_de_variables**. La primera se usa para declarar las variables de su tipo. El siguiente fragmento define una enumeración llamada `disco` que declara almacenamiento para ser de ese tipo.

```
enum almacenamiento { diskette, dd, cd, dvd, cinta };
```

```
enum almacenamiento disco;
```

Con la siguiente definición y declaración son válidas las siguientes sentencias:

```
disco = cd;
```

```
if ( disco == diskette )
    printf("Es de 1440 Kb\n");
```

Se inicializa el primer símbolo de enumeración a cero, el valor del segundo símbolo a 1 y así sucesivamente, a menos que se inicialice de otra manera. Por tanto,

```
printf("%d %d\n", dd, cinta)
```

muestra 1 4 en la pantalla.

Se puede especificar el valor de uno o más símbolos usando un inicializador.

Para hacerlo, poner un signo igual y un valor entero después del símbolo.

Por ejemplo, lo siguiente asigna el valor 250 a `cd`


```
enum disco { diskette, duro, cd=250, dvd, cinta };
```

Por lo tanto los valores de los símbolos son los siguientes:

| | |
|-----------------|-----|
| diskette | 0 |
| duro | 1 |
| cd | 250 |
| dvd | 251 |
| cinta | 252 |

Entonces, cuando defino un **tipo por enumeración**, debo identificar unívocamente cada uno de los valores que puede tomar una variable de ese tipo. Esto es, debo enunciar cada uno de los posibles valores utilizando comas para separarlos y paréntesis para indicar dónde comienza y dónde termina el conjunto de datos enumerados.

Evidentemente este tipo de declaración es sumamente precisa pero muy engorrosa al momento de generarla.

TIPO STRING (Cadena de caracteres)

En C no existe un tipo predefinido para manipular cadenas de caracteres (*strings*). Sin embargo, el estándar de C define algunas funciones de biblioteca para tratamiento de cadenas.

La forma de declarar una cadena en C es mediante lo que se conoce como un vector de caracteres:

```
char hola [5];
```

Toda cadena ha de terminar con el carácter especial 0 (cero).

El lenguaje C no tiene otra manera de detectar el final de una cadena.

Los literales tipo cadena son de la forma

```
"texto entre comillas"
```

Al declarar una vector de caracteres, se le puede inicializar con un literal:

```
char texto [4] = "abc";
```

Pero NO se puede hacer una asignación de ese tipo en una sentencia:

```
texto = "xyz";    /* ERROR */
```

En su lugar, hay que emplear ciertas funciones de biblioteca.

Tres formas equivalentes de inicializar una misma cadena son:

```
char hola [5] = { 'h', 'o', 'l', 'a', 0 };
```

```
char hola [5] = "hola";
```

```
main()
{
    char hola [5];
    hola[0] = 'h';
    hola[1] = 'o';
    hola[2] = 'l';
    hola[3] = 'a';
    hola[4] = 0;
}
```

Obsérvese que una cadena de N elementos es capaz de almacenar un texto de $N-1$ caracteres (el último siempre ha de ser un cero).



Presten atención a cómo inicializar cadenas y también a cuál es su longitud y su forma.

No importa que un vector de caracteres contenga una cadena de menos letras, el carácter cero marca el final de la cadena.

Lo que sí es un error (y además no lo detecta siempre el compilador) es intentar asignarle a un vector de caracteres una cadena de mayor tamaño.

Hay que cuidar mucho que las cadenas queden dentro del espacio reservado para ellas sino quedan truncadas y parte de la información que contienen las mismas se pierde.

La biblioteca **<string.h>** contiene un conjunto de funciones para manipular cadenas: copiar, cambiar caracteres, comparar cadenas, etc. Por eso, cada vez que utilicemos los strings en algún programa, debemos incluir esta librería en el encabezado.

Las funciones más elementales son:

strcpy (c1, c2); Copia **c2** en **c1**

strcat (c1, c2); Añade **c2** al final de **c1** (concatena)

int strlen (cadena); Devuelve la longitud de la **cadena**

int strcmp (c1, c2); Devuelve cero si **c1** es igual a **c2**; no-cero en caso contrario

Para trabajar con estas funciones, al comienzo del programa hay que escribir

```
#include <string.h>
```

y por eso, es recomendable incluir esta librería cada vez que trabajemos con strings.

Ejemplo:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
char completo [80];
```

```
char nombre[32] = "Pedro";
```

```
char apellidos [32] = "González Arenas";
```

```
main()
```

```
{
```

```
/* Construye el nombre completo */
```

```
strcpy ( completo, nombre );      /* completo <- "Pedro" */
```

```
strcat ( completo, " ");          /* completo <- "Pedro " */
```

```
strcat ( completo, apellidos );   /* completo <- "Pedro  
González Arenas" */
```

```
printf ( "El nombre completo es %s\n", completo );
```

```
}
```

Con lo cual, cuando enviemos a imprimir a pantalla la variable completo, lo que la misma mostrará es:



En teoría, podría utilizarse la opción **%s** de **scanf**, pero depende del tipo de compilador que se use, porque no funciona en todos.

Una mejor alternativa es emplear **gets**, que también viene en **stdio.h**

```
#include <stdio.h>
```

```
main ()
```

```
{
```

```
    char nombre [80];
```

```
    printf ( "¿Cuál es su nombre? " );
```

```
    gets ( nombre );
```

```
    printf ( "Parece que su nombre es %s\n", nombre );
```

```
}
```

Esta función **gets()**, lee caracteres desde el teclado, hasta que se encuentra un final de archivo (EOF) que veremos más adelante o hasta que se lee un carácter de nueva línea. El carácter de nueva línea que se lee es descartado, y un carácter nulo es escrito inmediatamente después del último carácter leído en la cadena para respetar la forma de los strings.

NOTA: **gets** no comprueba el tamaño de la cadena. Si el texto tecleado tuviera más de 80 caracteres, se destruirían posiciones de memoria incorrectas. Por eso es importante un buen control de flujo de datos al momento de la carga de los mismos desde teclado en la variable.

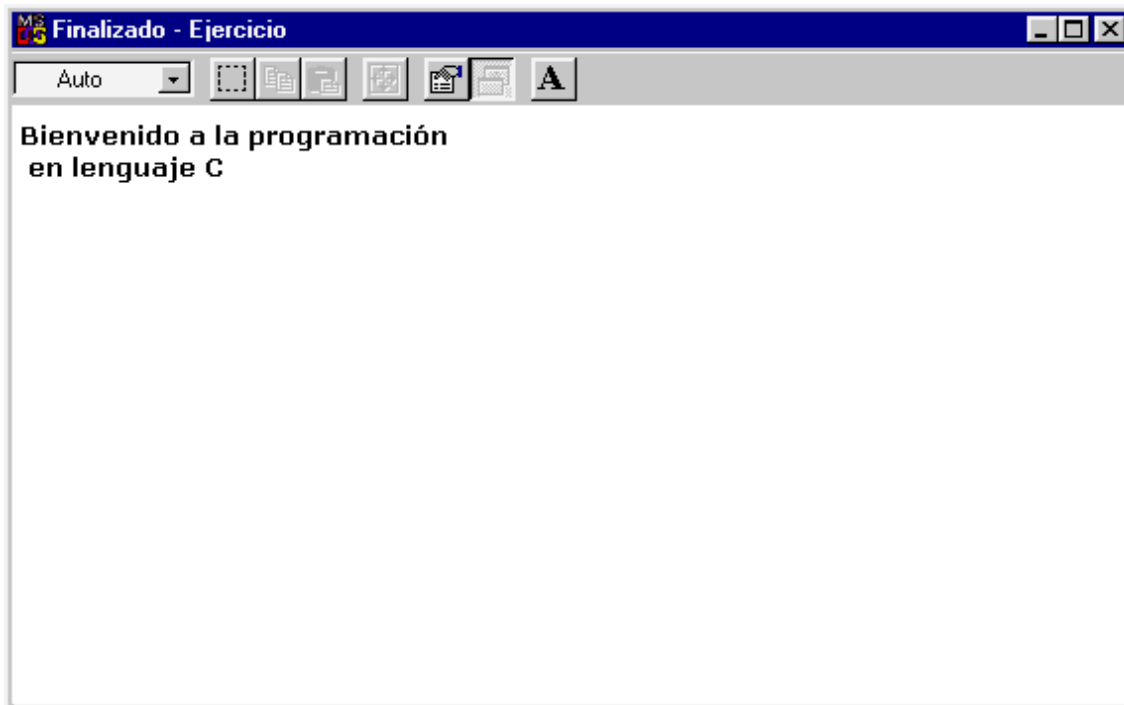
La función **gets()** se usa normalmente en compañía de la función **puts()**.

La función **puts()** simplemente imprime una cadena de caracteres en la salida estándar (y produce un salto de línea luego de la impresión que realiza). Le debemos proporcionar cuál es la cadena de caracteres que se quiere enviar a imprimir. Un código ejemplo puede ser:

```
#include <stdio.h>
```

```
main()
{
    puts("Bienvenido a la programación");
    puts(" en lenguaje C");
}
```

Lo que generará una salida en pantalla que será:



Como dijimos, la función **gets()** simplemente toma una cadena de caracteres de la entrada estándar (cuyo ingreso es preciso terminar con un ENTER) y la almacena en una variable string (cadena de caracteres). Supongamos este ejemplo de código:

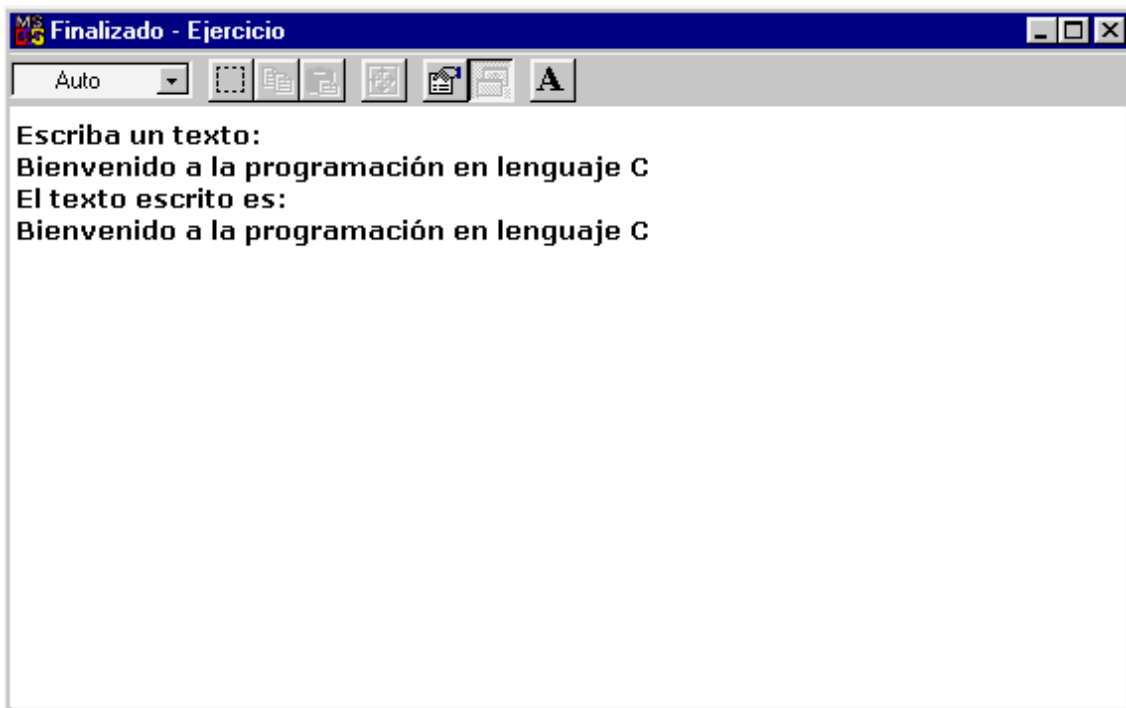
```
#include <stdio.h>
```

```
main()
{
    char cadena[50];
```

```

puts("Escriba un texto:");
gets(cadena);
puts("El texto escrito es:");
puts(cadena);
}
    
```

Lo que hará que se lea una cadena de teclado y que luego se imprima. La declaración `char cadena[50];` crea una variable llamada `cadena` que puede almacenar hasta 50 caracteres. Este código produce, cuando escribimos con el teclado el texto “Bienvenido a la programación en lenguaje C”. La forma en la que se presentará la pantalla una vez terminado de ejecutar todo el programa será:



Donde la segunda línea es la leída por teclado gracias al ingreso de datos que realiza el usuario, y la cuarta es la impresión de esa cadena completa leída.

Para leer caracteres hasta un límite máximo de posiciones, hay que usar **fgets**:

fgets (nombre, n, cadena_fuente);

Esta función lee como máximo uno menos que el número de caracteres indicado por **n** desde la fuente indicada **cadena_fuente** (por ejemplo el teclado (`stdin`), un archivo (`file`) o de otra cadena de caracteres (`string`)). Ningún carácter adicional es leído después del carácter de nueva línea (el cual es retenido) o después de un final de archivo (EOF). Un carácter nulo es escrito

inmediatamente después del último carácter leído. El resultado de la función se guarda en la variable cadena de caracteres **nombre**.

Por ejemplo, `fgets (nombre, 80, stdin);` lee de teclado (`stdin`) hasta 79 caracteres válidos a los que le agrega el 0(cero) como último carácter y lo guarda en una variable string llamada `nombre`.

Esta función tiene su compañera llamada **`fputs(nombre, cadena_destino);`**, que escribe la cadena `nombre` en `cadena_destino` (por ejemplo la pantalla (`stdout`), un archivo (`file`) u otra cadena de caracteres (`string`)). El carácter nulo que tiene la cadena en su última posición no es escrito.



Si ahora pensamos solamente en el manejo de caracteres, existen dos funciones equivalentes a las de cadenas pero para un solo carácter, se llaman: **`putchar()`** y **`getchar()`**.

La función **`putchar()`** escribe un único carácter en la salida estándar. Su uso es sencillo y generalmente está implementada como una macro en la cabecera de la biblioteca estándar. Por ejemplo, veamos el siguiente código:

```
#include <stdio.h>
```

```
main()
{
    putchar('H');
    putchar('o');
    putchar('l');
    putchar('a');

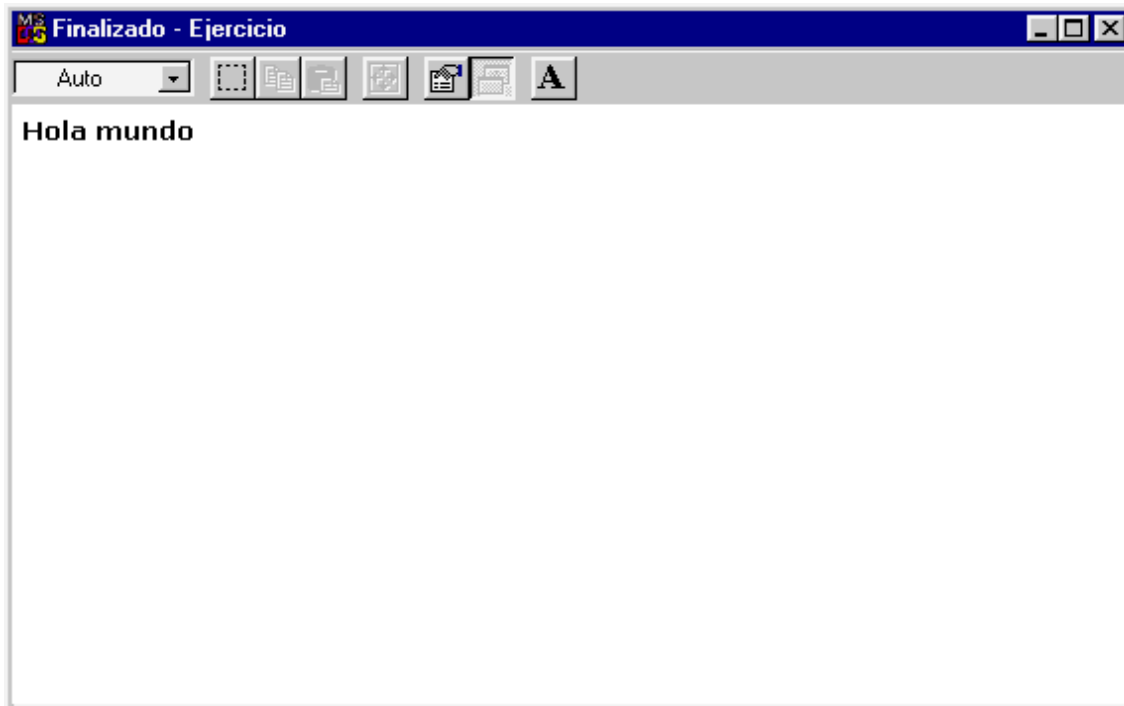
    putchar(32);

    putchar('m');
    putchar('u');
    putchar('n');
    putchar('d');
    putchar('o');

    putchar('\n');
}
```


En el código anterior **putchar(32);** muestra el espacio entre ambas palabras (32 es justamente el código ASCII del carácter espacio ' ') y **putchar('\n');** imprime un salto de línea tras el texto.

Lo que muestra la pantalla una vez ejecutado el código del programa es lo siguiente:



Por otro lado, la función **getchar()** devuelve el carácter que se halle en la entrada estándar. Esta función tiene dos particularidades: La primera es que aunque se utiliza para obtener caracteres no devuelve un carácter, sino un entero. Esto se hace así ya que con un entero podemos representar tanto el conjunto de caracteres que cabe en el tipo carácter (normalmente el conjunto ASCII de caracteres) como el carácter EOF de fin de archivo que veremos más adelante. Es habitual representar los caracteres usando el código ASCII. Estos caracteres se suelen representar como un entero que va del 0 al 256. Un ejemplo de código para el **getchar()**:

```
#include <stdio.h>
```

```
main()
{
```

```
    int c;
```

```
    c = getchar();    /* Nótese que getchar() no devuelve nada
                       hasta que se presiona ENTER */
```

```
    putchar(c);
```

}

Aquí se almacena en la variable de tipo entero **c** el carácter pulsado en el teclado que posteriormente se muestra con **putchar(c)**.

Posicionado del cursor y borrado de pantalla

Para el posicionado del cursor usamos la instrucción **gotoxy()**. Esta función sólo está disponible en compiladores de C que dispongan de la biblioteca `<conio.h>`.

Hemos visto que cuando usamos **printf()** se escribe en la posición actual del cursor y se mueve éste al final de la cadena que hemos escrito. Pero ¿qué sucede cuando queremos escribir en una posición determinada de la pantalla? La solución está en la función **gotoxy()**. Supongamos que queremos escribir 'Hola' en la fila 10, columna 20 de la pantalla, el código que permitirá realizar esto es:

```
#include <stdio.h>
#include <conio.h>

main()
{
    gotoxy( 20, 10 );
    printf( "Hola" );
}
```

Obsérvese que primero se indica la columna (x) y luego la fila (y). La esquina superior izquierda es la posición (1, 1).

Luego de la corrida del programa, la pantalla de salida mostrará:



Con la palabra “Hola” ubicada en la columna 20 y fila 10.

Vamos ahora al borrado o limpieza de la pantalla, para se utiliza la función `clrscr()` (clear screen, borrar pantalla). Esta función no sólo borra la pantalla, sino que sitúa el cursor en la posición (1, 1), en la esquina superior izquierda. Veamos un ejemplo de código:

```
#include <stdio.h>
#include <conio.h>

main()
{
    clrscr();
    printf( "Hola" );
}
```

Luego del **`clrscr()`** la pantalla queda limpia (vacía) y con el **`printf("Hola");`** se imprime la palabra “Hola” en la posición (1,1) de la misma. La pantalla, una vez ejecutado el programa quedará como se muestra a continuación:



Recordar siempre que el uso de la función `clrscr()` vale para los compiladores que permitan incluir a la librería de funciones `conio.h`

2.5. DECLARACIÓN DE CONSTANTES (CONST):

Una declaración de constantes (`const`) define un identificador que denota un valor no modificable en el tiempo.

La palabra clave `const` se usa para declarar una constante, como se muestra a continuación:

```
const a = 1;
```

```
int a = 2;
```

Notas:

- Se puede usar `const` antes o después del tipo.
- Es usual inicializar una constante con un valor, ya que no puede ser cambiada *de alguna otra forma*.

La directiva `#define` es un método más flexible para definir *constantes* en un programa.

Frecuentemente se ve la declaración const en los parámetros de la función. Lo anterior simplemente indica que la función no cambiará el valor del parámetro.

Por ejemplo, la siguiente función usa este concepto:

```
char *strcpy(char *dest, const char *orig);
```

El segundo argumento orig es una cadena de C que no será alterada, cuando se use la función de la biblioteca para copiar cadenas.

Siempre la declaración se realiza partiendo de un nombre definido por el programador y luego, el valor que la constante toma.

Las constantes no pueden ser modificadas (son invariables) a lo largo de la corrida del programa salvo algunas excepciones ya mencionadas y otras que iremos manejando más adelante.



La declaración de una constante se realiza utilizando la forma:

```
const <nombre> = valor;
```



Releer cómo se realiza la declaración de constantes.

2.6. OPERADORES ARITMÉTICOS

Al igual que en otros lenguajes de programación, en C se tienen los operadores aritméticos más usuales (+ suma, - resta, * multiplicación, / división y % módulo).

El operador de asignación es = como ya lo hemos venido trabajando, por ejemplo: `i=4; ch='y';`

Incremento ++ y decremento -- unario. Es unario porque suma uno o resta uno a la variable considerada. Son más eficientes que las respectivas asignaciones. Por ejemplo: `x++` es más rápido que `x=x+1`.

Los operadores ++ y -- pueden ser prefijos o postfijos. Cuando son prefijos, el valor es calculado antes de que la expresión sea evaluada, y cuando es postfijo el valor es calculado después que la expresión es evaluada.

En el siguiente ejemplo, ++z es prefijo y -- es postfijo:

```
int x,y,z;
```

```
main()
{
    x=( ( ++z ) - ( y-- ) ) % 100;
}
```

Es equivalente a:

```
int x,y,z;
```

```
main()
{
    z++;
    x = ( z-y ) % 100;
    y--;
}
```

El operador % (módulo o residuo) solamente trabaja con enteros, aunque existe una función para flotantes (fmod()) de la biblioteca matemática.

El operador división / es para división entera y flotantes. Por lo tanto hay que tener cuidado. El resultado de $x = 3 / 2$; es uno, aún si x es declarado como float. La regla es: si ambos argumentos en una división son enteros, entonces el resultado es entero. Si se desea obtener la división con la fracción, entonces escribirlo como: $x = 3.0 / 2$; o $x = 3 / 2.0$ y aún mejor $x = 3.0 / 2.0$.

Por otra parte, existe una forma más corta para expresar cálculos en C. Por ejemplo, si se tienen expresiones como: $i = i + 3$; o $x = x * (y + 2)$; , pueden ser reescritas como:

```
expr1 oper = expr2
```

Lo cual es equivalente, pero menos eficiente que:

$expr_1 = expr_1 \text{ oper } expr_2$

Por lo que podemos reescribir las expresiones anteriores como: $i += 3$; $y x *= y + 2$; respectivamente.

Operadores de Comparación

El operador para probar la igualdad es $==$, por lo que se deberá tener cuidado de no escribir accidentalmente sólo $=$, ya que:

`if (i = j) ...`

Es una sentencia legal de C (sintácticamente hablando aunque el compilador avisa cuando se emplea), la cual copia el valor de ```j``` en ```i```, lo cual será interpretado como VERDADERO.

Diferente es $!=$, otros operadores son: $<$ menor que, $>$ mayor que, $<=$ menor que o igual a y $>=$ (mayor que o igual a).

Lo correcto es entonces: `if (i == j) ...`

Operadores lógicos

Los operadores lógicos son usualmente usados con sentencias condicionales o relacionales, los operadores básicos lógicos son:

$\&\&$ Y lógico, $\|\|$ O lógico y $!$ negación.

Orden de precedencia

Es necesario ser cuidadosos con el significado de expresiones tales como $a + b * c$, dependiendo de lo que se desee hacer

$(a + b) * c$

o

$a + (b * c)$

Todos los operadores tienen una prioridad, los operadores de mayor prioridad son evaluados antes que los que tienen menor prioridad. Los operadores que tienen la misma prioridad son evaluados de izquierda a derecha, por lo que:

$a - b - c$

es evaluado como

(a - b) - c

| Prioridad | Operador(es) |
|------------------|-------------------------------|
| Más alta | () [] -> |
| | ! ~ ++ -- - (tipo) * & sizeof |
| | * / % |
| | + - |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | |
| | && |
| | |
| | ? |
| | = += -= *= /= |
| Más baja | , |



El orden de precedencia es importante para no escribir tantos paréntesis en las expresiones algebraicas.

2.7. VECTORES

Un vector es una estructura donde cada uno de los elementos que la componen son del mismo tipo de datos. Esta estructura tiene un tamaño fijo (es decir, una cantidad finita y determinada de elementos – esto es llamado muchas veces estructura estática --). Cada elemento puede ser identificado por su posición en la estructura. Esta posición es una posición relativa al primer elemento de la estructura. Si ejemplificamos la estructura con un gráfico, el mismo sería como se ve a continuación:

V1

| | |
|---|-----|
| 1 | 14 |
| 2 | 43 |
| 3 | -3 |
| 4 | 0 |
| 5 | 11 |
| 6 | -45 |
| 7 | 7 |

El vector es un nuevo tipo de datos que comenzaremos a utilizar desde ahora.

A los vectores también suele llamárselos arreglos.

El formato para declarar un arreglo unidimensional es:

tipo nombre_arr [**tamaño**]

Por ejemplo, para declarar un arreglo de enteros llamado *listanum* con diez elementos se hace de la siguiente forma:

```
int listanum[10];
```

donde, el nombre del tipo es elegido por el programador; el tipo_base es el tipo de los elementos del vector (en el vector que antecede el tipo de los elementos del vector es entero dado que los números que contiene el vector son todos int); mientras que el rango [0..9] son los índices del vector, estos son valores *discretos* que especifican la posición de los elementos en la estructura (para el caso del vector que nos antecede el rango de índices corresponde a los valores 0 a 9).



En C, todos los arreglos usan cero como índice para el primer elemento. Por lo tanto, el ejemplo anterior declara un arreglo de enteros con diez elementos desde **listanum[0]** hasta **listanum[9]**.

De esta forma hemos definido el vector.

Ahora nos interesa ver de qué manera podemos acceder a los valores que contiene el vector, esto es relativamente sencillo y se realiza de la siguiente forma: se coloca el nombre de la variable vector considerada y entre corchetes la posición a la cual se quiere acceder. Por ejemplo, en el vector `listanum[2] = 15;` /* Asigna 15 al 3er elemento del arreglo listanum */ , `num = listanum[2];` /* Asigna el contenido del 3er elemento a la variable num */.

2.8. MATRICES

Del mismo modo que hemos trabajado sobre vectores (teniendo sólo un índice para recorrer el mismo), es posible llevar este concepto a dos dimensiones (es decir, utilizar dos índices para recorrer una estructura que tiene ahora dos coordenadas para identificar una posición de la misma). La matriz es una estructura estática al igual que el vector (esto es, que no varía su tamaño durante la ejecución del programa –lo que si varía es el contenido de las

posiciones que tiene el vector o la matriz pero no varía la cantidad de elementos que ellos tienen). Es decir, que una vez que se define la cantidad de elementos que tiene una matriz, esa cantidad no varía durante el tiempo de corrida del programa. Los elementos de la matriz son todos del mismo tipo. Los índices son de un tipo discreto ordinal. Estas básicamente son las características de la matriz y la estructura física de la misma puede ser representada como se ve a continuación:

M1

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|----|----|----|----|----|----|
| 1 | 8 | 0 | -6 | 11 | 7 | -5 |
| 2 | 4 | 7 | 0 | -1 | -1 | 6 |
| 3 | 10 | 10 | 10 | 9 | 6 | 9 |
| 4 | 88 | 98 | 66 | 1 | 0 | -2 |

Esta matriz tiene 4 filas y 6 columnas.

Cuando tengamos que hacer una operación que involucre a todos los elementos de la matriz deberemos actuar incrementando dos índices (uno para las filas y otro para las columnas).

La definición del tipo matriz debe realizarse de la misma forma que el vector pero teniendo en cuenta que ahora debemos trabajar sobre dos índices en lugar de uno, en forma genérica, para un arreglo multidimensional, tenemos que:

tipo nombre_arr [**tam1**][**tam2**] ... [**tamN**];

Por ejemplo un arreglo de enteros bidimensionales (matriz) se escribirá como:
`int tabladenums[50][50];`

Observar que para declarar cada dimensión lleva sus propios paréntesis cuadrados.

Para acceder los elementos se procede de forma similar al ejemplo del arreglo unidimensional, esto es,

`tabladenums[2][3] = 15; /* Asigna 15 al elemento de la 3ª fila y la 4ª columna*/`
`num = tabladenums[25][16];`

A continuación se muestra un ejemplo que asigna al primer elemento de un arreglo bidimensional cero, al siguiente 1, y así sucesivamente.

```
main()
{
    int t,i,num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; i<4; ++i)
            num[t][i]=(t*4)+i*1;

    for(t=0; t<3; ++t)
    {
        for(i=0; i<4; ++i)
            printf("num[%d][%d]=%d ", t,i,num[t][i]);
        printf("\n");
    }
}
```

En C se permite la inicialización de arreglos, debiendo seguir el siguiente formato:

`tipo nombre_arr[tam1][tam2] ... [tamN] = {lista-valores};`

Por ejemplo:

```
int i[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
int num[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
```

Aquí se puede ver que el nombre es elegido por el programador. Los límites para los índices se definen utilizando los corchetes y definiendo el rango para cada uno de ellos (el rango para las filas y el de las columnas). El tipo_base es el tipo de los elementos de la matriz.



Cómo definir un tipo matriz:

```
tipo nombre_arr[ tam1 ][ tam2 ] ... [ tamN ] = {lista-valores};
```

Veamos un ejemplo sencillo de una matriz, en la que los elementos de la misma se cargarán simplemente multiplicando el valor del índice de la fila por el índice de la columna de la posición donde se cargará ese resultado.

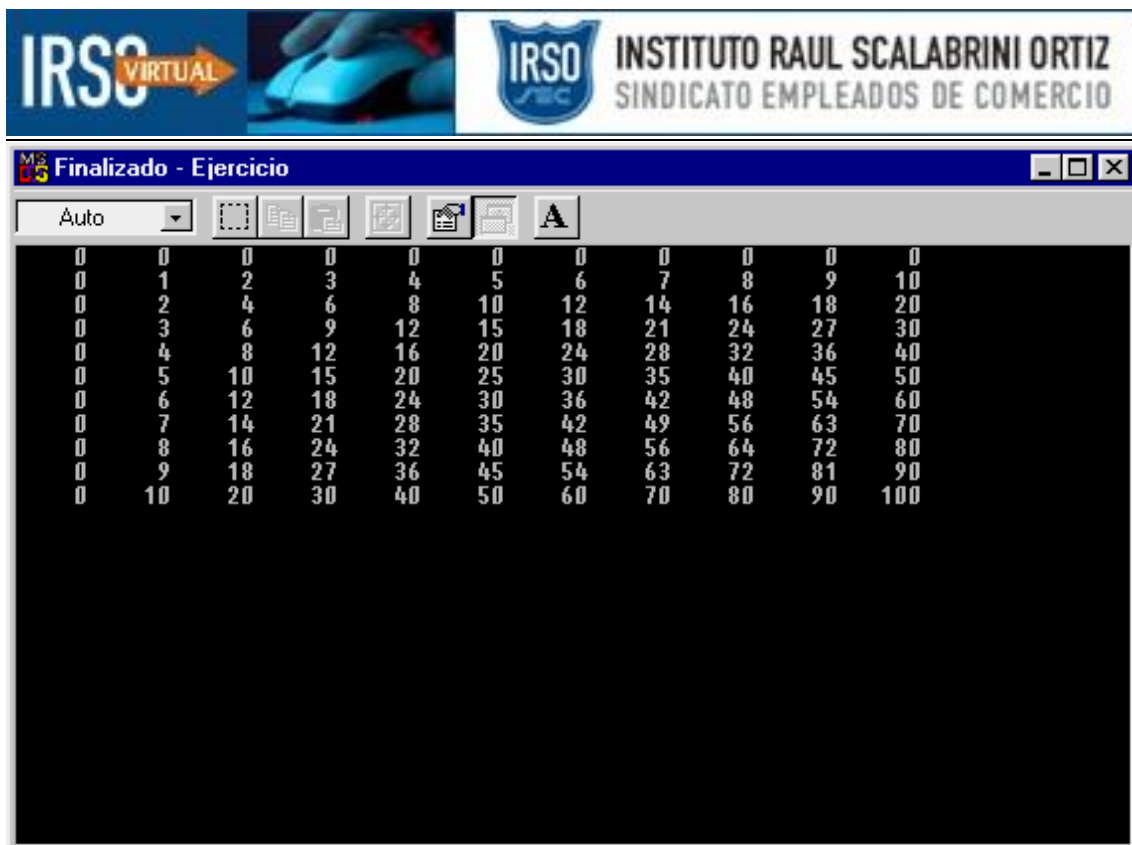
```
#include <stdio.h>
```

```
int main()
{
    int i, j;
    int multiplica[11][11];

    for (i = 0 ; i < 11 ; i++)
        for (j = 0 ; j < 11 ; j++)
            multiplica[i][j] = i * j;

    for (i = 0 ; i < 11 ; i++)
    {
        for (j = 0 ; j < 11 ; j++)
            printf("%5d ", multiplica[i][j]);
        printf("\n");
    }
    return 0;
}
```

Esto se mostrará en pantalla como sigue:



EJERCITACIÓN

Práctica 1: Vectores y matrices

Ejercicio 1: Realizar un programa que permita hacer nulos (cargarles valor 0) los elementos de un vector Q de enteros, con n componentes (donde n es una constante que vale 10).

Ejercicio 2: Realizar un programa que permita obtener e imprimir el resultado de la suma de los elementos de un vector Z de longitud k, donde los elementos del vector son reales (considerar k=5).

Ejercicio 3: Realizar un programa que permita imprimir las componentes de un vector A de longitud g, cuyos elementos son caracteres. La impresión deberá indicar el índice del vector y el valor de la componente. Los índices son enteros y g=8.

Ejercicio 4: Realizar un programa que permita obtener el producto de dos vectores A y B componente a componente, guardando el resultado en un nuevo

vector C. Los vectores tienen longitud 6, y los índices son enteros. Los elementos del vector también son enteros. Imprimir el resultado.

Ejercicio 5: Realizar un programa que permita asignar la identidad a una matriz R de dimensión $m \times m$ (considerar $m=4$). Los índices y las componentes son enteros. La matriz identidad es la que tiene valor 1 en la diagonal principal y 0 en el resto.

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Ejercicio 6: Realizar un programa que permita obtener la suma de dos matrices X y J de dimensión $a \times b$ ($a=3$ y $b=4$). Los elementos de la matriz son reales. El resultado debe ser impreso. Realizar la suma componente a componente.

Nota: Los valores de las matrices y los vectores deben ser cargados desde teclado y siempre impresos sobre la pantalla.

UNIDAD 3: ESTRUCTURAS DE DATOS COMPUESTAS

¿Cuáles son nuestros objetivos para esta unidad?

La idea en esta segunda parte es conocer el alcance que tienen las estructuras de datos compuestas, su uso y cuáles son los beneficios que las mismas nos brindan. Además se trabajará utilizando subprogramas, lo que permitirá modularizar las soluciones dotándolas de flexibilidad. Nuestros objetivos son, entonces:

Generar estructuras básicas para el almacenaje de la información.

Lograr modularizar un programa en C utilizando funciones y procedimientos.

3.1. ESTRUCTURAS O REGISTROS

¿Qué es un registro?

Un registro es una variable compuesta. Está formada por componentes de igual o distinto tipo de datos llamados campos. La definición de un tipo registro o estructura contempla cada uno de los campos que la forman (utilizando un nombre) y su tipo.

r

| | | | | | |
|--------|---------|-----------|---------|---------|--------|
| legajo | nomalum | dir_calle | dir_num | dir_loc | dir_cp |
|--------|---------|-----------|---------|---------|--------|

Esta es la variable r de tipo registro. Como vemos muy claramente en este gráfico, la misma está formada por seis campos.

En C, una estructura es una colección de variables que se referencian bajo el mismo nombre. Una estructura proporciona un medio conveniente para mantener junta información que se relaciona. Una *definición de estructura* forma una plantilla que se puede usar para crear variables de estructura. Las variables que forman la estructura son llamados *elementos estructurados*.

Generalmente, todos los elementos en la estructura están relacionados lógicamente unos con otros. Por ejemplo, se puede representar una lista de nombres de correo en una estructura. Mediante la palabra clave struct se le indica al compilador que defina una plantilla de estructura.

```
struct direc
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
};
```

Con el código anterior *no ha sido declarada ninguna variable*, tan sólo se ha definido el tipo. Para declarar una variable, se hará como sigue:

```
struct direc info_direc;
```

Se pueden declarar una o más variables cuando se define una estructura. Por ejemplo:

```
struct direc
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
} info_direc, binfo, cinfo;
```

observar que direc es una *etiqueta* para la estructura que sirve como una forma breve para futuras declaraciones. Como en esta última declaración se indican las variables con esta estructura, se puede omitir el nombre de la estructura tipo.

Las estructuras pueden ser también preinicializadas en la declaración:

```
struct direc info_direc={"Vicente Fernandez","Fantasia
2000","Dorado","MMX",12345};
```



No olvidar: En una estructura (struct) pueden coexistir campos de distinto tipo.

Para referenciar o acceder a un campo de la estructura, C proporciona el operador punto ., por ejemplo, para asignar a info_direc otro código, lo hacemos como:

```
info_direc.codigo=54321;
```



Cómo definir un tipo registro:

```
struct <tipo_nombre>
{
    <tipo_campo1> <nombre1>;
    <tipo_campo2> <nombre2>;
    <tipo_campo3> <nombre3>;
    .
    .
    <tipo_campoN> <nombreN>;
};
```

Entonces, una estructura es un tipo de datos definido por el programador. Al utilizar una estructura cada programador tiene en sus manos la posibilidad de definir un nuevo tipo de datos considerablemente más complejo que los tipos que hemos utilizado hasta ahora. Una estructura es una combinación de varios tipos de datos previamente definidos, incluyendo otras estructuras que hayamos definido previamente. Para recordarla, una definición simple es: "una estructura es un grupo de datos relacionados en una forma conveniente al

programador y que tienen que ver con elementos de la realidad que pueden agruparse de manera lógica y relacional". Veamos el siguiente ejemplo:

```
#include <stdio.h>

struct
{
    char inicial;    /* Letra inicial del apellido */
    int  edad;       /* Edad */
    int  calificacion; /* Nota */
} alumno, alumna;

int main()
{
    alumno.inicial = 'R';
    alumno.edad = 15;
    alumno.calificacion = 75;
    alumna.edad = alumno.edad - 1; /* Ella es un año menor que él */
    alumna.calificacion = 82;
    alumna.inicial = 'H';

    printf("%c tiene %d años y su calificación es de %d\n",
           alumna.inicial, alumna.edad, alumna.calificacion);
    printf("%c tiene %d años y su calificación es de %d\n",
           alumno.inicial, alumno.edad, alumno.calificacion);

    return 0;
}
```

El programa empieza definiendo una estructura utilizando la palabra reservada **struct** seguida de tres variables sencillas encerradas entre llaves, las cuales son los componentes de la estructura, después de la llave de cierre tenemos declaradas dos variables llamadas **alumno** y **alumna**. De acuerdo a la definición de una estructura, alumno es una variable compuesta de tres elementos, inicial, edad y, calificación. Cada uno de los tres campos están

asociados a alumno y cada uno almacena una variable de su respectivo tipo, lo mismo se puede decir para alumna.



Nota previa: Preste mucha atención al desarrollo del siguiente tema (VECTORES CON ESTRUCTURAS) dado que el mismo es de vital importancia en el desarrollo de la asignatura.

VECTORES CON ESTRUCTURAS/REGISTROS

Es evidente que una variable de tipo registro aislada y por sí sola no nos permite el manejo de una gran cantidad de datos. Es por esto, que podemos recurrir a una estructura quizás un poco más compleja, pero que admite un manejo bastante importante en lo que a cantidad de datos se refiere. Esta estructura no es ni más ni menos que un vector donde cada una de las posiciones del mismo en lugar de ser enteros, reales o caracteres son *registros*. Esto confiere a la nueva estructura de posibilidades concretas para el manejo de una determinada cantidad limitada de datos, asociada obviamente al tamaño del vector de registros. Podemos ejemplificar esta estructura tomando en consideración el gráfico que se ve a continuación:

V

legajo [entero] nombre [string] nota [byte]

| | | | |
|---|------|---------|----|
| 1 | | | |
| 2 | 1287 | | |
| 3 | | | |
| 4 | | Alberto | |
| 5 | | | |
| . | | | |
| . | | | |
| . | | | |
| . | | | |
| . | | | |
| n | | | 10 |

Este vector V de n posiciones tiene componentes que son registros de tres campos: legajo (de tipo entero), nombre (de tipo cadena de caracteres) y nota (de tipo byte).

Para definir este tipo de datos vector de registros, deberemos usar normalmente en C, el typedef:

Definición de tipos: typedef

Se puede dar un nombre nuevo a cualquier tipo de datos mediante typedef.

La sintaxis es

```
typedef declaración;
```

donde *declaración* tiene la forma de una declaración de variable, sólo que se está definiendo un tipo de datos.

```
typedef long pareja [2];
```

define un tipo pareja que se puede usar en declaraciones de variables, por ejemplo:

```
pareja p;
```

es equivalente a

```
long p [2];
```

Ejemplos de typedef con estructuras

```
typedef struct Persona PERSONA;
```

```
PERSONA dato; /* igual que struct Persona dato; */
```

Un uso típico es la redefinición de tipos estructurados:

```
typedef struct    /* estructura anónima */
```

```
{
```

```
    char nombre[80];
```

```
    char sexo;
```

```
    int edad;
```

```
} Persona;          /* se declara el tipo Persona */
```

```
...
```

```
Persona p;
```

```
...
```

```
p.edad = 44;
```

Con C también se pueden tener arreglos de estructuras:

```
typedef struct direc  
{  
    char nombre[30];  
    char calle[40];  
    char ciudad[20];  
    char estado[3];  
    unsigned int codigo;  
} info_direc;
```

```
info_direc artistas[1000];
```

por lo anterior, artistas tiene 1000 elementos del tipo info_direc. Lo anterior podría ser accedido de la siguiente forma:

```
artistas[50].codigo=22222;
```

¿Qué no debemos olvidar del manejo de este tipo de estructuras?

-La definición de los tipos se realiza siempre desde la estructura más sencilla a la más compleja. Es decir que si el programador debe definir algún tipo (por ejemplo un tipo cadena de caracteres) que luego será utilizado para definir un campo en un registro, debe comenzar definiendo este tipo base y luego el tipo registro. Siempre, por último, se define el tipo vector de registros que involucra las demás estructuras definidas con anterioridad.

-El acceso a cada una de las posiciones del vector se realiza utilizando el índice correspondiente.

-El acceso a cada uno de los campos se realiza utilizando el *punto* “.”

-El tamaño de una variable de tipo registro sólo es modificable en tiempo de edición. No es posible modificarse en tiempo de corrida.

Vamos a un ejemplo de utilización de esta estructura: definimos un vector de 12 variables alumnos, está claro que este programa contiene 12 veces 3=36

variables sencillas cada una de las cuales puede almacenar un ítem de dato siempre y cuando sea del tipo adecuado, se define además una variable común llamada índice para utilizarla en los ciclos, véase el siguiente código:

```
#include <stdio.h>
```

```
struct
{
    char inicial;
    int edad;
    int calificacion;
}
alumnos[12];

main()
{
    int indice;

    for (indice = 0; indice < 12; indice++)
    {
        alumnos[indice].inicial = 'A' + indice;
        alumnos[indice].edad = 16;
        alumnos[indice].calificacion = 84;
    }

    alumnos[3].edad = alumnos[5].edad = 17;
    alumnos[2].calificacion = alumnos[6].calificacion = 92;
    alumnos[4].calificacion = 57;

    /* Asignación de estructura solo en compiladores ANSI-C */
    alumnos[10] = alumnos[4];

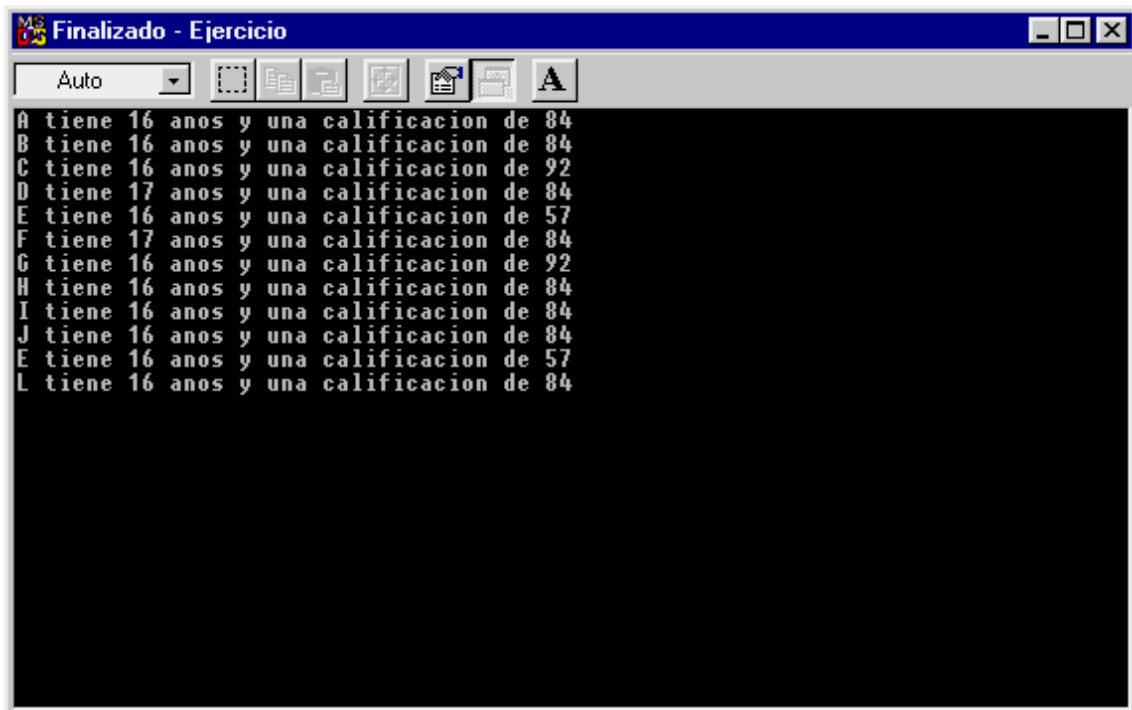
    for (indice = 0; indice < 12; indice++)
        printf("%c tiene %d años y una calificación de %d\n",
            alumnos[indice].inicial, alumnos[indice].edad,
            alumnos[indice].calificacion);

    return 0;
}
```

Para asignar un valor a cada uno de los campos utilizamos un bucle for, en cada ciclo del bucle se asignan todos los valores para uno de los alumnos, en una situación real ésta podría no ser la mejor manera de asignar datos, pero un bucle puede leer los datos de un archivo o de teclado y almacenarlos en la correcta ubicación en un programa real, este es solamente un ejemplo sencillo para comprender el manejo básico de la estructura. Como vemos, en la línea

que dice `alumnos[10] = alumnos[4]`; los tres campos de `alumnos[4]` son copiados en `alumnos[10]`, esto no siempre está permitido en Lenguaje C en todos los compiladores, pero es algo que debemos entender como posible porque se respetan las formas de las estructuras, es decir, hay un único tipo struct y eso hace que dicha operación de asignación sea posible.

El resultado de la ejecución del programa mostrado en una pantalla es el siguiente:



```

A tiene 16 años y una calificación de 84
B tiene 16 años y una calificación de 84
C tiene 16 años y una calificación de 92
D tiene 17 años y una calificación de 84
E tiene 16 años y una calificación de 57
F tiene 17 años y una calificación de 84
G tiene 16 años y una calificación de 92
H tiene 16 años y una calificación de 84
I tiene 16 años y una calificación de 84
J tiene 16 años y una calificación de 84
E tiene 16 años y una calificación de 57
L tiene 16 años y una calificación de 84
    
```



La definición de tipos se hace siempre desde la estructura más sencilla a la más compleja (que contiene a la más sencilla).

3.2. UTILIZACIÓN DE SUBPROGRAMAS:

INTRODUCCIÓN

Hasta aquí hemos manejado los programas como unidades totalmente monolíticas y sin ningún tipo de flexibilidad asociada a su programación. Es decir que hemos generado secuencias de instrucciones que deben ejecutarse una detrás de la otra siguiendo el orden de escritura de las mismas, desde la primera hasta la última. Este concepto lo seguiremos manteniendo, dado que el lenguaje de programación utilizado es de los denominados lineales pero le agregaremos una cierta flexibilidad que permitirá de alguna manera “segmentar” el programa u “ordenarlo”, con un criterio bastante amplio facilitando de alguna forma la programación del mismo y modularizándolo, haciendo que varias secuencias de instrucciones puedan ser utilizadas en muchos programas sin necesidad de retipear o reescribir los mismos. Estos módulos se llaman subprogramas y dentro de esta categoría se encuentran las funciones.

Vamos a estudiarlas orientando a las aplicaciones que se desprenden de su uso, de manera de familiarizarnos con las mismas rápidamente.



FUNCIONES

Una función es un conjunto de declaraciones, definiciones, expresiones y sentencias que realizan una tarea específica.

El formato general de una función en C es

```
especificador_de_tipo nombre_de_función( lista_de_parámetros )  
{  
    variables locales  
    código de la función  
}
```

El *especificador_de_tipo* indica el tipo del valor que la función devolverá mediante el uso de return. El valor puede ser de cualquier tipo válido. Si no se especifica un valor, entonces se asume por defecto que la función devolverá un resultado entero. No se tienen siempre que incluir parámetros en una función. La lista de parámetros puede estar vacía.

Las funciones terminan y regresan automáticamente al procedimiento que las llamó cuando se encuentra la última llave }, o bien, se puede forzar el regreso antes usando la sentencia return. Además del uso señalado la función return sirve para devolver un valor.

Se muestra a continuación un ejemplo que encuentra el promedio de dos enteros:

```
float encontprom(int num1, int num2)
{
    float promedio;

    promedio = (num1 + num2) / 2.0;
    return(promedio);
}
```

```
main()
{
    int a=7, b=10;
    float resultado;

    resultado = encontprom(a, b);
    printf("Promedio=%f\n",resultado);
}
```

Funciones void

Las funciones void dan una forma de emular, lo que en otros lenguajes se conocen como procedimientos (por ejemplo, en PASCAL). Se usan cuando no se requiere regresar un valor. Se muestra un ejemplo que imprime los cuadrados de ciertos números.

```
void cuadrados()
{
    int contador;
```

```
for( contador=1; contador<10; contador++)
    printf("%d\n",contador*contador);
}
```



La modularidad permite “desgranar una gran tarea en tareas más sencillas”. La codificación se hace más clara y reutilizable.

```
main()
{
    cuadrados();
}
```

En la función cuadrados no está definido ningún parámetro, y por otra parte tampoco se emplea la sentencia return para regresar de la función.

Funciones y arreglos

Cuando se usa un arreglo como un argumento a la función, se pasa sólo la dirección del arreglo y no la copia del arreglo entero. Para fines prácticos podemos considerar el nombre del arreglo sin ningún índice como la dirección del arreglo.

Considerar el siguiente ejemplo en donde se pasa un arreglo a la función imp_rev, observar que no es necesario especificar la dimensión del arreglo cuando es un parámetro de la función.

```
void imp_rev(char s[])
{
    int t;

    for( t=strlen(s)-1; t>=0; t--)
        printf("%c",s[t]);
}
```

```
main()
{
    char nombre[]="Facultad";
    imp_rev(nombre);
}
```

Observar que en la función `imp_rev` se usa la función `strlen` para calcular la longitud de la cadena sin incluir el terminador nulo. Por otra parte, la función `imp_rev` no usa la sentencia `return` ni para terminar de usar la función, ni para regresar algún valor.

Se muestra otro ejemplo,

```
float enconprom(int tam, float lista[])
{
    int i;
    float suma = 0.0;

    for ( i=0; i<tam; i++)
        suma += lista[i];
    return(suma/tam);
}
```

```
main()
{
    float numeros[]={2.3, 8.0, 15.0, 20.2, 44.01, -3.0, -2.9};
    printf("El promedio de la lista es %f\n", enconprom(7,numeros) );
}
```

Para el caso de que se tenga que pasar un arreglo con más de una dimensión, no se indica la primera dimensión pero, el resto de las dimensiones deben señalarse. Se muestra a continuación un ejemplo:

```
void imprtabla(int tamx,int tamy, float tabla[][5])
{
```



```
int x,y;

for ( x=0; x<tamx; x++ )
{
    for ( y=0; y<tamy; y++ )
        printf("t[%d][%d]=%f",x,y,tabla[x][y]);
    printf("\n");
}
}
```

Prototipos de funciones

Antes de usar una función C, se debe tener *conocimiento* acerca del tipo de dato que regresará y el tipo de los parámetros que la función espera.

Básicamente si una función ha sido definida antes de que sea usada (o llamada), entonces se puede usar la función sin problemas.

Si no es así, entonces la función se debe *declarar*. La declaración simplemente maneja el tipo de dato que la función regresa y el tipo de parámetros usados por la función.

Es una práctica usual y conveniente escribir el prototipo de todas las funciones al principio del programa, sin embargo esto no es estrictamente necesario.

Para *declarar* un prototipo de una función se indicará el tipo de dato que regresará la función, el nombre de la función y entre paréntesis la lista del tipo de los parámetros de acuerdo al orden que aparecen en la definición de la función. Por ejemplo:

```
int longcad(char []);
```

Lo anterior declara una función llamada longcad que regresa un valor entero y acepta una cadena como parámetro.

Entonces,

Las funciones son siempre **globales**, esto es, no se permite declarar una función dentro de otra.

Las funciones son visibles sólo después de que se han declarado.

Se pueden **declarar** funciones, especificando sólo su formato, pero no su cuerpo:

```
int suma ( int a, int b );
```

lo anterior es una declaración de la función **suma**, que queda disponible para su uso, a pesar de no haber sido definido su cuerpo.

La declaración de una función de esta forma se llama **prototipo**.

Es buena práctica declarar al comienzo del programa los prototipos de las funciones que vamos a definir, incluyendo comentarios sobre su finalidad.

A partir de ahora podemos trabajar utilizando estos subprogramas de manera de modularizar nuestros programas, facilitando de manera sumamente importante la escritura de código asociado a cada una de nuestras soluciones.



Genere a partir de lo visto sus comentarios, ventajas y desventajas en la utilización de este tipo de conceptos como así también en lo relativo al uso de registros y vectores con registros.

Uniones

Una unión es una variable la cual podría guardar (en momentos diferentes) objetos de diferentes tamaños y tipos. C emplea la sentencia union para crear uniones, por ejemplo:

```
union numero
{
    short shortnumero;
    long longnumero;
    double floatnumero;
} unumero;
```

con lo anterior se define una unión llamada numero y una instancia de esta llamada unumero. numero es la etiqueta de la unión y tiene el mismo comportamiento que la etiqueta en la estructura.

Los campos pueden ser direccionados de la siguiente forma:

```
printf("%ld\n", unumero.longnumero);
```

Cuando el compilador de C está reservando memoria para las uniones, siempre creará una variable lo suficientemente grande para que pueda ser alojado el tipo de variable más largo de la unión.

Con la finalidad de que el programa pueda llevar el registro del tipo de la variable unión usada en un momento dado, es común tener una estructura (con una unión anidada) y una variable que indica el tipo de unión.

Se muestra un ejemplo de lo anterior:

```
typedef struct
```

```
{  
    int maxpasajeros;  
} jet;
```

```
typedef struct
```

```
{  
    int capac_elev;  
} helicoptero;
```

```
typedef struct
```

```
{  
    int maxcarga;  
} avioncarga;
```

```
typedef union
```

```
{  
    jet jetu;  
    helicoptero helicoptero;  
    avioncarga avioncargau;  
} transporteaereo;
```

```
typedef struct
```

```
{  
    int tipo;
```

```
int velocidad;
transporteaereo descripcion;
} un_transporteaereo
```

en el ejemplo se define una unión base de transporte aéreo el cual puede ser un jet, un helicóptero o un avion de carga.

En la estructura un_transporeaereo hay un campo para el tipo, que indica cual es la estructura manejada en ese momento.

3.3. ARCHIVOS

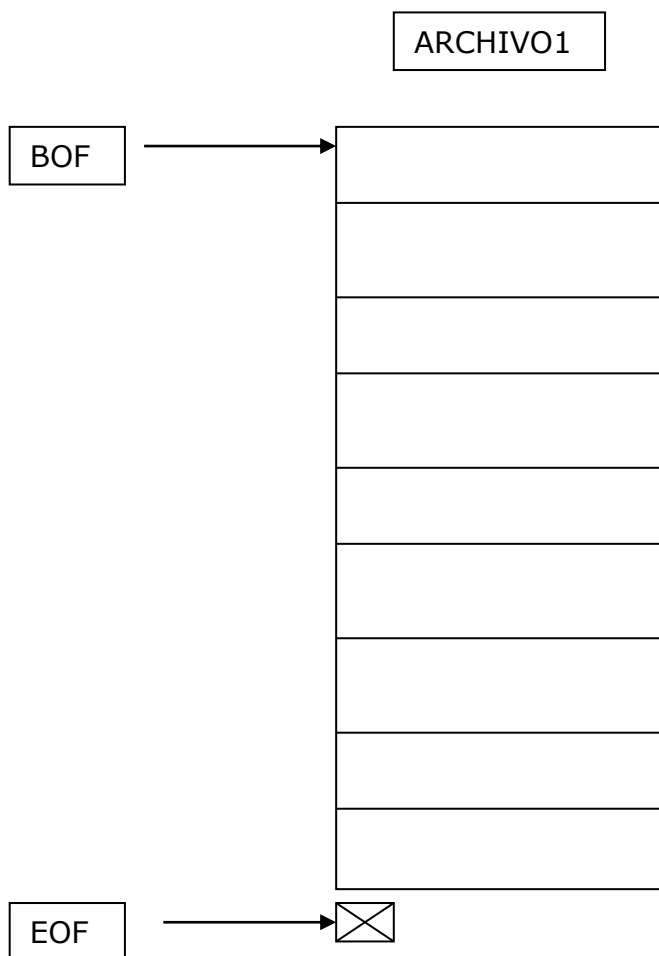
Ustedes ya han tenido contacto con la definición de archivo en la asignatura Estructura de un Computador.

Aquí vamos a refrescarla, un *archivo* es una estructura dinámica (puede variar su tamaño en tiempo de ejecución) utilizada para almacenar datos. En principio, en los archivos que vamos a utilizar, todos los datos van a ser del mismo tipo. Esto significa que si creamos un archivo de enteros, sólo servirá para trabajar con números enteros. El tamaño de un archivo no es fijo (como en el caso de un vector) y puede variar a medida que se agregan o quitan datos en tiempo de corrida de un programa. El archivo ocupa un espacio determinado del disco de almacenaje (sea este un disco duro, un diskette, un CD o cualquier otro tipo de elemento que permita resguardar este tipo de estructuras), se lo identifica con un nombre formado de la siguiente manera: <nombre>.<extensión> donde el <nombre> es elegido por el programador y la extensión tendrá que ver normalmente con el tipo de utilitario que se use para su manejo. Cuando el archivo es utilizado en un programa, se establece una relación directa entre el archivo que tenemos almacenado en el disco, y la variable de tipo archivo que será generada en el programa. De esta forma, los datos contenidos por el mismo pueden ser manejados por el programa. Otro elemento importante en un archivo es la identificación concreta del principio y fin de archivo (begin of file – BOF y end of file – EOF).





Un archivo es una estructura dinámica (permite variar su tamaño en tiempo de ejecución).



Aquí vemos la estructura básica de un archivo con sus partes componentes tal como hemos descripto precedentemente.

Vamos ahora a trabajar sobre las distintas instrucciones que permiten manejar los datos en esta estructura.

La biblioteca **stdio.h** contiene funciones para trabajar con archivos: básicamente leer y escribir datos de diferentes formas.

Antes de trabajar con un archivo, hay que **abrirlo**. Cuando se abre un archivo, se devuelve un puntero a una estructura de tipo **FILE** (definido en STDIO). Esta estructura, llamada **descriptor de archivo**, servirá para manipular posteriormente el mismo.

Tras su uso, hay que **cerrar** el archivo.

Modelo de uso:

```
FILE* fd;           /* variable para apuntar al descriptor de archivo */
```

```
...
```

```
/* abre el archivo en modo lectura */
```

```
fd = fopen ( "pepe.txt", "rt" );
```

```
    ... trabaja con el archivo ...
```

```
fclose (fd);        /* cierra el archivo */
```

Muchas de las funciones de STDIO para trabajar con archivos comienzan con la letra "f" (fopen, fclose, fprintf, fwrite, etc.)

¿Cómo abrimos el archivo?

fd = fopen (nombre, modo);

Devuelve **NULL** si no se pudo abrir correctamente el archivo.

nombre es el nombre del archivo.

modo es una cadena donde se define el modo de apertura:

r sólo lectura

w escritura

a apéndice / agregar elementos (escribir desde el final)

+ (combinado con r,w ó a) lectura/escritura

t archivo de texto

b archivo binario

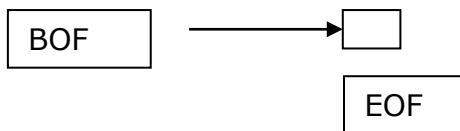
Ejemplo:

```
fd = fopen ( "c:\\ejemplos\\archivo.txt", "r+t" );
```

abre el archivo **c:\\ejemplos\\archivo.txt** en modo lectura/escritura y en modo texto. Obsérvese que las barras en el nombre del archivo tienen que escribirse duplicadas.

Un archivo abierto en modo texto convierte la combinación CR+LF en un caracter de salto de línea. Si es binario, CR+LF se consideran dos caracteres independientes.

Hay un archivo muy especial, el archivo vacío, que tiene la siguiente forma:



En el archivo vacío, el begin of file coincide con el end of file.

¿Cómo cerramos el archivo una vez que hemos realizado todas las operaciones?

```
fclose ( fd );
```

donde **fd** es un descriptor de archivo ya abierto.

¿Cómo leemos una cadena de un archivo?

```
fgets ( cadena, N, fd );
```

Lee una línea de texto.

cadena es el vector de caracteres donde se depositará la línea.

N es el máximo número de caracteres para leer. Si la línea ocupa más de **N** caracteres, sólo se lee esa cantidad.

fd es el descriptor del archivo.

Cuando se abre un archivo, se comienza a leer desde la primera línea.

Llamadas consecutivas a **fgets** van leyendo línea por línea. Si se cierra el archivo y se vuelve a abrir, se lee otra vez desde el principio.

Internamente existe un **puntero del archivo** que apunta a la posición relativa al archivo donde se va a leer el siguiente dato.

¿Cómo escribimos una cadena en un archivo?

fputs (cadena, fd);

cadena es el texto que se escribe. **fputs** incluye además un salto de línea.

fd es el descriptor del archivo.

Las líneas escritas se van añadiendo una detrás de otra. Se comienza por el principio del archivo (**w**) o por el final (**a**). También existe internamente un puntero de archivo para escritura.

fprintf (fd, formato, arg1, arg2, ...);

Escribe un texto con formato en un archivo. La sintaxis es idéntica a **printf**, salvo que el primer argumento es un descriptor de archivo.

¿Cómo detectamos el final de un archivo?

x = feof (fd);

Devuelve un 1 si se ha llegado al final del archivo. Un cero en caso contrario.

Es muy útil para saber cuándo dejar de leer información del archivo.

Algunas funciones especiales adicionales para trabajar con archivos

rewind (fd);

Vuelve al comienzo del archivo (rebobina).

fseek (fd, posición, de_dónde);

Se mueve a la **posición** del archivo indicada, relativa a un punto especificado en **de_dónde**.

posición es de tipo **long**.

de_dónde puede tener los siguientes valores:

SEEK_SET desde el inicio del archivo

SEEK_CUR desde la posición actual del archivo

SEEK_END desde el final del archivo

Ejemplo

```
FILE* fich;
char cadena [16];
/* lee los últimos 15 caracteres del archivo */
fich = fopen ( "archivo.txt", "rt" );
fseek ( fd, -15L, SEEK_END);
fgets ( texto, 15, fich );
fclose (fich);
```

¿Qué es un stream?

Los archivos de donde se lee o escribe información con las rutinas de STDIO se llaman **flujos** o *streams*. Un flujo puede ser un archivo, pero también puede ser el teclado (se puede leer de él) o la pantalla (se puede escribir en ella).

Los llamados **flujos estándar** son los siguientes:

| | |
|---------------|---|
| stdin | entrada estándar (normalmente el teclado) |
| stdout | salida estándar (normalmente la pantalla) |
| stderr | error estándar (normalmente la pantalla) |

Se pueden utilizar con las funciones de STDIO:

```
fgets ( cadena, 80, stdin );  
fprintf (stderr, "Se ha producido un error" );
```

¿Cómo se manejan los errores?

Muchas funciones de biblioteca devuelven un -1 o un NULL si ocurre un error.

Esto es habitual en las funciones de tratamiento de archivos.

La variable **errno** contiene un código de error relativo al último error producido.

Para trabajar con **errno** hay que incluir el archivo **errno.h**

En **errno.h** están definidos todos los códigos de error posibles.

Aparte, existe la función **perror()**, que da un mensaje sobre el error producido.

EJERCITACIÓN

Práctica 2: Estructuras

Ejercicio 1: Realizar un programa que permita leer de teclado y luego imprimir los contenidos de una variable tipo struct cuyos campos son: nombre (cadena de caracteres [10]), legajo (entero), dependencia de trabajo (entero), horas trabajadas (real).

Ejercicio 2: Realizar un programa que permita, dado un vector de estructuras/registros cuyos campos son: empresa (cadena de caracteres [30]), domicilio (cadena de caracteres [40]), código postal (cadena de caracteres [10]), localidad (cadena de caracteres [20]), país (cadena de caracteres [15]), imprimir un listado para generar el destinatario de unas cartas. El formato debe ser el correcto.

Ejercicio 3: Realizar un programa que permita, dado un vector A de estructuras/registros, cuyos campos son: producto (cadena de caracteres [10]), código (entero) y precio (real); obtener un nuevo vector B de estructuras/registros que contengan sólo el código y el precio del vector A.

Ejercicio 4: Dado un vector B de estructuras/registros de alumnos cuyos campos son: nombre (cadena de caracteres [10]) y nota (entero); realizar mediante un programa la impresión de los nombres de los alumnos cuyas notas superan el valor 7.

Nota: Los vectores y registros deben ser cargados desde teclado e impresos en pantalla.

Práctica 3: Archivos

Ejercicio 1: Dado un archivo de registros de tres campos (legajo: entero; sueldo: real; adicional: real), realizar un programa que permita actualizar el archivo, de modo que el sueldo sea incrementado en 0.5 veces el adicional. Los sueldos que se deben modificar son aquellos que cumplan con alguna de las siguientes condiciones:

- Legajo entre 1 y 5454, y el sueldo menor a \$350.
- Legajo entre 6000 y 9800, y el adicional sea menor a \$200.
- Legajo mayor a 15300.

Ejercicio 2: Realizar un procedimiento al que dado un archivo de reales, devuelva la suma de todos los elementos del mismo. El resultado de la operación debe devolverse en una variable.

Ejercicio 3: En una empresa textil se desea generar un programa que permita realizar el control de la producción de telas impermeables para mochilas. Se sabe que los datos se encuentran guardados en un archivo pero que los mismos fueron ingresados por alguien que desconocía una modificación introducida recientemente a los códigos de los distintos tipos de telas. Es así, que el código correspondiente al nylon de primera era 'NYA' y ahora es 'NY1', mientras que el de la tela de avión era 'AVI' y ahora es 'G25'. Se desea pues, actualizar el archivo de producción con estos nuevos códigos. El archivo es de registros de tres campos: partida (entero), código (cadena de tres caracteres) y cantidad de macrorrollos (entero). El nombre del archivo es PRODUC.

Ejercicio 4: En una empresa tabacalera, se desea generar un programa que permita realizar el control de la producción de cigarrillos y habanos. Se sabe que los datos se encuentran guardados en un archivo, pero que los mismos deben ser actualizados por cuanto dos de los códigos no serán utilizados en lo sucesivo. Se deberán cambiar 'JCS' por 'JCA' y 'PIP' por 'IPE'. El archivo es de registros de tres campos: partida (entero), código (cadena de tres caracteres) y cantidad de paquetes (entero). El nombre del archivo es PRODTAB.

Ejercicio 5: Dado un archivo de accidentes de tránsito, cuyos elementos son registros de tres campos: fecha (cadena de ocho caracteres), cantidad de accidentados (entero) y código de zona (entero). El código de zona varía entre 1 y 20. Realizar un programa que permita generar un vector donde se vayan acumulando la cantidad de accidentes por cada zona (el índice del vector coincide con el número de zona). Una vez obtenido el vector, imprimirlo.

Ejercicio 6: Realizar un programa que debe permitir imprimir los nombres de un archivo de registros cuyos campos son: legajo (entero), nombre (cadena de 30 caracteres) y edad (entero). Los nombres a imprimir son aquellos que cumplan las siguientes condiciones:

- Número de legajo entre 1 y 10000 sin importar la edad.
- Edad mayor a 18 años sin importar el número de legajo.

Ejercicio 7: Realizar un procedimiento en el que dado un archivo de registros de dos campos (código: cadena de tres caracteres y cantidad: entero), permita eliminar los elementos que cumplan alguna de estas condiciones: el código empieza con 'H', 'T' o 'ZA' o la cantidad es menor a 92.

UNIDAD 4: RECURSIVIDAD

¿Cuáles son nuestros objetivos para esta unidad?

La idea en esta tercera parte es aprender el uso de un recurso al que muchos programadores le rehuyen. El mismo se llama recursividad, y de alguna manera –bien utilizada- se puede convertir en una buena compañera del programador en lugar de constituir una carga. Muchos plantean su defensa diciendo que no la utilizan porque hay que pensar. Veremos que la forma de solucionar el problema a través de la recursividad no está muy lejos de la que usamos habitualmente, la única diferencia es que tiene otra forma.

Nuestros objetivos son, entonces:

Lograr modularizar un programa en C utilizando funciones recursivas.

4.1. RECURSIVIDAD

Existen muchas funciones matemáticas cuyos argumentos son números naturales, que pueden definirse de manera *recursiva*. Esto quiere decir que el valor de la función para el argumento n puede definirse en términos del argumento $n-1$ (o alguno anterior). En este tipo de definiciones siempre existirá un *caso base* a partir del cual parte la definición, el cual normalmente es el valor de la función en cero o en uno, aunque no necesariamente debe ser así. Por ejemplo, el factorial puede definirse de manera recursiva de la siguiente manera:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n \cdot (n-1)!, & \text{si } n > 0 \end{cases}$$

Para definir una función en forma recursiva es necesario especificar:

Caso(s) base: Donde la recursividad se detiene

Paso de recursión: Como se define un elemento distinto del base, en términos de elementos anteriores.

Usualmente los lenguajes de programación permiten definir funciones de manera recursiva. El lenguaje C es uno de ellos. La definición recursiva para el factorial sería:

```
int factorial(int n)
{
    if ((n == 0) || (n == 1))
        return(1);
    else
        return(n*factorial(n-1));
}
```

Normalmente las definiciones recursivas pueden expresarse en forma no recursiva. Sin embargo, dependiendo del caso, el resultado puede ser más confuso. Por ejemplo, una función en C que calcula el factorial en forma iterativa sería:

```
int factorial(int n)
{
    int i, fact = 1;

    for (i=2; i<=n; i++)
        fact = fact * i;
    return(fact);
}
```

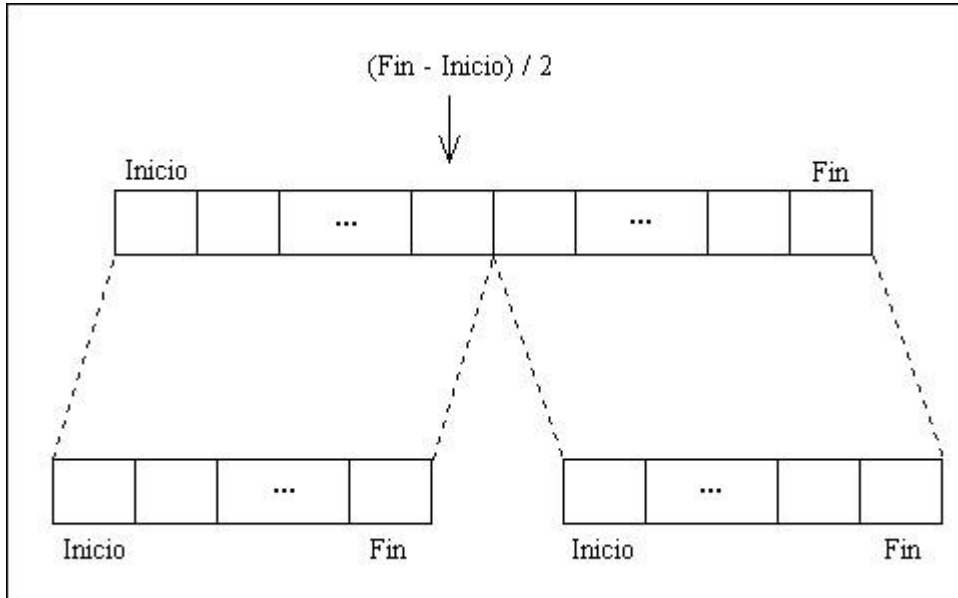
Sin embargo, los algoritmos iterativos tienen una ventaja en cuanto al uso de memoria, si se comparan con los recursivos. La recursividad requiere que se guarde el estado de la ejecución antes de cada llamado recursivo, implicando un gasto considerable de memoria. La aplicación de las definiciones recursivas puede ampliarse a una importante gama de problemas, en los que la solución más natural puede ser la que se expresa de esta forma. Por ejemplo, para buscar un número en un vector podemos tener una función que reciba como argumento el vector, el rango de búsqueda y el número buscado. El prototipo de esta función sería como:

```
int busqueda(int vec[], int inicio, int fin, int num);
```

La función que quiere hacer la búsqueda haría el llamado indicando los rangos apropiados para el vector:

```
resultado = busqueda(vector, 0, N, x);
```

La función *busqueda()* puede hacer su trabajo partiendo el vector en dos partes y llamándose a sí misma en forma recursiva, de la siguiente manera:



```
res1 = busqueda(vec, inicio, fin-inicio/2, num);
```

```
res2 = busqueda(vec, (fin-inicio/2)+1, fin, num);
```

El caso base sería cuando el vector que recibe la función *busqueda()* contiene un único elemento. En este caso simplemente compara el elemento con el número buscado y retorna el valor apropiado.

Veamos el siguiente ejemplo:

```
#include <stdio.h> /* Contiene el prototipo para printf */
```

```
void count_dn(int count) ; /* Prototipo para count_dn */
```

```
main( )
```

```
{
```

```
    int index ;
```

```
    index = 8 ;
```

```
    count_dn(index) ;
```

```
    return 0 ;
```

```
}
```



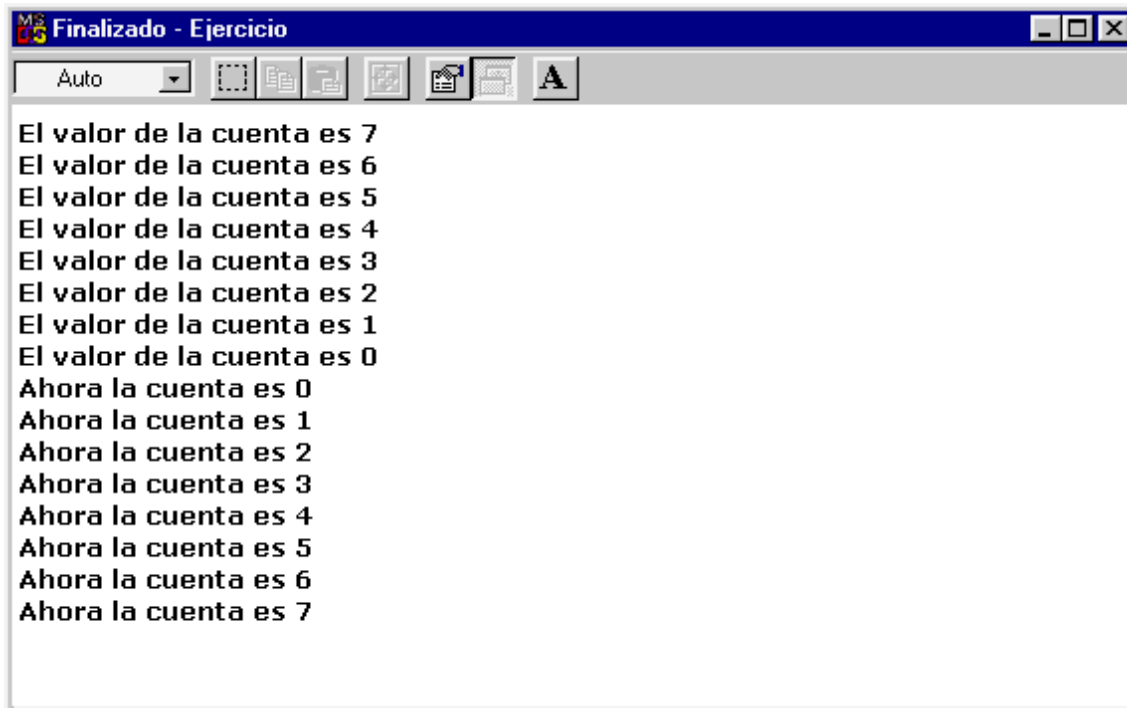
```
void count_dn(int count)
{
    count -- ;
    printf ( "El valor de la cuenta es %d\n", count ) ;
    if (count > 0)
        count_dn(count) ;
    printf ( "Ahora la cuenta es %d\n", count ) ;
}
```

Como vimos, la recursividad no es otra cosa que una función que se llama a sí misma, es por lo tanto un bucle que debe tener una forma de terminar, tiene necesariamente que existir una condición de corte. Si esto no sucede, el programa quedará en un lazo de iteración infinita y por lo tanto se estará ejecutando de manera continua. En el programa, la variable index es cargada con el valor 8 y es utilizada como el argumento de la función llamada count_dn (). La función simplemente decrementa la variable, despliega un mensaje, y si la variable es mayor que cero, se llama a sí misma donde decrementa la variable una vez más, despliega un mensaje, etc., etc., etc. Finalmente la variable alcanza el valor de cero y la función ya no se llama a sí misma (ésta es justamente la condición de corte), y el control del programa retorna al punto previo a su llamada. Esto se hará tantas veces como llamadas recursivas haya tenido el programa, hasta que finalmente retorna a la función main () y continúa con la ejecución de la siguiente línea a la del llamado a la función recursiva desde el main(). Tal vez esto los ayude a imaginarlo un poco más, piensen como si tuvieran ocho funciones llamadas count_dn disponibles y que van llamando una a la vez manteniendo un registro de cuál de las copias estuvo activa en cada uno de los llamados, esto no es en realidad lo que sucede en el programa pero a manera de comparación resulta útil para comprender el funcionamiento del mismo.

Cuando llaman a la función desde la misma función, esta almacena todas las variables y demás datos que necesita para completar la función en un bloque interno. La siguiente vez que es llamada la función hace exactamente lo mismo creando otro bloque interno, este ciclo se repite hasta alcanzar la última

llamada a la función, entonces empieza a regresar los bloques utilizando estos para completar cada llamada de función.

Luego de la ejecución del programa, la pantalla mostrará:



Palabras clave cuando nos referimos a recursividad:
 caso base, condición de corte y llamado recursivo.
 ¡No se olviden de todos estos conceptos!

EJERCITACIÓN

Práctica 4: Recursividad

Ejercicio 1: Realizar un subprograma que permita calcular la siguiente serie:

$$a^n + a^{(n-1)} + a^{(n-2)} + \dots + a^2 + a + 1$$

Ejercicio 2: Realizar un subprograma que permita calcular la siguiente serie:

$$1 + 2 + 3 + 4 + \dots + n$$

Ejercicio 3: Realizar un subprograma que permita leer un vector de registros con tres campos: alumno, legajo, nota.

Ejercicio 4: Realizar un subprograma que permita adicionar al campo nota del vector anterior, un punto si el apellido del alumno comienza con la letra L.

Ejercicio 5: Calcular la suma de los elementos de un vector de n componentes reales.

Ejercicio 6: Realizar un subprograma que permita determinar cuántas letras 'A' o 'a' hay en un string de n caracteres.

Ejercicio 7: Calcular la potencia n de un número a (a^n), con n entero.

Ejercicio 8: Calcular el producto $a*b$ con a entero y menor que cero, y b real.

Ejercicio 9: Un parámetro de importancia en la producción algodonera es el que permite determinar el promedio de toneladas producidas por quintal cada año. Dado un vector constituido por registros de dos campos: tipo (cadena de 20 caracteres) y toneladas por quintal (real), se desea hallar el promedio de toneladas por quintal producidos del algodón tipo 'Capullo-Calidad1'. El vector tiene 72 posiciones y el proceso que permita el cálculo debe ser recursivo.

Ejercicio 10: Realizar un subprograma que devuelva el producto de todos los elementos diagonales de una matriz de dimensión $n \times n$.

Ejercicio 11: Realizar un subprograma recursivo que permita imprimir 40 líneas en blanco en pantalla.

UNIDAD 5: PUNTEROS

¿Cuáles son nuestros objetivos para esta unidad?

La idea es conocer la utilidad de otra estructura dinámica cuyo uso ha sido muy controvertido a lo largo de la corta historia de la programación. Algunos consideran esta herramienta como algo que debe permanecer oculto al manejo del programador –o sea debe ser parte del manejo del firmware (hardware + software de sistema operativo) de un dispositivo-, mientras otros creen que es un excelente arma adicional al momento de programar.

Nuestros objetivos son:

Conocer el fundamento de un puntero.

Utilizar punteros para soluciones que requieran de variables dinámicas.

5.1. CONCEPTO DE PUNTERO

DEFINICIÓN DE PUNTERO

Los punteros son tipos de datos que permiten crear estructuras dinámicas, las cuales pueden variar en tamaño y necesidad de memoria requerida. Durante la ejecución de un programa, puede haber una posición de memoria específica asociada con una variable dinámica y posteriormente puede no existir ninguna posición de memoria asociada con ella.

Una estructura de datos dinámica es una colección de elementos que se enlazan o encadenan. Este enlace se establece asociando con cada elemento un puntero que apunta al siguiente elemento de la estructura.

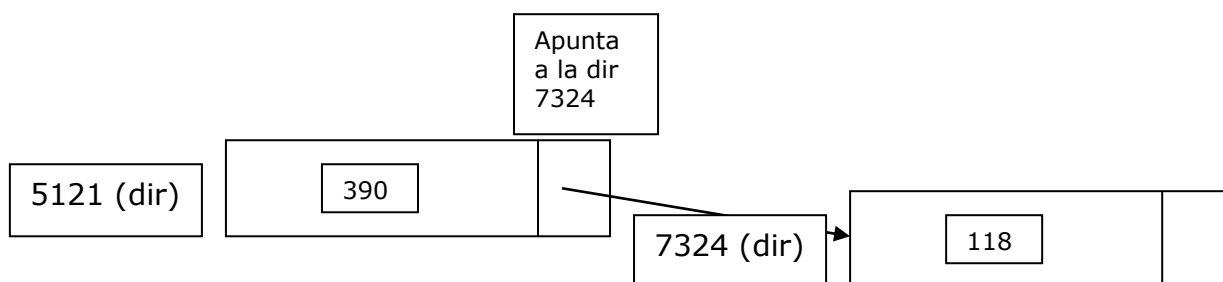
Las estructuras dinámicas –como ya lo hemos visto- son útiles para almacenar y procesar conjuntos de datos cuyos tamaños cambian durante la ejecución del programa.

El tipo de datos *puntero* es de tipo simple pues no se puede romper en otros componentes más pequeños, como sí sucede con el array o el registro. Los *punteros* son variables que se utilizan para almacenar la dirección de memoria de otra variable. Las variables que se utilizan para almacenar direcciones son llamadas *variables puntero* o simplemente *puntero*.

Al definir un puntero se debe indicar el tipo de valores que se almacenarán en las posiciones designadas por los punteros. Esto se debe a que los diferentes tipos de datos requieren distintas cantidades de memoria para almacenar sus constantes, una variable puntero puede contener una dirección de una posición de memoria adecuada sólo para un tipo dado.

Por esta razón se dice que un puntero apunta a una variable particular, es decir, a otra posición de memoria.

Una variable tipo puntero contiene la dirección de la posición de otra variable.



El puntero guardado en la dirección 5121, tiene el valor 390 guardado como dato y apunta a la dirección 7324, o sea que el próximo elemento de la estructura después del que está en la dirección 5121 es el de la dirección 7324. El dato que está guardado en esta dirección es 118.

Entonces, un puntero es una *variable* que contiene la dirección en memoria de otra variable. Se pueden tener punteros a cualquier tipo de variable.

El operador **unario** o **monádico** & devuelve la dirección de memoria de una variable.

El operador de **indirección** o **de referencia** * devuelve el "contenido de un objeto apuntado por un puntero".

Para declarar un puntero para una variable entera, por ejemplo, se debe escribir:

```
int *puntero;
```

Se debe asociar a cada puntero un tipo particular. Por ejemplo, no se puede asignar la dirección de un short int a un long int.

Para tener una mejor aproximación, podemos considerar el siguiente código:

```
main()
```

```
{
```

```

int x = 1, y = 2;
int *ap;
ap = &x;
y = *ap;
x = ap;
*ap = 3;
}

```

Con el objetivo de entender el comportamiento del código supongamos que la variable x está en la posición de memoria 100, y en 200, y ap en 1000.

Nota: un puntero es una variable, por lo tanto, sus valores necesitan ser guardados en algún espacio de memoria.

```

int x = 1, y = 2;
int *ap;
ap = &x;

```

| | | |
|-----|-----|--------|
| 100 | 200 | 1000 |
| x 1 | y 2 | ap 100 |

Las variables x e y son declaradas e inicializadas con 1 y 2 respectivamente, ap es declarado como un puntero a entero y se le asigna la dirección de x (&x). Por lo que ap se carga con el valor 100.

```
y = *ap;
```

| | | |
|-----|-----|--------|
| 100 | 200 | 1000 |
| x 1 | y 1 | ap 100 |

En el ejemplo ap apunta a la posición de memoria 100 -- la posición de x. Por lo tanto, y obtiene el valor de x -- el cual es 1.

```
x = ap;
```

| | | | | | |
|---|-----|---|-----|----|------|
| | 100 | | 200 | | 1000 |
| x | 100 | y | 1 | ap | 100 |

Como se ha visto C no es muy estricto en la asignación de valores de diferente tipo (puntero a entero). Así que es perfectamente legal (aunque el compilador genera un aviso de cuidado) asignar el valor actual de ap a la variable x. El valor de ap en ese momento es 100.

*ap = 3;

| | | | | | |
|---|-----|---|-----|----|------|
| | 100 | | 200 | | 1000 |
| x | 3 | y | 1 | ap | 100 |

Se asigna un valor al contenido de un puntero (*ap).



Un puntero es una variable que contiene la dirección de memoria de otra variable.

Importante: Cuando un puntero es declarado apunta a algún lado. Se debe inicializar el puntero antes de usarlo. Por lo que:

```
main()
{
    int *ap;
    *ap = 100;
}
```

puede generar un error en tiempo de ejecución o presentar un comportamiento errático.

El uso correcto será:

```
main()
{
    int *ap;
    int x;
    ap = &x;
    *ap = 100;
}
```

Con los punteros se puede realizar también aritmética entera, por ejemplo:

```
main()
{
    float *flp, *flq;
    *flp = *flp + 10;
    ++*flp;
    (*flp)++;
    flq = flp;
}
```

5.2. PUNTEROS Y FUNCIONES

Cuando C pasa argumentos a funciones, los pasa *por valor*, es decir, si el parámetro es modificado dentro de la función, una vez que termina de ejecutarse la función, el valor pasado de la variable permanece inalterado. Hay muchos casos que se quiere alterar el argumento pasado a la función y recibir el nuevo valor una vez que la función ha terminado. Para hacer lo anterior se debe usar una *llamada por referencia*, en C se puede simular pasando un puntero al argumento. Con esto se provoca que la computadora pase la dirección del argumento a la función.

Para entender mejor lo anterior consideremos la función `swap()` que intercambia el valor de dos argumentos enteros:

```
void swap(int *px, int *py);
```

```
main()
{
    int x, y;
    x = 10;
    y = 20;
    printf("x=%d\ty=%d\n",x,y);
    swap(&x, &y);
    printf("x=%d\ty=%d\n",x,y);
}
```

```
void swap(int *px, int *py)
{
    int temp;
    temp = *px; /* guarda el valor de la direccion x */
    *px = *py; /* pone y en x */
    *py = temp; /* pone x en y */
}
```

5.3 PUNTEROS Y ARREGLOS

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo. Considerar lo siguiente:

```
int a[10], x;
int *ap;
ap = &a[0]; /* ap apunta a la direccion de a[0] */
x = *ap; /* A x se le asigna el contenido de ap (a[0] en este caso) */
*(ap + 1) = 100; /* Se asigna al segundo elemento de 'a' el valor 100 usando ap*/
```

Como se puede observar en el ejemplo la sentencia **a[t]** es idéntica a **ap+t**. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo,

por lo que se puede ir fácilmente más allá del arreglo en memoria y sobreescribir posiciones de memoria que no correspondan al mismo.

C sin embargo es mucho más sutil en su relación entre arreglos y punteros, hay varias formas de escribir lo mismo. Por ejemplo se puede teclear solamente:

`ap = a;` en vez de `ap = &a[0];`

y también `*(a + i)` en vez de `a[i]`, esto es, `&a[i]` es equivalente con `a+i`.

Y como sigue, el direccionamiento de punteros se puede expresar como:

`a[i]` que es equivalente a `*(ap + i)`

Sin embargo los punteros y los arreglos son diferentes:

- Un puntero es una variable. Se puede hacer `ap = a` y `ap++`.
- Un arreglo no es manejable como una variable. Hacer `a = ap` y `a++` no es correcto.

Esto es muy importante, asegúrese haberla entendido.

Con lo comentado se puede entender cómo los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le está pasando es la ubicación de su elemento inicial en memoria.

Por lo tanto:

`strlen(s)` es equivalente a `strlen(&s[0])`

Esta es la razón por la cual se declara la función como:

`int strlen(char s[]);` y una declaración equivalente es `int strlen(char *s);`

ya que `char s[]` es igual que `char *s`.

La función `strlen()` es una función de la *biblioteca estándar* que regresa la longitud de una cadena. Se muestra enseguida la versión de esta función que podría escribirse:

```
int strlen(char *s)
{
    char *p = s;
    while ( *p != '\0' )
        p++;
    return p - s;
}
```

Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

```
void strcpy(char *s, char *t)
{
    while ( (*s++ = *t++) != '\0' );
}
```

En los dos últimos ejemplos se emplean apuntadores y asignación por valor.

Nota: Se emplea el uso del caracter nulo con la sentencia while para encontrar el fin de la cadena.

5.4. ARREGLOS DE PUNTEROS

En C se pueden tener arreglos de punteros ya que los punteros son variables.

A continuación se muestra un ejemplo de su uso: **ordenar las líneas de un texto de diferente longitud.**

Los arreglos de punteros son una representación de datos que manejan de una forma eficiente y conveniente líneas de texto de longitud variable.

¿Cómo se puede hacer lo anterior?

- Guardar todas las líneas en un arreglo de tipo char. Observando que \n marca el fin de cada línea. Ver figura.

- Guardar los punteros en un arreglo diferente donde cada puntero apunta al primer caracter de cada línea.

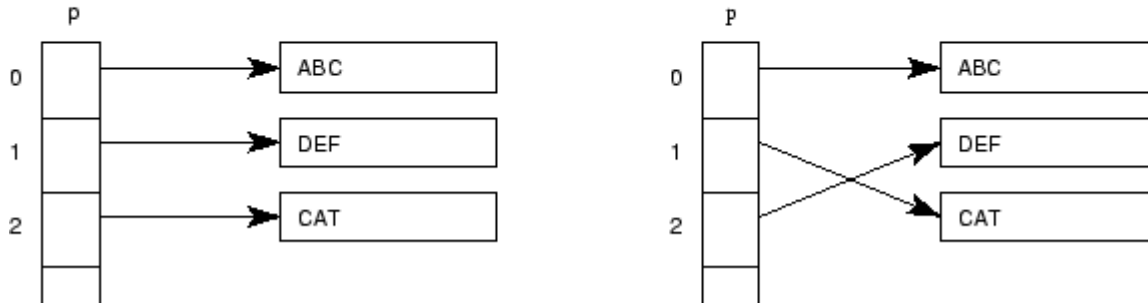
- Comparar dos líneas usando la función de la biblioteca estándar strcmp().

- Si dos líneas están desacomodadas -- intercambiar (swap) los punteros (no el texto).

```

ABC ..... \nDEF .... \n CAT .... \n ....
    
```

p[0] p[1] p[2]



Arreglos de punteros (Ejemplo de ordenamiento de cadenas).

Con lo anterior se elimina:

- el manejo complicado del almacenamiento.
- alta sobrecarga por el movimiento de líneas.

5.5. INICIALIZACIÓN ESTÁTICA DE ARREGLOS DE PUNTEROS

La inicialización de arreglos de punteros es una aplicación ideal para un arreglo estático interno, por ejemplo:

```

func_cualquiera()
{
    static char *nomb[] = { "No mes", "Ene", "Feb", "Mar", .... };
}
    
```

Donde, con el especificador de almacenamiento de clase *static* se reserva en forma permanente memoria el arreglo, mientras el código se está ejecutando.

5.6. PUNTEROS Y ESTRUCTURAS

Los punteros a estructuras se definen fácilmente y en una forma directa.

Considerar lo siguiente:

```

main()
{
    struct COORD { float x,y,z; } punto;
    struct COORD *ap_punto;
    punto.x = punto.y = punto.z = 1;
}
    
```

```

ap_punto = &punto; /* Se asigna punto al puntero */
ap_punto->x++;      /* Con el operador -> se acceden los elementos */
ap_punto->y+=2;      /* de la estructura apuntados por ap_punto */
ap_punto->z=3;
}

```

Otro ejemplo son las listas ligadas:

```

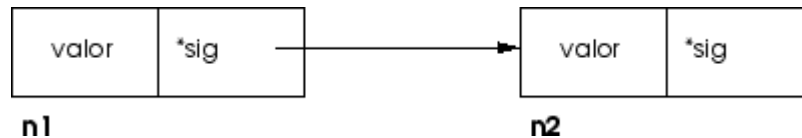
typedef struct {
    int valor;
    struct ELEMENTO *sig;
} ELEMENTO;

```

ELEMENTO n1, n2;

n1.sig = &n2;

La asignación que se hace corresponde a la figura



Esquema de una lista ligada con 2 elementos.

Nota: Solamente se puede declarar sig como un apuntador tipo ELEMENTO.

No se puede tener un elemento del tipo variable ya que esto generaría una definición recursiva la cual no está permitida. Se admite poner una referencia a un apuntador ya que los bytes se dejan de lado para cualquier apuntador.

Resumiendo, y dicho muy simplemente, un puntero es una dirección de memoria. Veamos el siguiente ejemplo:

```
#include <stdio.h>
```

```
main ()
```

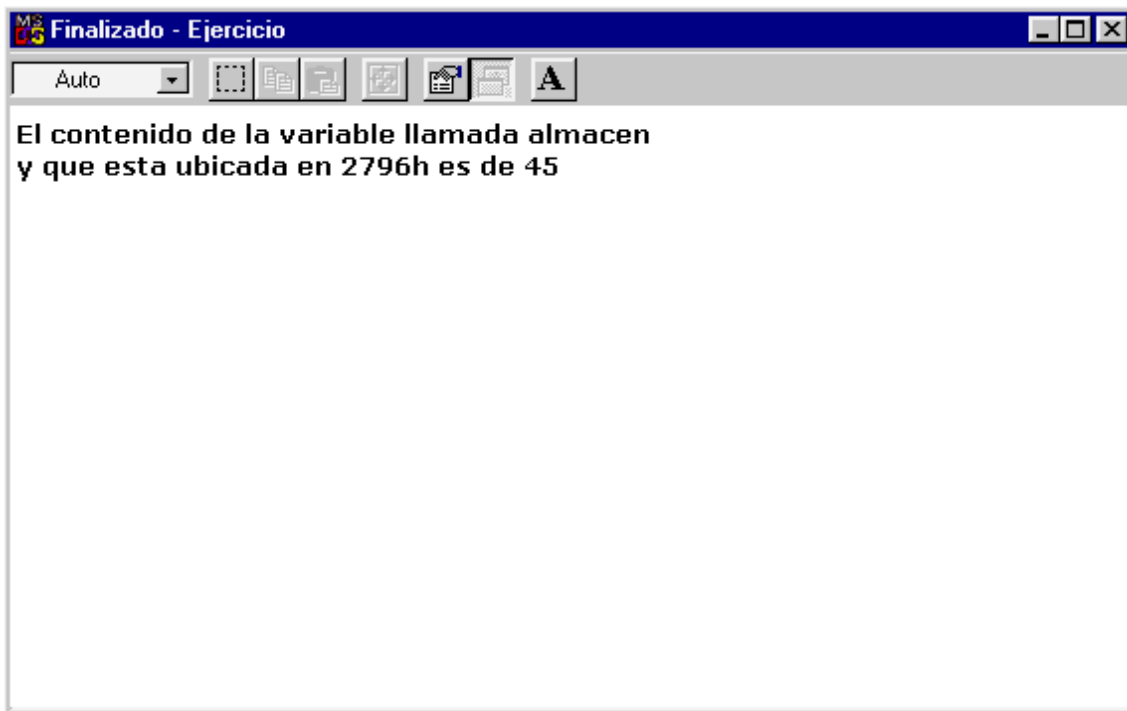
```
{
```

```
    /* Una variable entero y un puntero de tipo int */
```

```
int almacen, *puntero;

almacen=45;          /* Se asigna un valor cualquiera a la variable */
puntero=&almacen;     /* La dirección de almacén */
printf("El contenido de la variable llamada almacen\n"
"y que está ubicada en %xh es de %d\n", puntero, *puntero);
return 0;
}
```

Lo que muestra la pantalla cuando se ejecuta el programa es:



Con este ejemplo sencillo es posible ver cuál es la diferencia fundamental que existe entre un puntero (o dirección de memoria) y el contenido del mismo.

5.7. LISTAS SIMPLEMENTE ENLAZADAS

El siguiente tema con el que vamos a trabajar está relacionado con darle una utilidad práctica al uso de punteros. Para ello vamos a pensar en una estructura de almacenamiento dinámico muy básica: consta de un campo numérico (que por ejemplo puede ser int) y un campo puntero (donde este puntero apunta al

siguiente elemento de la estructura). Gráficamente se representaría de la siguiente forma:



Donde el primer elemento es un puntero que apunta al primer registro de la estructura. Entonces, cada elemento de la estructura dinámica está formado por un registro que tiene la siguiente forma:

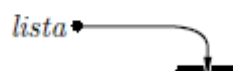
```
1 struct Nodo {
2     int info;
3     struct Nodo * sig;
4 };
```

El primer campo del registro (struct) es un campo entero (int) llamado info. El segundo campo es un puntero llamado sig que apunta a un registro con la misma forma del registro del que forma parte. Esto parece un trabalenguas o una definición cíclica, pero es de gran ayuda servirse del dibujo que representa la lista simplemente enlazada para darse cuenta de que el puntero del primer registro apunta a un segundo registro que tiene la misma forma que el primer registro. El campo puntero del último registro de la lista apunta a NULL. Si quisiéramos representar una lista vacía, su estructura se definiría de la siguiente manera:

```
int main(void)
{
    struct Nodo * lista = NULL;

    ...
}
```

Y gráficamente podría representarse de la siguiente manera:



Para insertar un primer registro a esa estructura lista que se encuentra vacía, se debe codificar de la siguiente forma:

```
int main(void)
{
    struct Nodo * lista = NULL;

    lista = malloc( sizeof(struct Nodo) );
    lista->info = 8;
    lista->sig = NULL;
    ...
}
```

Donde la función malloc() permite asignarle un espacio en memoria al nuevo registro. Una vez asignado el espacio se procede a la carga de los valores correspondientes al campo entero info y al campo puntero sig (en este caso se le asigna NULL porque se trata del último elemento de la lista).

Luego de ejecutarse esas instrucciones se tiene que la estructura toma la forma:



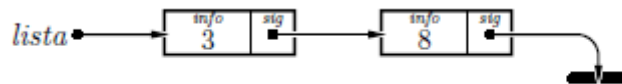
Si queremos acceder al campo info de la estructura indicada anteriormente deberíamos codificar:

*(lista).info

O

lista->info

Si ahora se desea agregar un nuevo elemento a la lista de modo que la misma quede de la siguiente forma –o sea un elemento al inicio de la lista-:

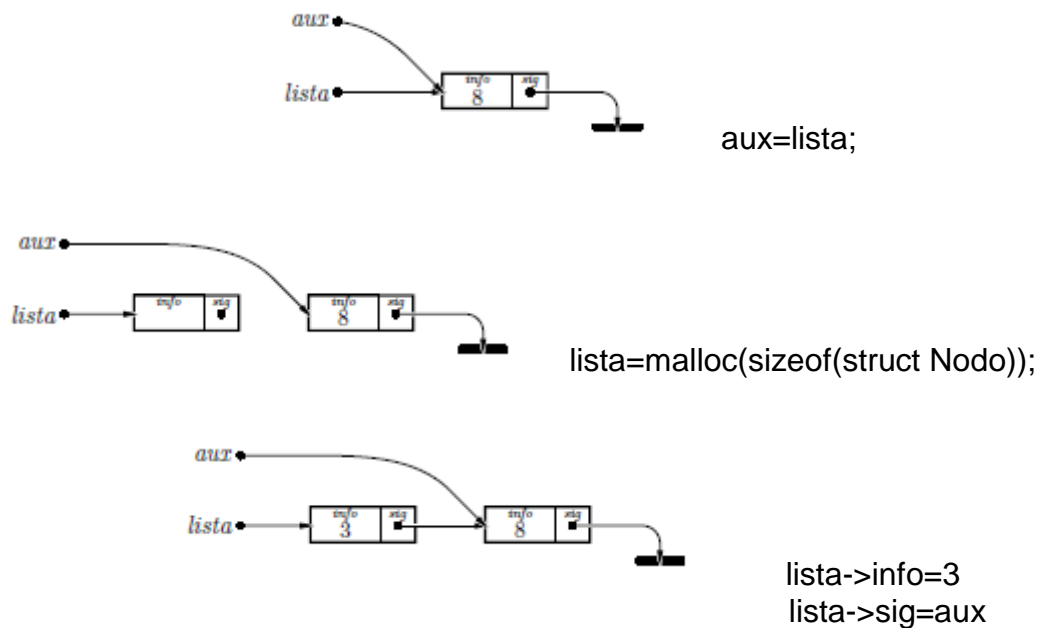


el código será el siguiente:

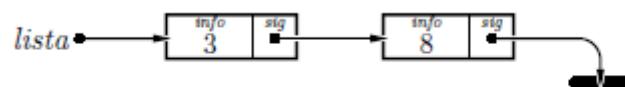
```
int main(void)
{
    struct Nodo * lista = NULL, * aux;

    ...
    aux = lista;
    lista = malloc( sizeof(struct Nodo) );
    lista->info = 3;
    lista->sig = aux;
    ...
}
```

Eso, abierto en los sucesivos pasos que se van dando sería:



Si ahora queremos agregar un nuevo elemento al final de la lista, tendríamos el siguiente esquema, a partir de esta lista:



Queremos lograr esto:



Las instrucciones necesarias serán:

```
int main(void)
{
    struct Nodo * lista = NULL, * aux;

    ...

    aux = malloc( sizeof(struct Nodo) );
    lista->sig->sig = aux;
    aux->info = 2;
    aux->sig = NULL;

    return 0;
}
```

5.8. LISTAS DOBLEMENTE ENLAZADAS

En algunas aplicaciones podemos necesitar recorrer la lista hacia adelante y hacia atrás. Las listas doblemente enlazadas nos permiten esa posibilidad.

Desde cualquier elemento podemos movernos hacia adelante o hacia atrás. Es por eso que se define un puntero al componente anterior y otro al posterior. La siguiente figura ilustra lo enunciado.



La definición de una estructura que contemple la forma que comentamos es la siguiente:

```
typedef struct celda{
    tipoelemento elemento;
    struct celda *siguiente,*anterior;
}tipocelda;
```

```
typedef tipocelda *posicion;
```

¿Cuáles son las acciones que requerimos más usualmente para trabajar con esta estructura?

Por ejemplo, un procedimiento para borrar un elemento en la posición p en una lista doblemente enlazada es:

```
void borrar (posicion p)
{
    if (p->anterior != NULL)
        p->anterior->siguiente = p->siguiente;
    if (p->siguiente != NULL)
        p->siguiente->anterior = p->anterior;
    free(p);
}
```

Esto muestra lo sencillo y complejo a la vez que es el direccionamiento sobre cada una de las celdas de lista si queremos hacer referencia a un elemento posterior o anterior desde una celda cualquiera hacia otra celda cualquiera.

Ejemplo de evaluación a distancia:

| | |
|--|----------------|
| ▶ Carrera: Analista en Sistemas Informáticos (Modalidad: IRSO VIRTUAL) | |
| ▶ Materia: Programación I ▶ Docente: Daniel Lombardero ▶ Alumno: ▶ DNI: | |
| Primera evaluación | Fecha: 20/6/11 |
| Fecha límite para la devolución | 4/7/11 |

Objetivo de la evaluación: Nuestro objetivo es conocer cuáles son los avances logrados a partir de los principales conceptos llevados a la práctica en la primera parte de la asignatura. La misma involucra especialmente la forma en la cual se escribe un programa; cuáles son las definiciones que deben hacerse en los mismos (encabezado, constantes, tipos, variables, cuerpo principal); las instrucciones básicas (su forma y uso); las estructuras básicas (variables complejas) que existen en C, vectores y matrices. Una de las cuestiones importantes es tener en cuenta cuál es el grado de integración de todos esos conceptos en la práctica (formas de pensar y escribir un programa como solución a un problema planteado).

¿Qué estamos evaluando?

- Conocer y utilizar las instrucciones básicas en el lenguaje C (if, for, while, etc)
- Estructurar correctamente un programa en C: definición del encabezado; adecuado uso de las declaraciones de constantes, tipos y variables; correcta identificación del cuerpo principal del programa.
- Utilizar diferentes tipos de variables. Conocer las diferencias entre los tipos predefinidos y los tipos definidos por el programador, diferentes clasificaciones de los mismos.
- Interpretar los enunciados de los problemas y generar una solución de calidad con los conocimientos básicos que se poseen del lenguaje.
- Manipular estructuras complejas como vectores o matrices.
- Integración de las diferentes herramientas, conceptos y estructuras en la resolución de las consignas planteadas.

Algunas recomendaciones:

Para la realización del trabajo, enumeramos algunas recomendaciones que le pueden ser de utilidad:

- Lea atentamente cada uno de los enunciados de nuestros problemas. Lea todo el trabajo práctico varias veces antes de comenzar a desarrollarlo.
- Identifique cuál o cuáles son las instrucciones básicas que más se acercan al manejo de los datos que debe hacerse para generar una adecuada solución al problema.
- Escriba cada uno de los programas con el mínimo de código posible.
- Para los desarrollos teóricos y los ejercicios sea claro y conciso.

Desarrollo de la evaluación

- 1:** ¿Qué es un tipo de datos? ¿Para qué sirve?
- 2:** Se desea hacer el siguiente cálculo: $11 * x / y$, donde x e y son enteros, y se ingresan por teclado. El resultado debe ser entero también e imprimirse en pantalla. Considerar que y sea distinto de cero, por cuanto está en el denominador de la división.
- 3:** Realizar un programa que permita cargar un vector de enteros de 15 posiciones, donde los índices también son enteros. Luego, indicar cuántos números pares e impares hay en el vector.
- 4:** ¿Qué características tiene un vector?
- 5:** ¿Cuál es la diferencia entre una matriz y un vector?
- 6:** Generar un programa que permita calcular lo siguiente: $5 * x + \frac{1}{4} * y$, donde x e y son valores reales que se ingresan por teclado. El resultado debe imprimirse en pantalla.
- 7:** Explicar para qué sirven scanf y printf.

Ejemplo de evaluación final:

FORMACIÓN DE ANALISTAS DE SISTEMAS INFORMÁTICOS PROGRAMACIÓN I.

Calificación:

Apellido y Nombre:

.....

D.N.I:

.....

Fecha:

Nº de Hojas:

Consideraciones Generales

- Intente responder en forma completa y sintética a lo solicitado.
- Se estima un máximo de 1 (una) hora y 40 minutos reloj para su realización.
- El examen se realizará sin material de apoyo o consulta.
- Se recomienda la lectura completa del examen antes de comenzar a responder.
- No se considerarán los borradores.

Calificación

- Se califica el examen de 1 (uno) a 10 (diez).
- Se aprueba con un puntaje mínimo de 4 (cuatro) puntos
- Deben responderse todas las preguntas, es decir, no se puede aprobar el examen con alguna pregunta sin responder aunque con las resueltas se llegara a la calificación de 4 (cuatro) o más puntos.
- En cada pregunta se indica la puntuación de la misma entre paréntesis.

Criterios de evaluación

- Correcta interpretación de los enunciados.
- Pertinencia de las respuestas.
- Aplicación de conceptos teóricos y prácticos trabajados.
- Coherencia en la respuesta de los enunciados.
- Comprensión de los núcleos esenciales de los contenidos.
- Utilización correcta de las instrucciones en C.
- Manejo adecuado de la programación estructurada.
- Conocimiento de los diferentes tipos y estructuras de datos con los que cuenta el lenguaje.

Consignas para desarrollo

1 (1,5 punto): Generar un programa que permita realizar el siguiente cálculo:

$a+147*b+253$, donde:

a y b son números reales que se ingresan por teclado

Considerar que a y b deben ser pares, ésa es la única restricción. En el caso de que alguno no sea par, se debe devolver un mensaje indicando "No se puede realizar el cálculo porque alguno de los números ingresados es par".

2 (1,5 punto): Indique qué son los parámetros, para qué sirven.

3 (3,5 puntos): Dado un vector de 1040 posiciones donde sus índices son enteros y sus datos reales (que se leen de teclado), se quiere generar un archivo de reales donde se copian los datos del vector. Para ello deberá hacer un programa que contenga al menos tres procedimientos: uno para leer los datos de teclado y cargarlos en el vector, otro para tomar los elementos del vector y copiarlos en el archivo, y un tercero para imprimir los valores almacenados en el archivo e imprimirlos en pantalla.

4 (0,5 punto): ¿Qué hace la siguiente función?

```
int f_vec(int v_cantidad,int *v_vector )
{
    int v_index;
    for(v_index=0;v_index<v_cantidad;v_index++)
    {
        v_vector[v_index]=v_index;
    }
    return 1;
}
```

5 (0,5 punto): Defina qué es un archivo y cuáles son sus características/propiedades más importantes. ¿Para qué sirven en la práctica? ¿qué diferencia hay en la definición de un archivo de texto y uno de enteros? Defina un ejemplo de cada uno de ellos.

6 (0,5 punto): ¿Para qué sirve el prototipo de una función? Dar un ejemplo.

7 (2 puntos): Realizar una función que devuelva el mayor de dos números reales que se pasan por teclado.

GLOSARIO

A

Abrir un archivo.- Identificación de un archivo para conocer sus atributos.

Algoritmo.- Conjunto de pasos ordenados para resolver un problema. Los términos Algoritmo y Lógica son sinónimos (algorithm).

Análisis.- Examinar y comprender un problema para encontrar una solución óptima.

Archivo.- Conjunto de registros afines tratados como una unidad de almacenamiento de datos; por ejemplo, archivos de direcciones, archivos de agenda, etc. También se les conoce como ficheros o files.

Arreglo.- Recorridos o accesos secuenciales de un vector o matriz. Sus elementos se almacenan individualmente en celdas de memoria.

Array.- Vea Arreglo.

Asignación.- Dar valor a un identificador de variable o constante. Paso de parámetros hacia una variable.

B

Bucle.- Repetición dentro de un programa.

Buffer.- Segmento de memoria reservado para el almacenamiento de datos mientras estos son procesados.

Byte.- Unidad de almacenamiento de información comúnmente utilizado en computadoras. Está formado por ocho bits (Binary digiT) mas un bit extra de paridad.

C

Caché.- Banco dedicado de memoria de alta velocidad o una sección reservada de la memoria regular que se utiliza para mejorar el desempeño. (Ver Memoria Caché).

Codificación.- Expresar el algoritmo como un programa en un lenguaje de programación adecuado.

Código espagueti.- Código escrito sin seguir una estructura coherente. La lógica puede moverse de rutina en rutina sin regresar al punto de origen (salida desde la que se mandó llamar a otra rutina). Usualmente el uso de la instrucción GOTO es la que permite este tipo de programación.

Compilador.- Software capaz de traducir programas de alto nivel en un nivel de representación mas bajo. Usualmente genera primero un código ensamblador para después traducirlo en lenguaje máquina.

Computadora.- Máquina de propósito general que procesa información de acuerdo a un conjunto de instrucciones que están almacenadas internamente, de forma temporal o permanente. A la computadora y a todo su equipo se le conoce como Hardware (también llamado ordenador). Software son las instrucciones que le dicen a una computadora que hacer. El conjunto de instrucciones que realiza un tarea en particular es llamado programa.

Constante.- Es un dato que permanece sin cambios durante el desarrollo de un algoritmo y la ejecución de un programa. Ver Tipos de Constantes.

CPU.- Siglas para "Central Process Unit" (Unidad Central de Proceso, UCP).

D

Data.- Vea Datos.

Datos.- Cualquier forma de información ya sea en papel o en forma electrónica. Esta última se refiere a archivos y bases de datos, documentos de texto, imágenes y voz y video digitalizados.

Declaración.-

1. CASE.- Variación de la declaración IF-THEN-ELSE. Se utiliza cuando el número de condiciones es muy grande.
2. IF-THEN-ELSE.- Permite la comparación entre 2 o mas condiciones de datos. Si el resultado es verdadero, entonces (THEN) realiza una acción a seguir; en caso contrario (ELSE) permite llevar a cabo otra acción distinta.

Debug.- Encontrar errores en un programa lógico o computacional. Usualmente se revisa el código línea por línea.

Diagrama de Flujo.- Es una representación que utiliza símbolos unidos por flechas denominadas líneas de flujo, que indican la secuencia en que se deben ejecutar. Dentro de cada símbolo o caja se indican las operaciones que deben ser realizadas.

Diagrama N-S (Nassi-Schneiderman)..- También conocidos como Diagramas de Chapin. Es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas, y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Documentación de Programas.- Descripción narrativa y gráfica de un programa computacional. Se usa para describir cada paso realizado por el

programador, desde la etapa de análisis hasta la etapa de mantenimiento de un programa.

E

EAPROM.- Memoria de sólo lectura alterable electrónicamente. Puede ser borrada y programada por el usuario. Ver Tipos de Memoria.

EEPROM.- Memoria electrónicamente borrrable (por medio de luz ultravioleta). Ver Tipos de Memoria.

Ejecución de programa.- Lectura y realización de las instrucciones que forman un programa (Run, correr el programa), se realiza en forma secuencial.

EPROM.- Memoria PROM borrrable. Se puede eliminar la información programada y volverla a programar. Ver Tipos de Memoria.

Entrada.- Información que se introduce a un programa.

Enunciado.- Unidad mínima que se puede ejecutar. También llamado instrucción, son los comandos que componen el programa.

Expresiones.- Son conjuntos de constantes, variables, operadores y paréntesis. Normalmente se utilizan para definir operaciones matemáticas. Consta de operandos y operadores.

Expresiones Aritméticas.- Son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas y las operaciones son las aritméticas. Los operadores más comunes son:

| Símbolo | Significado |
|---------|-----------------|
| () | Paréntesis |
| ^ | Exponenciación |
| DIV | División entera |
| MOD | Módulo |
| * | Multiplicación |
| / | División |
| + | Suma |
| - | Resta |
| = | Asignación |

Expresiones lógicas.- Formato: Operador lógico . Ver Operadores lógicos

Expresiones Relacionales.- Permiten realizar comparaciones de valores de tipo numérico o carácter. Sirven para expresar las condiciones en algoritmos o programas. Formato: Operador relacional . Ver Operadores relacionales.

F

Fichero.- Vea Archivo.

File.- Vea Archivo.

H

Hardware.- Maquinaria y equipo (CPU, discos, cintas, módem, cables, etc.). En operación, la computadora es tanto software como hardware, uno es inservible sin el otro.

I

Indexación.- Técnica de acceso a los datos en base a la utilización de un índice o un registro de índices.

Indexar.- Asociar un índice a un dato para su acceso.

Información.- El conjunto de los datos.

Instrucción.- Elemento básico constitutivo de los programas. Una instrucción es una formulación de una orden a la computadora y que se manifiesta en la expresión del conjunto de operaciones que debe ejecutar la computadora.

Intérprete.- Software que traduce programas fuente a programas máquina. La traducción se realiza instrucción por instrucción cada vez que el programa es ejecutado.

L

Lenguaje de alto nivel.- Permite generar código máquina a partir de sentencias independientes de la máquina y representar problemas de un usuario de modo simple y claro. Una sentencia en lenguaje evolucionado se traducirá con frecuencia en una serie de instrucciones máquina o de subprogramas en lenguaje máquina.

Lenguaje de bajo nivel.- Lenguaje de programación en código máquina o en el que a cada instrucción simbólica le corresponde una en código máquina equivalente. El lenguaje de bajo nivel depende de la máquina.

Lenguajes de Programación.- Un **lenguaje de programación** es un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Es utilizado para controlar el comportamiento físico y lógico de una máquina.

Aunque muchas veces se usan los términos 'lenguaje de programación' y 'lenguaje informático' como si fuesen sinónimos, no tiene por qué ser así, ya que los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como, por ejemplo, el HTML.

Un lenguaje de programación permite a uno o más programadores especificar de *manera precisa* sobre qué datos debe operar una computadora, cómo estos datos deben ser almacenados o transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar *relativamente* próximo al lenguaje humano o natural, tal como sucede con el lenguaje Léxico. Una característica relevante de los lenguajes de programación es precisamente que más de un programador puedan tener un conjunto común de instrucciones que puedan ser comprendidas entre ellos para realizar la construcción del programa de forma colaborativa.

Los procesadores usados en las computadoras son capaces de entender y actuar según lo indican programas escritos en un lenguaje fijo llamado lenguaje de máquina. Todo programa escrito en otro lenguaje puede ser ejecutado de dos maneras:

- Mediante un programa que va adaptando las instrucciones conforme son encontradas. A este proceso se lo llama *interpretar* y a los programas que lo hacen se los conoce como intérpretes.
- Traduciendo este programa al programa equivalente escrito en lenguaje de máquina. A ese proceso se lo llama *compilar* y al programa traductor se le denomina compilador.

Lenguaje máquina.- Las instrucciones se especifican en binario, de tal forma que sean cargadas directamente en memoria y ejecutables. Es el lenguaje de más bajo nivel que permite utilizar una computadora y es el único lenguaje que sabe interpretar la unidad de control de la misma.

Lista.- Arreglo de datos. Usualmente se utiliza su formato es de columna o renglón.

Lista doblemente enlazada.- Una lista cuyos nodos utilizan punteros para permitir el movimiento hacia delante y hacia atrás de un nodo perteneciente a la estructura lista.

M

Memoria.-Área de trabajo de la computadora (físicamente, una colección de chips de RAM). Es un recurso muy importante

Memoria Caché.- Es el banco de memoria que conecta la memoria principal y el CPU. Es más rápida que la memoria principal y permite que las instrucciones sean ejecutadas y que los datos sean leídos a una velocidad más alta.

Métodos de búsqueda.- FIFO(First In First Out), LIFO (Last In First Out),

N

Nodo.- Punto o vértice de un árbol o grafo; normalmente representa un dato.

O

Operaciones.- Se clasifican en aritméticas (números), lógicas (falso, verdadero), relacionales ($a > b$, $a < > b$) y de carácter.

Operador.- Símbolo especial o palabra reservada que especifica una operación aritmética o lógica.

Operador booleano o lógico.- Operador que combina expresiones lógicas utilizando aritmética lógica. Los más comunes son: AND, OR y NOT.

Operadores Aritméticos.- Pueden variar según el lenguaje de programación a utilizar. Usualmente, el orden de prioridad más común es el siguiente:

Operadores Lógicos.-

| Operador | Expresión | Resultado | |
|----------|-----------|------------------------------|---------------------|
| NO, NOT | NO [A] | Negación de A (lo contrario) | Ver Tabla de verdad |
| Y, AND | [A] Y [B] | Unión de A y B | Ver Tabla de verdad |
| O, OR | [A] O [B] | Intersección entre A y B | Ver Tabla de verdad |

Operadores Relacionales.-

| Símbolo | Significado | Símbolo | Significado |
|---------|-------------|---------|-----------------|
| > | Mayor que | <> | Diferente a |
| < | Menor que | >=, => | Mayor o Igual a |
| = | Igual a | <=, =< | Menor o Igual a |

Ordenador.- Vea Computadora.

P

Palabra reservada.- Palabras utilizadas por los lenguajes de programación para representar variables, sentencias u órdenes.

Pseudocódigo .- Es un lenguaje de especificación de algoritmos y no puede ser ejecutado por una computadora

Pila.- Estructura dinámica de datos en la que los elementos se añaden y eliminan por un extremo.

Procedimiento.- Subrutina o Función.

Procesamiento de Datos.- Capturar, guardar, actualizar y regresar datos e información.

Programa ejecutable.- Programa compilado y traducido a código máquina que puede ser ejecutado directamente por una computadora.

Programa fuente.- Programa escrito en lenguaje de alto nivel. Requiere de compilación o interpretación para su ejecución.

Programa objeto.- Programa en lenguaje máquina resultante de la traducción del programa fuente .

Programación No-numérica.- Programación con objetos (palabras, personas, cosas).

Programación orientada a objetos.- Basado en la programación modular. Maneja los conceptos de encapsulamiento, herencia y poliformismo.

Programación visual.- Construcción de programas mediante el uso de herramientas que permiten desarrollar programas utilizando plantillas con instrucciones preestablecidas. No es necesario escribir código.

Programador.- Persona encargada de diseñar la lógica de un programa y escribirla utilizando líneas de código para un programa computacional.

PROM.- Memoria programable (irreversible). Programada por el usuario o comerciante, una vez programada esta memoria no puede modificar su contenido. Ver Tipos de Memoria.

R

RAM.- Memoria de Acceso Aleatorio (volátil, lectura - escritura). Ver Tipos de Memoria.

Registro.- Estructura de datos formada por diferentes campos.

Reglas de Prioridad para expresiones aritméticas.-

1. Evaluar todas aquellas expresiones que estén encerradas entre paréntesis. Comenzando con el paréntesis más interno y terminando con el más externo.
2. Las expresiones se evalúan con orden izquierda a derecha (notación polaca inversa), siguiendo el orden de prioridad de los

operadores aritméticos, según el lenguaje de programación a utilizar (ver operadores aritméticos).

ROM.- Memoria de Sólo Lectura (irreversible). Ver Tipos de Memoria.

S

Salida.- Información generada por un programa computacional. Transferir o transmitir desde la computadora hasta un dispositivo periférico o una línea de comunicación.

Sistema operativo (S.O.).- Programa encargado del control general de la computadora. La parte principal del S.O., llamada Kernel, es colocada en la memoria al encender la máquina para monitorear cada operación realizada.

Software.- Instrucciones computacionales almacenadas en un programa.

Subproblemas.- División de un problema en problemas más pequeños y simples de resolver.

Subrutina.- Instrucciones que realizan operaciones específicas. Son llamadas mediante módulos o procedimientos.

T

Tabla de Verdad de NO (NOT).- Comportamiento.

| A | NO A |
|-----------|-----------|
| Falso | Verdadero |
| Verdadero | Falso |

Tabla de Verdad de O (OR).- Comportamiento.

| A | B | (A) O (B) |
|-----------|-----------|-----------|
| Falso | Falso | Falso |
| Falso | Verdadero | Verdadero |
| Verdadero | Falso | Verdadero |
| Verdadero | Verdadero | Verdadero |

Tabla de Verdad de Y (AND).- Comportamiento.

| A | B | (A) Y (B) |
|-------|-----------|-----------|
| Falso | Falso | Falso |
| Falso | Verdadero | Falso |

| | | |
|-----------|-----------|-----------|
| Verdadero | Falso | Falso |
| Verdadero | Verdadero | Verdadero |

Tipos de Datos.- Simple, estructurados o abstractos; definidos por el usuario o por el sistema; sólo existen mientras el programa es ejecutado; manipulados por el hardware o el software.

Pueden ser constantes, variables y estructuras que contienen números (enteros y reales), texto (caracteres y cadenas) o direcciones (de variables y estructuras). Un dato tiene un contenido (valor) y un tipo asociado. Por esta razón algunos lenguajes de programación son conocidos como Strongly Typed (tipos fuertes). El tipo de un dato determina: el rango de valores válidos, el conjunto de operaciones válidas, el espacio de memoria que ocupa que ocupa el dato y la representación e interpretación del dato binario.

Permiten al compilador detectar errores, limitar la mezcla de tipos y evaluar expresiones de cualquier grado de complejidad.

Mezcla de diferentes tipo de datos.- En general cuando una operación involucra diferentes tipos, ambos tipos se convierten internamente en lo que se denomina como tipo común. El tipo común es un tipo interconstruido con el mínimo valor posible para almacenar la unión de los valores de ambos tipos.

Principales tipos de datos:

| Tipo de datos | Intervalo | Byte |
|--------------------------|----------------------------|---------------|
| Booleano | False o True | 1 |
| Char | Carácter del código actual | 1 |
| Byte | 0 .. 255 | 1 |
| Shortint, Short | -128 .. 127 | 1 |
| Integer, int | -32,768 .. 32,767 | 2 8 y 16 bits |
| Integer, int | 2,147,483,647 | 4 32 bits |
| Word, cardinal | 0 a 65,535 | 2 |
| Longint, Long | 2,147,483,647 | 4 |
| Single | 1.5e-45 .. 3.4e38 | 4 |
| Real, | 2.9e-39 .. 1.7e38 | 6 |
| Double, double precision | 5.0e-324 .. 1.7e308 | 8 |
| Comp | -9.2e18 .. 9.2e18 | 8 |
| Extended | 3.4e-4932 .. 1.1e4932 | 10 |
| String | Cadena de caracteres | 255 |
| Bitset | 0 a 15 | |
| Float | | |

| | | |
|------------------|--|------------------|
| Signed | | |
| Unsigned | | |
| 1 Bit = 0,1 | 1 Byte = 8 Bits | 1Mb = 1024 Bytes |
| 16 Bits = 1 Word | 32 Bits = Double Word (Procesadores 80386) | |

Tipo de datos ordinales.- Los tipos de datos ordinales son todos aquellos tipos que no sean reales (integer, longint, shortint, byte, word, char, enumeraciones y subrangos).

Probablemente, el tipo de datos más útil es longint (abreviatura en inglés de long integer o entero largo), este tipo de datos maneja valores de hasta 2,147,483,647 en vez de 32,767 del tipo integer. Por ejemplo, para almacenar el producto de 2,000 * 2,000 (donde el resultado es cuatro millones) se debe utilizar una variable de tipo longint en vez de una de tipo integer.

Usualmente es recomendable utilizar integer en vez de longint debido a que este último tipo utiliza más memoria, además de reducir significativamente la velocidad de procesamiento. Por lo tanto, cuando se está seguro de que el uso de un integer será suficiente, es mejor que este sea utilizado.

Tipos de constantes.-

| Tipo | Ejemplo |
|--------------------------|----------------------------------|
| a) Entera: | -1, 0, 25 |
| b) Reales: | 0.002, 3.14159, -2.34567, 5432. |
| c) Carácter: | 'a', 'A', '+', ',', '9' |
| d) Cadena de caracteres: | 'A1', 'Hola', 'PRIMER ALGORITMO' |
| e) Lógicas: | falso, verdadero; 0, 1; Si, No |

Tipos de memoria.-

| Tipo | R (lectura) | W (escritura) | Irreversible |
|--------|-------------|-------------------------|--------------|
| RAM | X | X | |
| ROM | X | | X |
| PROM | X | | X |
| EPROM | X | X | |
| EAPROM | X | X (electrónicamente) | |
| EEPROM | X | X (electrónicamente) | |

Tipos de variables.- Los tipos más comunes son: enteras, reales, carácter, cadena y lógicas.

U

Unidades de almacenamiento.- Disco duro (Hard Disk), disquete (Floppy Disk), discos compactos (CD-ROM), Disco láser, DVD (Digital Video Disk), DVCR (Digital Video Cassette Recorder), cinta magnética (DAT Tape Cartridge), disco óptico (Magneto-Optic Cartridge), entre otros.

Unidades de E/S de datos (Entrada/Salida o Input/Output).- Permiten la entrada o salida de información al CPU.

- Entrada: teclado, scanner, mouse (ratón), mouse óptico, lápiz óptico.
- Salida: Impresora impacto (matricial, tambor, margarita), impresora no-impacto (electrostática, tinta, magnetográfica, termal, láser,), monitor, data show (Liquid Cristal Display panel), bocina, plotter (impresora gráfica).

UCP.- Vea CPU (Unidad Central de Proceso).

V

Validar.- Verificar que los datos arrojados por el programa sean correctos.

Variable.- Localidad que almacena un valor modificable a lo largo de la ejecución de un programa.

Variable (otras definiciones).- Dato que puede cambiar durante el desarrollo del algoritmo y la ejecución de un programa. Se identifican mediante los atributos Nombre (de la variable) y Tipo (de variable). Cada variable puede ser únicamente declarada de un tipo, el cual puede definirse o declararse antes de su utilización. También se les conoce como identificador. Ver Tipos de Variables.