Lenguajes de Programación

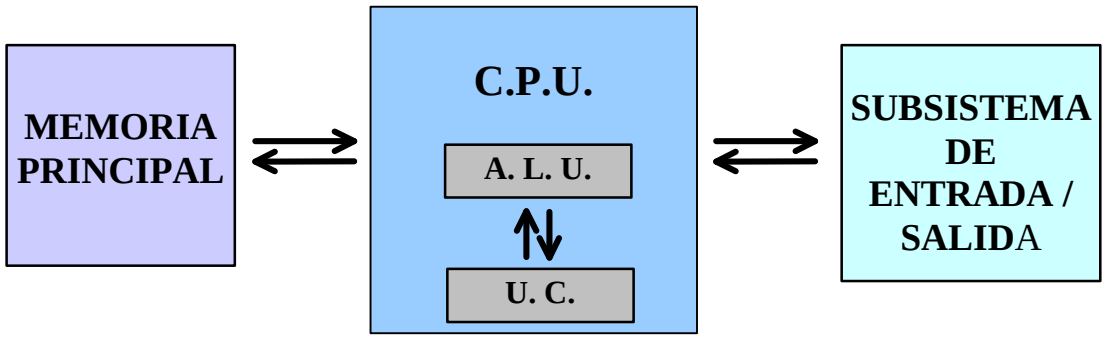
Introducción

En este documento se hace una breve introducción a los distintos tipos de lenguajes de programación existentes, como también así a los procesos de traducción hacia un código entendible por un sistema de computación.

1. Arquitectura de Computadoras

Arquitectura de Von Neumann (fines de ’40): computadora compuesta por las siguientes unidades funcionales:

- Memoria: almacena datos e instrucciones.
- Unidad Aritmético-Lógica: opera con datos.
- Unidad de Control: interpreta y ejecuta instrucciones.
- Subsistema de Entrada-Salida.



Las *computadoras convencionales* están basadas en la *arquitectura de Von Neumann*. Esto implica que tienen la siguiente característica de funcionamiento:

- La ejecución de control del programa está dada por flujo de instrucciones, el cual es secuencial (de hecho las instrucciones de salto condicional o incondicional son un salto hacia otra secuencia)

- El estado de la memoria puede ser modificado por medio de un cambio en el contenido de una celda

2. Paradigmas de Programación

Se llama así a conceptualizaciones de técnicas y lenguajes empleados para construir programas. Existen distintos paradigmas de programación, y pasamos a enumerarlos y caracterizarlos:

- **Imperativo**: Un programa es una secuencia de alteraciones de estados (datos) localizados en un espacio de direccionamiento (la memoria).

- **Funcional**: Un programa puede ser descripto como funciones entre conjuntos.

- **Lógico**: Un programa puede describirse definiendo ciertas relaciones sobre un conjunto de datos a partir de los cuales otras pueden ser calculadas empleando reglas de deducción

Sintaxis

La sintaxis de un lenguaje especifica cómo construir los programas en dicho lenguaje. La sintaxis sirve tanto para el traductor (compilador, etc.), como para los programadores.

Todo lenguaje de programación posee formas sintácticas para poder expresarse en el mismo.

Semántica

Expresa lo que hace determinado código.

Los programas realizados por programadores deben poseer una semántica clara para que sean fáciles de entender, modificar y adaptar.

Característica del Paradigma Imperativo:

Los programas de este paradigma se construyen a partir del concepto de la ***asignación***, y las estructuras de control ***permiten coordinar el flujo de control*** (cambiar secuencias de instrucciones). Las formas de control que ofrecen éstos lenguajes son: secuencia, selección, iteración y corte de control incondicionales (***goto*** o variantes).

Obviamente, el paradigma imperativo es el que mejor se adapta al modelo de computadora convencional (Arquitectura de Von Neumann). Por este motivo es que existen tantos lenguajes para este tipo de paradigmas: Basic, Fortran, C, Pascal, ADA, Algol 60, Algol 68, Modula-2, etc.

Sin embargo, esta perfecta adaptación no es gratuita, ya que la eficiencia se logra a expensas de:

- La presencia de la asignación, que hace que el valor denotado por una variable sea **dependiente del lugar que ocupa en el programa.** Esto implica que para conocer el significado de una parte del programa puede ser necesario conocer todo el resto. (Recordar el contenido de los registros y de las direcciones de memoria en Z80).
- La existencia de formas **sintácticas de semántica poco clara**. Por ejemplo, resulta complejo deducir el efecto de un programa escrito con ***goto***.

Para mejorar los problemas que surgen de lo antes mencionado, se establecieron ciertas técnicas:

- **Modularización**: subdividir el problema en sub-problemas de menor complejidad. **Cada módulo puede desarrollarse por separado y combinarse en base exclusivamente a su semántica, desconociendo su estructura interna**.

- **Programación estructurada**: cada módulo debe tener un único punto de entrada y un único punto de salida, para evitar acoplamientos. (Recordar que en Z80 por medio de ***call*** siempre se salta al comienzo de la subrutina y que la única forma de salir de ella es con ***ret***)

- **Desalentar el uso de saltos incondicionales *(goto y sus variantes)***

3. Génesis de los Lenguajes de Programación

3.1 Lenguaje de Máquina

Como ya sabemos, la computadora sólo entiende **Lenguaje de Máquina.** Cada tipo de computadora tiene un **lenguaje de máquina diferente** (unos y ceros), implementado por el fabricante.

Desventajas

- Realizar programas en lenguaje de máquina resulta más que tedioso.

- Si se desea llevar el mismo programa para que funcione a una computadora con distinta arquitectura, se debe re-escribir el código (no es reusable)

3.2 Lenguaje Ensamblador

Tratando de solucionar la primera desventaja del lenguaje de máquina, surge el **Lenguaje Ensamblador** (Assembler), que permite escribir un código con un cierto nivel de abstracción: por lo menos se pueden definir identificadores para las zonas de memoria, y se pueden definir subprogramas y macros.

Obviamente se necesitó introducir un "traductor", llamado ensamblador para poder traducir el programa fuente escrito en lenguaje ensamblador al lenguaje que interpreta la máquina.

Ventaja:

- Mayor nivel de abstracción (recordar la ventaja de programar en Z80 en vez de código binario)

Desventajas

- Realizar programas en lenguaje ensamblador sigue siendo complejo

- El código ensamblador tampoco es portable (sigue siendo relación uno a uno con el lenguaje de máquina)

3.3 Lenguajes de Alto Nivel

Al extenderse el uso de las computadoras a diferentes áreas, aparecieron problemas cada vez más complejos, complicados de por sí lo suficiente como para requerir herramientas más sencillas para expresar la solución de los mismos. Surgen así los **Lenguajes de Alto Nivel,** que permiten escribir código con mayor nivel de abstracción. Obviamente cada instrucción en lenguaje de alto nivel no necesariamente se corresponde con una única instrucción en lenguaje Assembler, en muchos casos se traduce en varias. Es claro que se necesitó disponer de un traductor (compilador o intérprete) para poder traducir el programa fuente en un lenguaje que finalmente fuera interpretado por la máquina.

Ejemplos de lenguaje de alto nivel: Basic, Logo, Pascal, C, Fortran, Modula, C++, Smalltalk, Java, etc.

Ventajas:

- Mayor nivel de abstracción, respecto de los anteriores. Existe una variedad de lenguajes de alto nivel que se orientan a determinados propósitos: algunos están orientados a cálculos científicos, otros para propósito general, etc.

- Mayor portabilidad para los programas de usuarios, ya que sólo basta con recompilar el código en alto nivel al lenguaje de máquina correspondiente

(utilizando el traductor para dicha máquina). En Assembler esto no se podía hacer, ya que las instrucciones entre una máquina y otra podían no coincidir, por lo tanto había que re-escribir todo el código (Por ejemplo: la carga de un registro en una arquitectura Z80 se obtiene por la instrucción ld A,5 pero en la x86 se obtiene por mov AX,5)

Desventaja:

- El conjunto de instrucciones en que se traduce una instrucción de alto nivel, puede ser menos eficiente que si lo hubiera pensado directamente en lenguaje ensamblador **un experto**. No hay una relación uno a uno entre una instrucción de lenguaje alto nivel y una de lenguaje de máquina. Por este motivo se sigue programando en Assembler para protocolos de comunicaciones, drivers para periféricos, partes de un sistema operativo, parte de un motor de base de datos, etc.

4. Traductores de Programas

Todo programa fuente no escrito en lenguaje de máquina debe pasar por un traductor, para poder ser entendido por la máquina.

Programa Fuente Þ TRADUCTOR Þ Programa en Lenguaje de Máquina

Existen distintos tipos de traductores:

- **Compiladores** (Ejemplos: C, Pascal.)
- **Intérpretes** (Ejemplos: Smalltalk, Java)

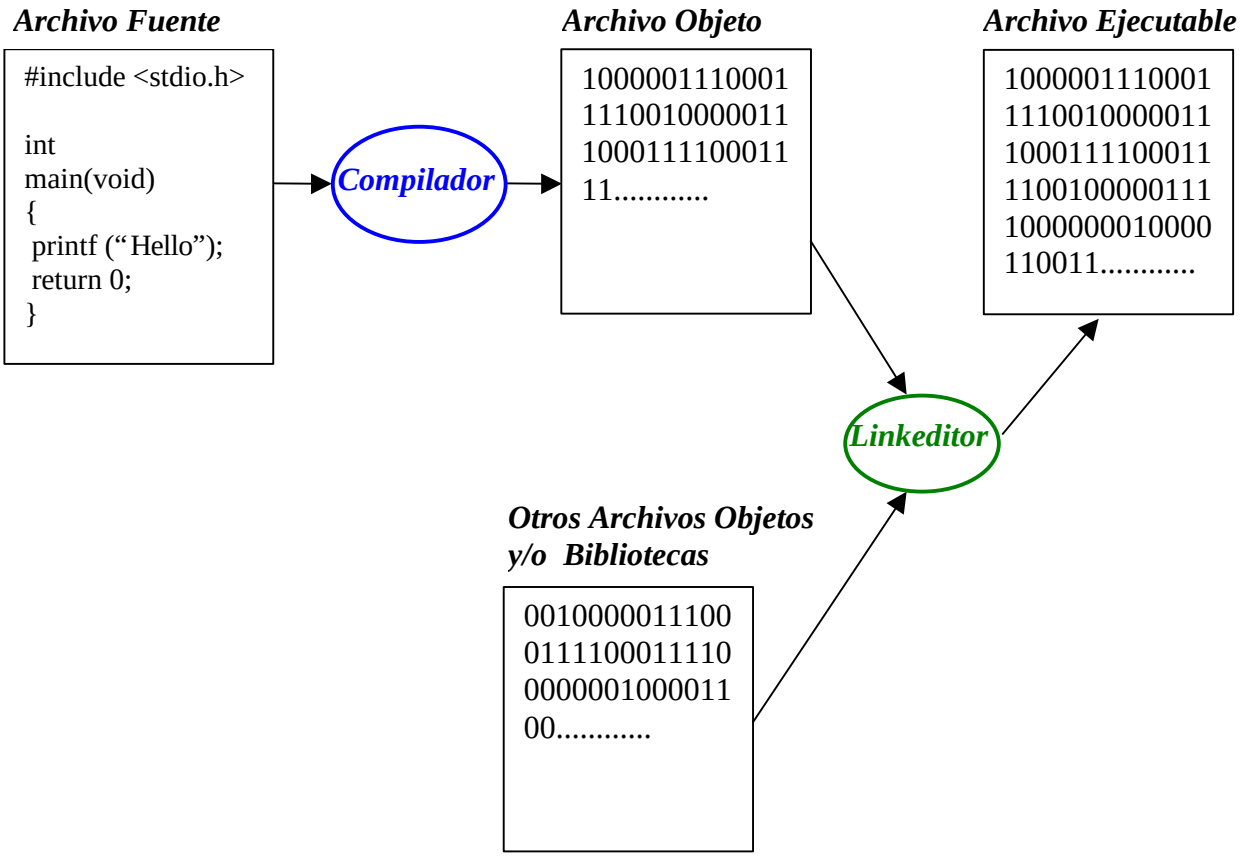
4.1 Compiladores

El código de un programa se almacena en un archivo denominado programa fuente. Este es el único archivo de todo el proceso de traducción que contiene líneas que el humano pueda leer y entender.

Una vez que se tiene el programa fuente, el próximo paso a ejecutar es usar el compilador para traducir el programa fuente en un formato que la computadora pueda entender directamente. Este proceso varía de una máquina a otra. El compilador traduce el archivo fuente en un segundo archivo llamado archivo objeto, que contiene instrucciones apropiadas para el sistema de computación.

Dicho archivo objeto puede combinarse con otros archivos objetos o bibliotecas, generando un archivo ejecutable que pueda correr en el sistema. El proceso de combinar todos los archivos objetos individuales en un ejecutable final se llama **linkedición**.

En algunos sistemas todos estos pasos individuales ocurren sin la participación del programador, es decir los pasos se llevan a cabo automáticamente, y en otros hay que hacer explícitamente paso por paso.



4.2 Intérpretes

Los intérpretes no generan un programa ejecutable, sino un programa que puede ser interpretado en una "maquina virtual". O sea, el código que generan no es lenguaje de máquina para la arquitectura en la que están ejecutando, sino un código intermedio que cuando se intenta ejecutar va siendo interpretado paso a paso. Suelen ser más lentos pero más portables. Ejemplos: Smalltalk, Java.

Desarrollo y Testeo de Software

Introducción

La Ingeniería de Software es la disciplina de la Ciencia de la Computación que se ocupa de las técnicas necesarias para el desarrollo y mantenimiento de Sistemas de Software (inclusive a gran escala).

En el presente documento se desarrolla una serie de temas que introducen pasos importantes para el desarrollo y testeo de software.

1. Técnicas para el Desarrollo de Software

El sistema deberá ser desarrollado adaptándose a las necesidades de los usuarios y teniendo en cuenta que será utilizado por diferentes usuarios en distintas computadoras.

Es deseable que el software:

- pueda correr en distintos entornos (portabilidad)
- se adapte a las necesidades de los usuarios (especificadas a través de un relevamiento)
- surja de un análisis y diseño (siendo así fácil de mantener y adaptar)
- se implemente con algoritmos correctos y eficientes
- funcione correctamente (testeo de software y debuggeo)

1.1 Portabilidad de un Programa

Un programa es portable si se adapta fácilmente a nuevos entornos sin tener que re-escribirlo. Esto implica que el programa es desarrollado sin asumir características dependientes del entorno donde ejecutará el mismo (no se utilizan cuestiones específicas de la arquitectura de la computadora, ni de hardware ni de software).

La ventaja de un programa "portable" reside en que puede ejecutarse en distintas plataformas con "solo recompilarlo" (sin cambiar ninguna línea de código). Así es como se tiene la potencia de un programa "multiplataforma" con solo un desarrollo.

Cuando nos referimos a distintas plataformas estamos señalando plataformas de 16 bits gráficas (ej: Windows 3.11), de 16 bits no gráficas (Ej: DOS), de 32 bits gráficas (Ej: Windows NT, Unix Sparc con Solaris, Unix x86 con XWindows), de 32 bits no gráficas (Unix en x86), etc.

Importante:

Todos los programas que se entreguen serán ejecutados por la cátedra en todas estas plataformas, por lo tanto no presuponer nada respecto de la plataforma en la que va a correr. Este es el motivo por el cual elegimos ANSI C. Por lo tanto no utilizar funciones que sólo puedan ser ejecutados en Windows, o DOS, etc.

1.2 Implementación de Algoritmos

Algoritmo genérico:

Es una secuencia finita de pasos para resolver un problema dado.

Algoritmo para ciencias de la computación:

Se le agregan condiciones para que pueda ser ejecutado por una computadora, a saber:

1. Debe ser simple y no ambiguo.
2. Debe ser efectivo.
3. Debe satisfacer la propiedad de finitud.

Aclarando la definición:

1. Simple y no ambiguo significa que debe ser presentado en forma clara y de tal modo que sea posible entender los pasos que involucra. Debe ser determinístico.
2. Efectivo quiere decir que es posible llevarlo a la práctica. Ejemplo: calcular el perímetro de una circunferencia como 2*R*π no es efectivo, ya que en computación el valor de π no es exacto (en matemática puede dejarse expresado).
3. No puede ejecutar indefinidamente, o sea debe garantizarse que el mismo finaliza luego de un número finito de pasos.

Conclusiones que se deducen de la definición anterior:

- Para poder codificar en forma sencilla y sin ambigüedades, conviene partir de la descripción del problema y plantear un algoritmo (primer nivel de refinamiento) en pseudo-código (mezcla de lenguaje coloquial y ciertas formas sintácticas de los lenguajes de computación más comunes). Luego en pasos sucesivos se va llegando a un nivel de abstracción menor (sucesivos refinamientos) hasta llegar a la codificación del mismo en el lenguaje elegido para implementarlo.

- Debe realizar correctamente lo pedido, o sea debe verificar la condición de correctitud.

- Debido a la tercera de las especificaciones resulta **inadmisible** tener un ciclo infinito.

Resolver un problema para una computadora consiste en dos pasos conceptualmente distintos:

1) Se necesita desarrollar un algoritmo o escoger alguno ya existente (quizás adaptándolo) para resolver el problema. Este paso se llama **diseño algorítmico**.

2) Expresar el algoritmo como un programa de computación en algún lenguaje de programación. Este paso se llama **codificación**.

Importante:

En un proyecto de desarrollo de software a gran escala, es necesario comenzar a codificar en el momento correcto: **no demasiado temprano**.

Si se comienza a codificar antes de que estén todas las especificaciones enunciadas en forma precisa (el requerimiento es incompleto), se asumirán cosas que cambiarán más tarde y complicarán el código notablemente.

Teniendo en cuenta el futuro mantenimiento de un sistema podemos decir que suele gastarse demasiado tiempo codificando algoritmos que después resultan difíciles de mantener y/o modificar, y por lo tanto se suele "tirar el trabajo realizado previamente" porque es más fácil comenzar uno nuevo que arreglar uno hecho antes.

Esto ocurre debido a que **no se utilizan técnicas diseñadas exclusivamente** para obtener códigos "correctos, legibles y entendibles".

Muy Importante:

Al elegir un algoritmo para un problema en particular, se deberán tener en cuenta las siguientes heurísticas:

1. **Correctitud:** que haga lo que tiene que hacer.
2. **Claridad:** que pueda ser entendido no sólo por el que lo desarrolló.
3. **Mantenibilidad:** que pueda ser fácilmente adaptado en el tiempo a otros problemas.
4. **Eficiencia:** que lo haga en el menor tiempo de ejecución posible y con el menor gasto de almacenamiento posible. Hacer que un programa sea eficiente es un objetivo admirable, pero lo primordial es que sea correcto y claro.

1.3 Testeo de Software

Testeo de Software

Es el proceso de ejecutar un programa con el fin de encontrar errores.

Así es como el objetivo de la prueba de software consiste en "encontrar errores". Existen beneficios secundarios que aparecen al realizar pruebas de software:

1. Demostrar que el software responde a la funcionalidad para la cual fue desarrollado (concuerda con su especificación).
2. Verificar si la performance es la esperada.
3. La colección de datos obtenidos durante el testeo es un buen indicador de la confiabilidad y la calidad de software.

Importante

- **Un test es bueno** si posee alta probabilidad de encontrar un error no descubierto todavía.
- **Un test es exitoso** si descubre un error no descubierto aún.

Como se observa existe analogía en el terreno médico: si un paciente consulta a un médico porque padece de cierto malestar, el doctor le solicita que realice ciertos estudios y si al término de los mismos los resultados no sirven para determinar el origen de la enfermedad, entonces se considera que el test **no fue exitoso** (además el paciente se siente defraudado porque invirtió tiempo y dinero), caso contrario tuvo sentido la inversión porque el médico puede ahora realizar la prescripción para su enfermedad.

Así como el doctor no envía a los pacientes a realizar **todos los estudios posibles** para diagnosticar una enfermedad, los diseñadores de test de software planean qué tests ejecutar con el fin de alcanzar el objetivo de detectar errores con un mínimo de tiempo y esfuerzo.

Aclaración:

La calidad de un test de software **no** está dado por la **cantidad** de pruebas diseñadas.

El flujo de información en las pruebas de software puede representarse con el siguiente esquema:

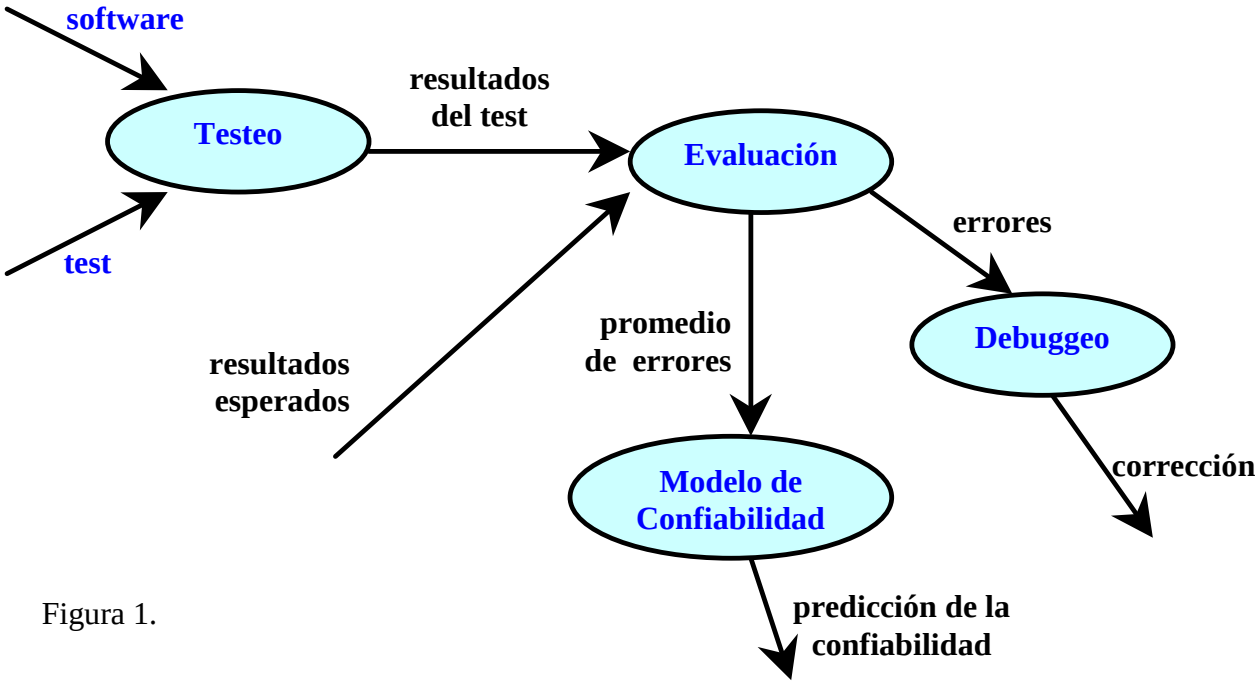


Figura 1.

Aclaración:

El testeo de software sólo se puede usar para mostrar la presencia de errores (bugs), desgraciadamente nunca muestra su ausencia.

Existen distintos tipos de test. No son alternativos sino **complementarios**.

- **Caja Negra** (Test Funcional)
- **Caja Blanca** (Caja de Cristal o Test Estructural)

1.3.1 Caja Negra

Intenta encontrar errores por los cuales un módulo del sistema no se comporta de acuerdo a su especificación.

Se puede utilizar para testear el funcionamiento global del sistema, pero también pueden utilizarse para probar cada módulo del mismo (unidad con menor o mayor granularidad).

La estrategia consiste en considerar a la unidad a testear como una caja negra y desentenderse completamente de su estructura interna. Se focaliza en el dominio de la información (desconoce la lógica de la unidad a testear). Sólo hace falta saber "qué" hace la unidad y no "cómo" lo hace. Consiste en buscar datos de entrada para la unidad a testear, y verificar si la salida obtenida concuerda con los valores esperados según la funcionalidad de la misma.

Importante

Para asegurar que un módulo funciona correctamente, de acuerdo con el test de caja negra, se debe probar la entrada de datos exhaustivamente. O sea emplear toda posible combinación de entrada de datos como un caso de prueba.

Ejemplo: Si hay que testear un programa que lee tres valores de datos e informa si los mismos corresponden a los lados de un triángulo habría que probar con los valores: (2, 0, -3), (4, '¿', '5'), ('u', 'r', 3), ("hola"), etc., etc., etc.

Muy Importante

Armar un test con **todas** las posibles combinaciones de datos ingresables resulta **impracticable**.

Debido a la imposibilidad real de realizar el test de caja negra exhaustivamente vamos a ***aplicar heurísticas*** para generar pruebas más económicas (en tiempo y esfuerzo) pero que intenten formar un buen testeo de caja negra.

Las mismas consisten en:

- ***Particiones de equivalencia***: Ya que nos conformaremos con usar sólo un subconjunto de todas las posibles entradas, trataremos que el mismo tenga la *mayor probabilidad posible de encontrar gran parte de los errores*. Para ello se procede en dos partes:

  - Se divide el **dominio de entrada en clases de equivalencia** (pueden ser más de dos) de acuerdo a la validez o invalidez de los mismos. Si existe alguna razón para creer que ciertos elementos pertenecientes a una clase de equivalencia no

son tratados en forma idéntica por el programa se los divide en más clases de equivalencia. Después se completa la siguiente tabla:

Entrada	Clase de equivalencia válidas	Clase de equivalencia inválidas
	...	...

- Se elige **un valor representante de cada clase válida y todos ellos deben ser cubiertos por lo menos por un test**. Hay que elegir tantos tests como sean necesarios hasta cubrir todos los valores elegidos como representantes válidos. Pueden ser menos test que la cantidad de valores elegidos si se busca involucrar simultáneamente a varios valores en un mismo test. Finalmente se busca un **valor representante de cada clase inválida y se construye un test para cada uno de ellos por separado** (la idea de testear los inválidos por separado es tratar de asegurarse de que cada uno de ellos esté testeado). Ejemplo: Si se pidiera ingresar dos números positivos y el test consistiera en ingresar el par (-5, 0) según como esta construido el programa, y obviamente esto no se sabe en caja negra, puede haberse validado el primero pero no el segundo y no nos daríamos cuenta.

- ***Análisis de los valores límite***: Los valores límites son aquellos que están justo por debajo o arriba de lo márgenes de **las clases de equivalencia para las entradas y las salidas** (en el paso anterior solo se toma las clases de equivalencia de la entrada). Notar que para los valores límites también se *analizan los espacios de resultados que se esperan obtener* para saber si lo que el módulo dice calcular realmente coincide con lo anunciado. Si el rango de entrada o salida fuera [n,m] probar la funcionalidad de la unidad con n, n-1, n+1, m, m-1 y m+1. Notar que para el rango de salida es más difícil porque hay que pensar en la entrada que habría que ingresar para que la salida fuera la buscada.

- ***Conjetura de errores***: La experiencia muestra que hay ciertas personas que poseen una mejor predisposición a construir muy buenos test de software, quizás en base a su experiencia. A modo de ejemplo citaremos **algunos trucos** a tener en cuenta que pueden ayudar a completar los tests de caja negra: si el módulo realiza una división (aunque no se vea el código, por su funcionalidad se puede deducir) tratar de provocar una entrada que haga que el divisor sea cero, si solicita una lista de entrada hacer que sea vacía, que todos los valores sean iguales, etc.

- ***Gráficos de causa-efecto***: consiste en trazar ciertos gráficos que ayudan a encontrar errores. No la vamos a estudiar.

1.3.2 Caja Blanca

Esta estrategia permite examinar la estructura interna del módulo a testear. Los datos del test surgen de examinar la lógica del mismo, pudiéndose detectar errores de lógica, tipográficos y otros especiales asumidos incorrectamente. Le hace falta saber "cómo" hace la unidad y no "qué" hace.

Resulta imposible aplicarla a un sistema completo, es preciso aplicarla a cada módulo por separado.

El análogo a la prueba exhaustiva en ***caja negra*** correspondería a generar tests tales que se generen todas las combinaciones posibles de secuencias de flujo de control del programa. Obviamente ésto es impracticable, y terminamos conformándonos con generar un subconjunto de pruebas que nos ayude a detectar errores de lógica en el modulo a analizar.

Una vez más vamos a ***aplicar heurísticas*** para generar pruebas más económicas (en tiempo y esfuerzo) pero que intenten formar un buen testeo de caja blanca. Las mismas consisten en:

- ***Cobertura de sentencias***: armar test que aseguren que por lo menos se ejecuta una vez cada instrucción.

- ***Cobertura de decisión/condición***: armar casos de prueba para que la condición involucrada en una decisión (si entonces, si entonces sino, mientras, desde hasta) se evalúe una vez como verdadera y otra como falsa (o los demás casos en el caso de un switch) y además que cada criterio involucrado en dicha condición sea evaluado con todos los valores posibles (una vez como verdadero y otra como falso). Ejemplo: si la condición de decisión fuera if (a AND b) ... los tests (a con verdadero, b con falso) y (a con falso, b con verdadero) no es suficiente porque verifica la segunda parte pero no la primera (las dos expresiones se evaluarían en falso). Los tests (a con verdadero, b con verdadero) y (a con falso, b con falso) verificarían la cobertura de decisión/condición**.**

- ***Cobertura de valores limites***: armar casos de prueba para que los criterios involucrados en una condición que especifiquen un rango [n, m] se prueben con valores justo alrededor de sus límites: n-1, n, n+1, m-1, m, m+1. Notar la similitud con caja negra, pero ahora a nivel interno y no externo.

Importante:

Nunca aplicar Caja de Pandora que significa no testear nada y rogar para que el usuario del software no tenga problemas con el mismo.

1.3.3 Conveniencia de Ambos Testeos

Usar la metodología de *"prueba de azar"* que consiste en construir pruebas con valores de entradas elegidos al azar es *la peor elección* porque hay muy poca probabilidad que los casos de test así elegidos, se acerquen al subconjunto óptimo.

Con respecto a la técnica de caja blanca y caja negra, hemos visto que cada una de ellas tiene sus puntos fuertes y débiles. En esta materia usaremos la combinación de ambas.

Hay un viejo dicho que dice **"si Ud. cree que diseñar y codificar ese programa ha sido difícil, entonces aún no ha visto nada"** y se refiere a la gran tarea que implica diseñar y aplicar test de software para detectar la calidad del mismo, por tal motivo hemos estudiado técnicas que servirán como herramientas de ayuda en el desarrollo de software confiable.

Muy Importante

Notar que todo lo que se expuso hasta ahora corresponde al buen diseño de tests de software. No perder de vista que una vez construidas dichas pruebas lo que debe hacer es ejecutar el módulo con cada una de ellas, prediciendo de antemano qué debería hacer el módulo frente a los valores de entrada propuestos y convalidando si la salida concuerda con lo esperado (ver figura 1), caso contrario el test ha resultado un éxito, y se procede a arreglar el código fuente, para luego testearlo nuevamente. Tener en cuenta que en la mayoría de los casos, al arreglar un error suelen surgir errores nuevos.

1.3.4 Debuggeo

Los errores que suelen surgir en un programa podrían clasificarse globalmente en:

- **Errores en tiempo de compilación (sintácticos).** Los lenguajes de programación tienen su propia sintaxis, que conforma las reglas que indican cómo construir una sentencia válida en el mismo. En la etapa de compilación el traductor realiza el chequeo de errores sintácticos, avisando la lista de errores cometidos.

- **Errores en tiempo de ejecución (lógicos o bugs).** Estos errores son los que producen el funcionamiento incorrecto de un programa. Debido a la confianza que tiene todo programador sobre su propio código, son muy difíciles de detectar. Por este motivo es muy importante aplicar las distintas técnicas de testeo de software ya vistas, para lograr encontrarlos y corregirlos. El éxito de un programador se debe en parte a aplicar adecuadamente buenas técnicas de testeo de software. Una vez detectados los errores lógicos, se suelen corregir mediante la ***técnica de debuggeo***, que consiste en poder ejecutar paso a paso el programa, inspeccionando el estado de las variables, verificando si el flujo de control es el esperado, etc.

2 Ejercicios de Aplicación

Ejercicio 1

Se tiene un módulo para calcular la media aritmética de un conjunto no nulo de números. Para el ingreso de datos se solicita primero la cantidad de números a ingresar y luego los mismos.

- Aplicando el testeo de Caja Negra, proponer lotes de prueba para probar si el módulo funciona correctamente
- Aplicando el testeo de Caja Blanca, proponer lotes de prueba para probar si el módulo funciona correctamente. El código de dicho algoritmo es (pseudo-código):

```
Enteros: cantidad, Acumulador
Real: Nro

Leer(cantidad)

blanquear(Acumulador)

mientras cantidad > 0
comienzo
        Leer(Nro)
        Incrementar(Acumulador, Nro)
        Decrementar(cantidad, 1)
fin

escribir( Acumulador/cantidad)
```

Posible Solución

- ***Caja Negra***

➢ ***Particiones de equivalencia***

Armamos las clases de equivalencia para la entrada. Podría ser que los caracteres sean tratados diferentes a los números desde el punto de vista de programación, y por lo tanto partimos la clase de equivalencia de lo invalido:

Entrada	*Clase de equivalencia valida*	*Clases de equivalencia invalidas*	
cantNro	**Enteros >= 0**	**Enteros < 0 o reales**	**caracteres**
nro	**Entero o real**	**caracteres**	

Ahora armamos los primeros test

T1= { cantNro= 3, nro= -3, nro= 0.4, nro= 5 } con esto abarcamos los representantes de cada clase válida en la entrada.

T2= { cantNro= -1 }
T3= { cantNro= 7.4}
T4= { cantNro= "hola"}

T5= { cantNro= 2, nro= "que", nro= 4}
T6= { cantNro= 2, nro= 3, nro= "que"}

Cada representante de las clases invalidas en un test separado!!! (un solo elemento inválido en cada test)

- ***Análisis de los valores límite***

El valor acotado es la cantidad de números a ingresar que debe ser mayor o igual que cero. No hay cota explícita indicada para la cantidad ni para los valores. En este caso el límite es inferior: cantNro >=1.

Así debemos probar con un test que oscile en uno: cantNro= 0, otro con cantNro= 1 y otro con cantNro= 2.

T7= { cantNro= 0 }
T8= { cantNro= 1, nro= 5 }
T9= { cantNro = 2, nro = 2.3, nro= 7}

Con respecto a la salida, identificar rangos validos y no validos, y en base a eso busca cuales deberían ser las entradas para que eso fuera posible. Como la salida es un promedio, los valores inválidos sería cualquier cosa distinta de un numero y no resulta viable inventar una entrada para ésto.

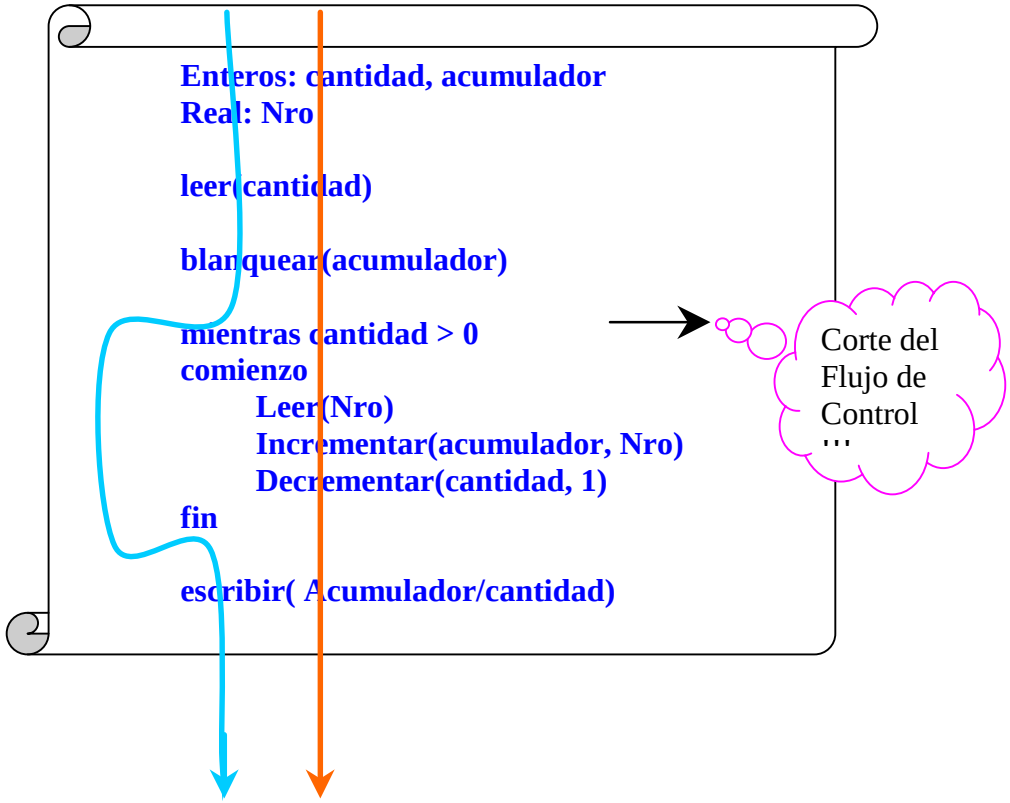
- ***Conjetura de errores***

Como el módulo realiza una división para hacer el promedio, podríamos pensar un test que haga que la división incorrecta. Para eso cantNro debería ser cero, y eso ya lo contemplados en T7.

❖ *Caja Blanca*

➢ *Cobertura de sentencias*

Marcamos flujos como para que las sentencias se an ejecutadas por lo menos una vez.



Si comenzamos pensando en el flujo de control marcado con rojo, no surgiría el flujo de control marcado con turquesa ya que el mismo no agrega ningún camino nuevo. En cambio si comenzamos pensando en el turquesa, es necesario agregar el camino alternativo marcado con rojo para incluir las instrucciones dentro del ciclo. Así es como de este análisis surgen:

T10= { cantNro= 3, nro= -3, nro= 6.4, nro= 2 }

o bien

T11 = { cantidad= -3}
T12= { cantidad= 2, nro= 2, nro= -5.2}

Nótese además que si el diseñador de este test fuese el mismo que ideó las pruebas para caja negra, podría haber considerado explícitamente: T10 = T1 (la calidad del test no está dado por la cantidad de las pruebas, sino por tal potencia de descubrir errores).

- ***Cobertura de decision/condicion***

Para la condición cantidad > 0 debemos considerar un test para que la misma se evalúe como verdadera y otra como falsa. O sea qué datos habría que entrar para que esto fuera posible. Si cantidad fuera 2, se evaluaría hasta llegar a valer 0 y no debería entrar más, o sea se evaluaría como falsa.

Para esto nos sirve **T12**

- ***Cobertura de valores límites***:

Como el límite inferior de la condición del ciclo es mayor que cero, eso implica que el valor entero límite con el cual se entra al cuerpo del ciclo es el entero 1. Por lo tanto habría que hacer pruebas con los valores cero, 1 y 2 para cantidad y eso ya lo hicimos en caja negra.

❖ ***Paso Final del Testeo***

Ahora habría que ejecutar (en un papel o con una computadora) cada uno de los tests para ver si producen los valores esperados.

Los test ideados para caja negra sólo pueden ejecutarse en computadora. Los otros pueden correrse en papel o con computadora.

Una vez ideado los tests, se procede a la etapa final, que consiste en ejecutar el módulo o programa con esos valores y ver si se obtienen los valores esperados. Podríamos completar la siguiente tabla:

Test de Prueba	*Valor esperado*	*Valor obtenido*	*Coincide?*
T1	**2.4/3 = 0.8**	**Aborta: División por cero**	**NO**
T2	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T3	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T4	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T5	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T6	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T7	**"Divisor erróneo"**	**Aborta: División por cero**	**NO**
T8	**5**	**Aborta: División por cero**	**NO**
T9	**9.3/2 = 4.65**	**Aborta: División por cero**	**NO**
T10	**5.4/3 = 1.8**	**Aborta: División por cero**	**NO**
T11	**"Error de Datos"**	**Aborta: División por cero**	**NO**
T12	**-3.2/2 = -1.6**	**Aborta: División por cero**	**NO**

Son de caja negra, hay que ejecutarlos

Obviamente hay que revisar el código por qué siempre se anula el divisor!

Ejercicio 2

Se tiene un módulo que solicita un cartel de 5 letras como máximo y lo escribe transformado a minúscula. Cualquier símbolo que no sea letra mayúscula queda intacto. El cartel podría ser nulo.

- Aplicando el testeo de Caja Negra, proponer lotes de prueba para probar si el módulo funciona correctamente
- Aplicando el testeo de Caja Blanca, proponer lotes de prueba para probar si el módulo funciona correctamente. El código de dicho algoritmo es (pseudo-código):

```
leer(cartel)
longitud = largo del cartel leído

mientras ( ( longitud > 0)  and  (longitud<6)
comienzo
        letra= letra en posición longitud
        si letra >= 'A' and letra <= 'Z'
        entonces
        comienzo
                incrementar(letra, 'a' – 'A' )
                escribir(letra)
        fin
        sino
                escribir (letra)

        decrementar(longitud, 1)
fin
```

Posible Solución

- ***Caja Negra***

- ***Particiones de equivalencia***

Armamos las clases de equivalencia para la entrada:

Entrada	*Clase de equivalencia valida*	*Clases de equivalencia invalidas*
cartel	Hasta 5 caracteres de longitud	Mas de 5 caracteres de longitud

Ahora armamos los primeros test

T1= { "hola5" } con esto abarcamos los representantes de cada clase válida en la entrada.

T2= { "encuentro" } con esto abarcamos los representantes de la clase inválida en la entrada

- ***Análisis de los valores límites:***

El valor acotado es la cantidad de caracteres a ingresar, o sea la cantidad deber pertenecer al intervalo [0, 5]. Oscilando en los extremos podríamos querer ingresar un cartel de longitud -1 , pero esto resulta imposible. Nos quedamos con carteles de longitud 0, 1, 4, 5, 6.

T3= { "" }
T4= { "a"}
T5= { "mono" }
T1
T6= { "123456" }

Con respecto a la salida, habría que identificar rangos válidos y no válidos, y en base a eso busca cuáles deberían ser las entradas para que eso fuera posible. Como el enunciado consiste en pasar a minúsculas cualquier letra original que esté en mayúsculas y deja intacta al resto, vamos a pensar en función de la salida. La clase de equivalencia válida y no válida de la salida está dada por

Salida	*Clase de equivalencia valida*	*Clases de equivalencia invalidas*
Cartel	Hasta 5 caracteres de longitud, en minúsculas	Caracteres en mayúsculas o más de 5 caracteres de longitud

Habría que pensar, sin conocer el código, qué casos se pueden haber escapado al programador que hace que falle la transformación: si el carácter a pasar es el primero, es del medio o es el ultimo, o bien si vienen todos iguales. Con un poco de intuición construimos:

T7= { "T"}
T8= { "aSi" }
T9= { "asI" }
T10= { "VEZ" }
T11= { "No es" }

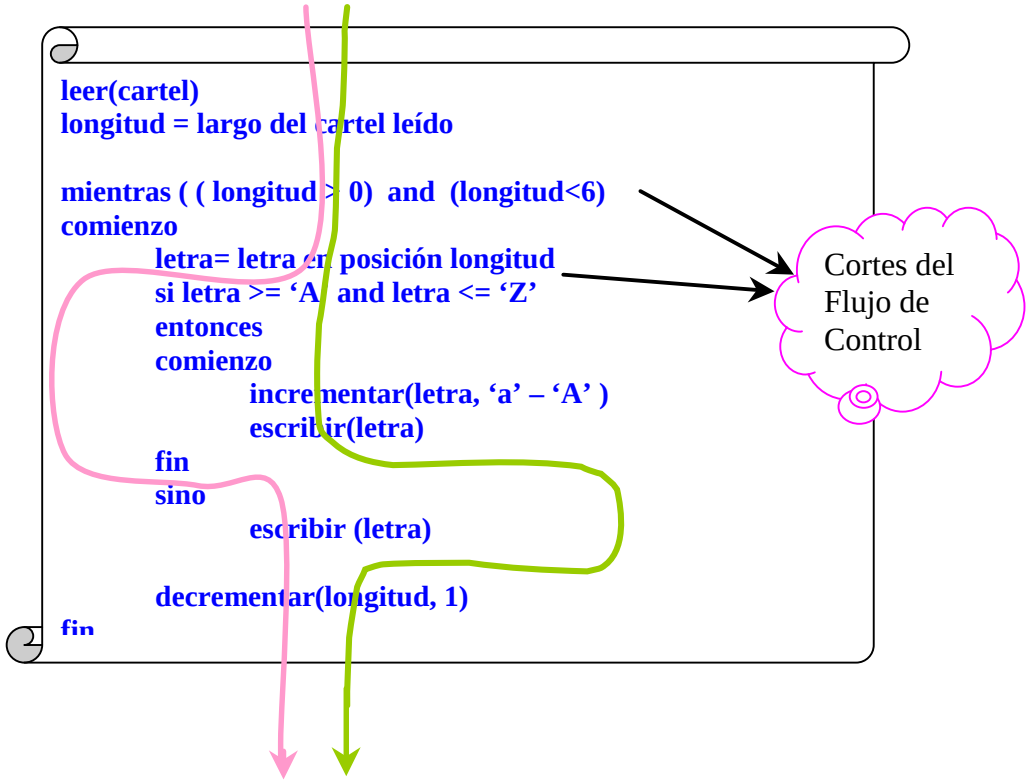
- ***Conjetura de errores***

Ya fueron contemplados en el caso anterior.

❖ *Caja Blanca*

➢ *Cobertura de sentencias*

Marcamos flujos como para que las sentencias sean ejecutadas por lo menos una vez:



Así es como de este análisis surgen test que usen minúsculas y mayúsculas. Para esto nos sirve **T8.**

➢ *Cobertura de decision/condicion*

Para el mientras tenemos la condición (longitud >0 and longitud < 6) como toda ella debe evaluarse como verdadera y como falsa, y a cada subcondición le debe pasar lo mismo podríamos pensar en tests que combinen: (verdadero, verdadero), (falso, verdadero) y (verdadero, falso).

T11
T3
T6

Para la condición del "si entonces" tenemos letra > ='A' and letra <= 'Z'. Los test podrían ser los que permitan obtener (verdadero, verdadero), (falso, verdadero), (verdadero, falso)

T12= { "M3m" } que contempla las tres situaciones.

- ***Cobertura de valores límites***

  - Para ***cantidad*** tenemos el rango [1, 5], por lo tanto buscamos entradas que oscile con las cantidades de letras 0, 1, 2, 4, 5, 6:

    T3, T4, T13= { "nO" }
    T5, T1, T6

  - Para ***letra*** tenemos el rango ['A', 'Z'], por lo tanto buscamos entradas que oscile con las cantidades de letras '@', 'A', 'B', 'Y', 'Z', '[' o por separado:

    T14 = { "@AB" }
    T15 = { "YZ[" }

- ***Paso Final del Testeo***

Una vez ideados los tests, se procede a la etapa final, que consisten en ejecutar el módulo o programa con esos valores y ver si se obtienen los valores esperados.

Test de Prueba	*Valor esperado*	*Valor obtenido*	*Coincide?*
T1	**"hola5"**	**"5aloh"**	**NO**
T2	**"Exceso de letras"**	**-----**	**SI (*)**
T3	**""**	**""**	**SI**
T4	**"a"**	**"a"**	**SI**
T5	**"mono"**	**"onom"**	**NO**
T6	**"Exceso de letras"**	**-----**	**SI (*)**
T7	**"t"**	**"t"**	**SI**
T8	**"asi"**	**"isa"**	**NO**
T9	**"asi"**	**"isa"**	**NO**
T10	**"vez"**	**"zev"**	**NO**
T11	**"no es"**	**"se on"**	**NO**
T12	**"m3m"**	**"m3m"**	**SI**
T13	**"no"**	**"on"**	**NO**
T14	**"@ab"**	**"ba@"**	**NO**
T15	**"yz["**	**"[zy"**	**NO**

Son de caja negra, hay que ejecutarlos

Obviamente hay algo en el código que hace que el cartel se escriba al reves, y eso se nota en carteles con más de un carácter.

(*): Se podría mejorar con un cartel explicativo, pero no funciona mal.

3. Principios de un Buen Estilo de Programación

3.1. Modularización

- Dividir el problema original en pequeños subproblemas. Dichos subproblemas no debe tener más que unas pocas líneas de código (jamás debe exceder el tamaño de una pantalla, aproximadamente 20 líneas)

- Cada módulo debe realizar una única función, el nombre del mismo debe resumir lo que hace (modularización)

- Utilizar parámetros para pasar la información a los módulos. No usar variables globales que puedan traer efectos colaterales indeseables

3.2. Claridad

- Utilizar nombres claros para los identificadores de constantes, variables, y funciones (no letras o nombres no significativos)

- Cada módulo debe ser documentado, incluyendo todos los comentarios suficientes para comprender lo que hace. No olvidarse que en grandes proyectos el programador que comienza un módulo no siempre es el que lo mantiene o modifica posteriormente, por lo tanto la documentación deber ser más que clara.

- Poner cada instrucción en una línea.

- Indentar el código para mayor claridad.

3.3. Estructuración

- Utilizar las siguientes estructuras de control: secuencia, decisión e iteración. No usar estructuras de salto incondicional (goto y variantes).

- Utilizar programación estructurada: la misma fue creada para garantizar que un módulo tuviera un único punto de entrada y un único punto de salida. Si bien el lenguaje ANSI C no es estructurado, se exigirá que cuando se levante esta condición sea para claridad y en ningún caso para enturbiar la lógica del módulo.

3.4. Eficiencia

- No basta con crear algoritmos inéditos. Es importante también ver algoritmos realizados por otros programadores y discutir si son eficientes, claros y correctos. En la materia se sugerirá la consulta de algoritmos para alentar el desarrollo del espíritu crítico y tomar ideas sobre buen estilo de programación.

- Tampoco es suficiente quedarse con el primer algoritmo que se nos ocurre. Debemos analizar diferentes alternativas y quedarnos con las que sean más eficientes (en memoria y tiempo de ejecución) y fáciles de modificar y mantener.

- No producir códigos que para generar una cierta eficiencia sean imposibles de entender en su semántica. La lógica de los programas debe ser lo más importante (sin por eso degradar la performance).

3.5. Correctitud

- Programar defensivamente, es decir **no presuponer que ciertos valores son imposibles o que ciertas acciones nunca van a ocurrir.**

- El módulo desarrollado debe ser testeado utilizando técnicas desarrolladas para tal fin. La calidad del test es más importante que la cantidad del mismo.

- Contemplar todos los posibles errores que puede tener un programa en tiempo de ejecución, e informar para dichos casos un mensaje de error muy claro que permita entender a qué se debió el error producido (Ej: valor fuera del rango, disquetera abierta, etc.)

Importante:

Para la escritura de programas en C se seguirán en esta materia la mayoría de los estándares de AT&T's Indian Hill labs.

De esta forma se obtendrá un estilo respecto de la indentación, comentarios, nombre de identificadores, cantidad de columnas usadas, etc., que facilitará la lectura de códigos escritos.

Estrategias de Programación - Introducción al Lenguaje C

Introducción

En este documento se detallan las técnicas Top-Down y Bottom_Up, para la resolución de problemas y se presenta una breve introducción al Lenguaje C, que comprende sus características básicas, su proceso de traducción y la forma de estructurar un programa.

1. Estrategias de Programación

1.1 Metodología Top-Down

El proceso de dividir un problema en partes más pequeñas, que son individualmente fáciles de entender, se denomina descomposición. Esto constituye una estrategia fundamental de programación.

Si se eligen correctamente esas piezas individuales, cada una tendrá una integridad conceptual como unidad y hará el programa más fácil de comprender.

En el momento del ***desarrollo de un programa*** (etapa temprana) es bueno comenzar con un diseño Top-Down o de refinamiento, que consiste en comenzar por el programa principal y, desde esa perspectiva, pensar en el programa como un todo, tratando de identificar las principales piezas que lo componen. Si dichas piezas continúan siendo demasiado grandes para ser manejadas, se continúan subdividiendo hasta que todas las piezas del problema sean lo suficientemente simples como para ser resueltas por sí mismas.

Para representar esta metodología, utilizaremos la siguiente especificación:

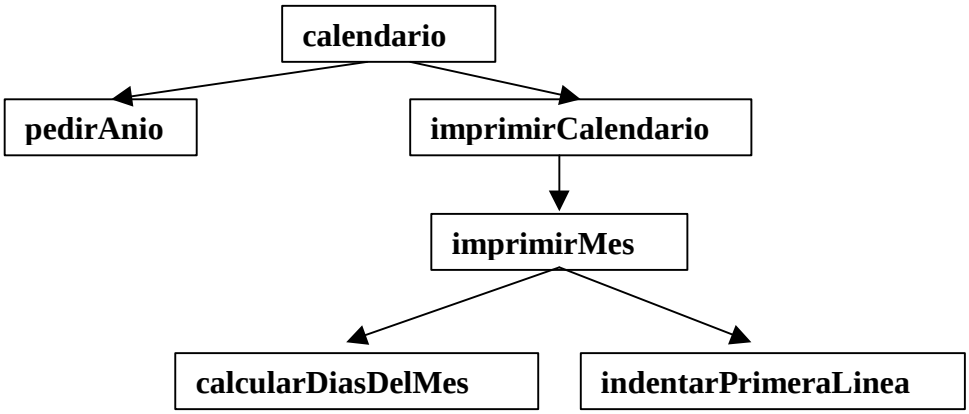
- ***Representación Gráfica:*** La estructura para representar la metodología top-down es un árbol. Cada nodo del mismo se corresponde con una de las piezas del problema. Una pieza tiene hijos si ella misma es demasiado grande como para poder realizar la tarea sola, y precisa ser refinada en otras sub-piezas. Todas las hojas del árbol (nodos sin hijos) son lo suficientemente chicas como para no precisar de nuevo refinamiento.
  En cada nodo sólo se especifica el nombre de la pieza, que deberá coincidir con el nombre que lleve en la codificación final. Dicho nombre debe ser representativo, indicando la funcionalidad de dicha pieza.

- ***Documentación de la Interfaz:*** La representación anterior se complementa con una tabla de doble entrada, que para cada pieza indica: ***Nombre*** (que la identifica unívocamente), ***Descripción*** (breve explicación de qué es lo que hace y no cómo lo hace) y ***Parámetros*** (lista que indica, para cada uno de ellos, su tipo, si es de entrada, de salida o de entrada/salida y su valor esperado, inicial y/o final según corresponda)

Ejemplo:

Se tiene el problema de dibujar un calendario completo para un año solicitado, que debe ser posterior a 1900

Representación Gráfica:



Documentación de la Interfaz

Nombre	***Descripción***	***Parámetros***	
calendario	Genera en formato gráfico el calendario correspondiente a un año leído de la entrada estándar	No tiene	
pedirAnio	Solicita un numero válido para un año posterior a 1900	**anio**	Tipo: entero De salida Valor final: numero mayor a 1900
imprimirCalendario	Imprime todos los meses de un año válido posterior a 1900	**anio**	Tipo: entero De Entrada Valor inicial: numero mayor a 1900
imprimirMes	Imprime un mes de un año dado posterior a 1900	**mes**	Tipo: entero De entrada Valor inicial: numero entero entre 1 y 12
		anio	Tipo: Entero De entrada Valor inicial: numero entero mayor a 1900

Nombre	*Descripción*	*Parámetros*	
calcularDiasDelMes	Calcula los días que tiene un mes dado para un cierto año	**mes**	Tipo: entero
			De entrada
			Valor inicial: numero entre 1 y 12
		anio	Tipo: entero
			De entrada
			Valor inicial: numero mayor a 1900
		dias	Tipo: entero
			De Salida
			Valor Final: cantidad de días del mes y año dados
indentarPrimeraLinea	Calcula en que día de la semana se encuentra el primer día del mes dado para un cierto año	**mes**	Tipo: entero
			De entrada
			Valor inicial: numero entre 1 y 12
		anio	Tipo: entero
			De entrada
			Valor inicial: numero mayor a 1900
		primerDia	Tipo: entero
			De salida
			Valor Final: número entre 1 y 7 que representa el día de la semana correspondiente al primer día del mes y año dados (1 es el lunes)

1.2 Metodología Bottom_Up

Una vez especificadas las piezas mediante la metodología Top-Down, se pasa a al ***momento concreto de implementación*** (etapa tardía) en el cual la estrategia utilizada es la Bottom-Up, que consiste en codificar cada una de las piezas individuales, testearlas por separado y luego, a partir de las mismas, construir el programa completo.

2. Lenguaje de Programación C

El **lenguaje C** fue desarrollado por **Dennis Ritchie** en los laboratorios Bell, a partir de ciertas ideas del lenguaje B, desarrollado previamente por Ken Thompson, el cual a su vez estaba basado en el lenguaje BCPL, creado por Martin Richards en 1967.

El lenguaje y el compilador C introducen la noción de tipos que su antecesor no tenía. En 1972 al implementarlo en una computadora PDP-11 de DEC se hizo que su traductor fuera un compilador (generando código para dicha máquina), mientras que su antecesor era interpretable. Este nuevo lenguaje fue utilizado, muy tempranamente, para escribir partes del Sistema Operativo UNIX.

Existían ciertas características del lenguaje que no estaban bien detalladas por los autores, y como los desarrolladores de compiladores sólo utilizaban como referencia el libro "**The C Programming Language"** escrito por **Brian Kernighan y Dennis Ritchie**, surgían algunas ambigüedades entre los distintos compiladores. Así es como los programadores encontraban que sus programas no podían ser realmente multiplataforma porque existían variaciones entre los distintos compiladores del lenguaje C. Para eliminar la libre interpretación de los implementadores de compiladores C, se decidió estandarizar el lenguaje.

En 1983 se creó el comité técnico bajo **ANSI** (American National Standard Institute) para proporcionar una definición del lenguaje, no ambigua e independiente de la máquina.

En 1989 el estándar se concentró en la sintaxis y la semántica del lenguaje y especificó un mínimo entorno (nombre y contenido de archivos de encabezamiento y especificaciones de algunas funciones de la biblioteca estándar), y quedó aprobado como ANSI X3.159-1989. Se pueden conseguir copias del documento en:

American National Stantards Institute
Sales Departament
1430 Broadway
New York, NY 10018
(voice) (212) 642-4900
(fax) (212) 302-1286

Importante:

En esta materia vamos a trabajar dentro del paradigma imperativo, con el lenguaje ***ANSI C.***

2.1 Características del Lenguaje C

El lenguaje C ofrece un conjunto de tipos de datos y un conjunto **reducido** de instrucciones con las cuales construir programas.

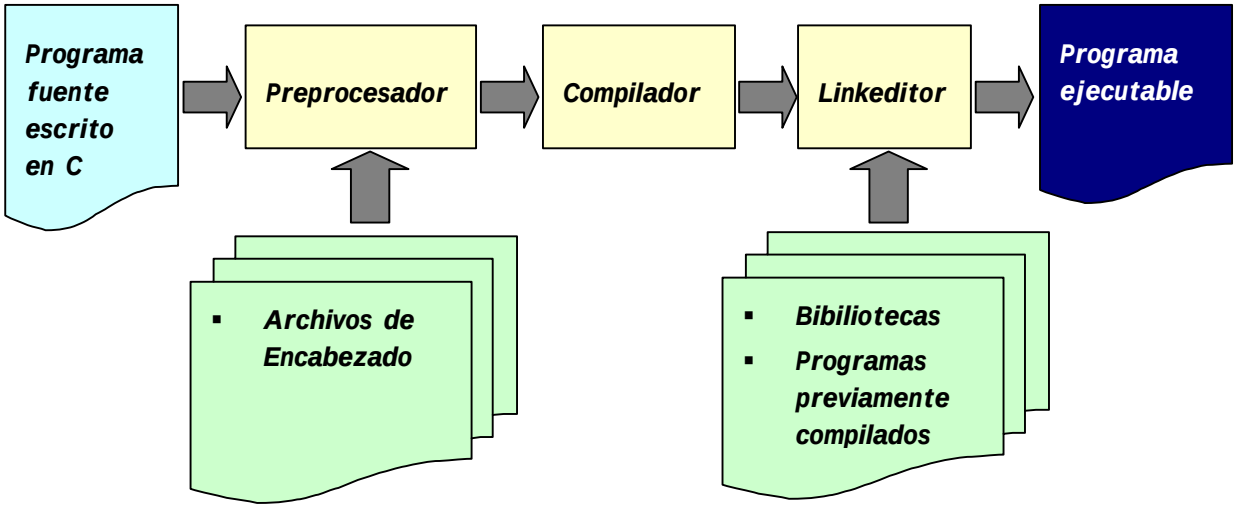
Esto no significa que el lenguaje no sea potente. Muchas funciones son implementadas en la Biblioteca Estándar la cual posee una colección de funciones para realizar cálculos matemáticos, manejos de cadenas de caracteres, entradas/salida, etc.

Aclaración:

Las funciones de la Biblioteca Estándar **no forman parte del lenguaje,** aunque se garantizan que vienen con el paquete del mismo, por lo tanto para lograr un ejecutable hay que linkeditarla

2.2 El Traductor del Lenguaje C

Las fases por las que pasa un programa fuente escrito en C para poder ejecutarse son las siguientes:



Aclaración:

Si todas las fases citadas anteriormente han resultado exitosas, se está en condiciones de pedirle al sistema operativo que ejecute el programa obtenido. Así es como el programa será cargado en Memoria Principal por un módulo del Sistema Operativo (el cargador). En algún momento el Sistema Operativo le dará el control al programa (la UCP empieza a ejecutar sus instrucciones)

2.2.1 El Preprocesador

Es un módulo separado del compilador de C, y tiene sus propias reglas y sintaxis.

Los programas utilizan al preprocesador para incluir archivos de encabezamientos (header files), expandir macros, definir símbolos, etc. (Recordar el proceso de macro-expansión que ocurría en Z80 antes del ensamblado)

Toda directiva al preprocesador comienza con el símbolo #.

El preprocesador acepta directivas en cualquier lugar del programa, y permite anidamientos entre directivas.

El mismo deja un archivo de salida transitorio (salvo que hayan habido errores) para que el compilador pueda procesar.

2.2.2 El Compilador

Es un módulo que toma el archivo de salida transitoria que generó el preprocesador y se encarga de chequear la sintaxis (reglas gramaticales) del lenguaje C y hacer la traducción al lenguaje de máquina, generando un modulo objeto, no ejecutable todavía. (Recordar el módulo *zas* de Z80)

- ¿Por qué es que no puede generar un programa ejecutable?
- ¿Qué cosas puede no poder resolver el compilador?

2.2.3 El Linkeditor

Es un modulo que toma el programa objeto generado por el compilador, y lo vincula con librerías u otros módulos objeto, resolviendo las referencias externas. La salida, de ser exitosa, es un programa listo para ejecutarse. (Recordar el *link* de Z80).

2.2.4 Algunos Traductores C del Mercado

❖ En DOS, el paquete ***Borland C*** ofrece un entorno para editar el programa, y un intérprete incremental para ejecutarlo. Pero también se lo puede editar con cualquier otro programa e inclusive compilarlo desde la línea de comandos.

El intérprete incremental suele usarse a los fines de aplicar debuggeo.

❖ En UNIX, el compilador de C se llama ***cc***, y de no haber errores en el proceso de traducción, genera un programa ejecutable llamado ***a.out*** (por default). Este módulo ***cc*** también se encarga de invocar, en forma transparente, al linkeditor ***ld***, por lo cual la mayoría de las veces no es necesario invocar al modulo linkeditor en forma explícita.

2.3 Estructura de un Programa en C

❖ En el lenguaje C un programa está formado por funciones. Una función contiene, entre llaves, proposiciones que especifican las operaciones que se van a realizar. Todo programa en C debe tener una función especial llamada ***main***, que debe ser única. El compilador de C se encarga de traducir el código a lenguaje de máquina de manera tal que luego, el módulo cargador del sistema operativo, le entregue el control a la primera instrucción que aparece dentro del módulo llamado *main*. (Recordar la directiva ***end rótulo*** de Z80)

En lenguaje C, toda función puede devolver un valor a quien la invoca, y la función ***main*** no es la excepción. Pero, ¿Quién es el llamador de dicha función? Puede ser el intérprete de comandos del sistema operativo, o bien programas escritos en un lenguaje especial que invocan a otros programas llamados ***command scripts***. Existen intérpretes de comandos que ignoran el valor que esta función regresa, pero otros en cambio lo tienen en cuenta. Como queremos que los programas sean portables, por convención haremos que la función *main* devuelva un 0 cuando ha finalizado exitosamente (terminación normal) y cualquier otro valor en caso contrario (terminación anormal).

❖ Generalmente en un programa existen otras funciones además del *main*. Eso hace necesario en muchos casos (especialmente cuando dichas funciones están en bibliotecas u otros módulos objetos) indicarle al preprocesador que incluya un archivo, llamado de encabezamiento, con cierta información sobre esas funciones, a través de la directiva ***#include***.

Sintaxis

#include **<nombreDeArchivo>**

ó bien

#include **“nombreDeArchivo”**

Las dos formas son esencialmente idénticas en su operación. En ambos casos el preprocesador busca un archivo con el nombre indicado y reemplaza la línea del *include* por el contenido de dicho archivo. La única diferencia reside en el lugar donde se hace la búsqueda: si el nombre del archivo esta entre <> el preprocesador lo busca en un directorio especial reservado previamente, y si el nombre está entre comillas lo busca en el directorio actual de trabajo (*current directory*).

❖ Como ya hemos visto anteriormente, es de buen estilo y clarifica la programación, el incluir comentarios dentro del código fuente. Cabe señalar que los mismos son ignorados por el compilador y sirven a los programadores para mejorar la mantenibilidad del código que ellos documentan.

En lenguaje C, para incluir un comentario se debe colocar su texto encerrado entre los símbolos **/*** y ***/**. Un mismo comentario puede ocupar varias líneas, pero no está permitido el anidamiento de comentarios.

Ejemplo: Este el primer programa que proponen K & R

```
/*
 *      Programa hello.c
*/

#include <stdio.h>

int
main (void)
{
        printf("Hello World \n");
        return 0;
}
```

Comentarios

Inclusión de Encabezamientos

Funciones

3. *Estandarización en la Codificación y el Estilo del Lenguaje C*

El uso de un buen estilo ayuda a la claridad, portabilidad y mantenibilidad de un código, reduciendo la probabilidad de cometer errores.

Es importante que un grupo de programadores que pertenecen a una misma institución adopten reglas claras de estilo.

Muchas de las reglas elegidas (indentaciones, comentarios, convenciones en los nombres, etc.) pueden parecer arbitrarias, pero son el resultado de la experiencia de programadores expertos en el lenguaje.

Esta materia ha adoptado ciertas reglas basadas, en parte, en el documento del comité formado en los laboratorios Indian Hill de AT&T.

A lo largo del dictado de la materia, se irán dando a conocer dichas reglas, en el momento oportuno.

Reglas sobre la Organización de Archivos

Cada una de las líneas, dentro de un archivo, no debería tener más de 79 columnas (ya que no todas las terminales se manejan bien al pasar este límite).

Si sobrepasar este ancho es el resultado de mucha indentación, algo malo está ocurriendo con la organización de dicho código.

Reglas sobre el Nombre de Archivos

Se considera que el nombre de un archivo puede pensarse formado por una parte fija, llamada prefijo o base, y una parte opcional, compuesta de un punto y un sufijo.

Los nombres de los archivos deben comenzar con un caracter letra, continuando con letras o números, sin sobrepasar el límite de 8 caracteres para la parte prefija. Dichos nombres deberían contener solo letras minúsculas.

Por otra parte, se sugiere que los archivos que contienen código fuentes en lenguaje C tengan el sufijo ***c*** , y que los archivos de encabezado cuenten con el sufijo ***h.***

Es conveniente el uso del archivo nombrado README (en mayúsculas), como resumen del contenido de los archivos que se encuentran en cada directorio.

Reglas sobre el Orden de las Secciones en un Archivos Fuente

1ro: Prólogo que indique una descripción del propósito del contenido en dicho archivo. Debe además contener autor, versión, referencias, etc.

2do: Inclusión de Encabezamientos. No es necesario comentar esta sección, salvo que alguna inclusión no resulte obvia.

3ro: Definición de constantes simbólicas, macros, *typedef* y *enumerativos*, que se apliquen a todo el archivo.

4to: Funciones que conforman el programa. Las mismas deben aparecer en el orden correspondiente al recorrido por niveles, de izquierda a derecha (BreadthFirst), del árbol obtenido en el refinamiento de la metodología Top-Down aplicada.

Reglas sobre los Comentarios

Los comentarios deben explicar qué se está haciendo y no cómo se lo está haciendo (Ejemplo: **/* se calcula el valor medio */** en vez de **/* se suman los valores y se dividen por la cantidad total */**). Sólo resulta de gran utilidad explicar el por qué se implementó de determinada forma un fragmento de código, cuando éste proviene de realizarlo de otra forma y se notó que era inaceptable por su performance, decidiendo el cambio por la actual implementación.

Además, los comentarios deben explicar el significado de los parámetros, las supuestas restricciones y los errores que no se pudieron arreglar

Se deben evitar los comentarios triviales que surgen del código y los que no se corresponden con el mismo (confunden más que si no estuvieran).

Cuando el comentario se refiere a un cierto bloque de código, debe comenzar con una línea que sólo contenga los símbolos /* ocupando las columnas 1 y 2, continuar en cada renglón con una * en la segunda columna (o bien ** ocupando las columnas 1 y 2) precediendo al texto del comentario, y finalizar con una última línea que sólo contenga los símbolos */.

El motivo de este formato se debe a que permite encontrar los bloques de comentarios en un archivo mediante el uso del comando ***grep '^.*'***

Ejemplos:

```
/*
 *    Esto es un hermoso comentario
 *    Sobre el bloque que va a comenzar a continuación
 */
```

```
/*
**   Esto es un hermoso comentario
**   Sobre el bloque que va a comenzar a continuación
*/
```

Los comentarios dentro de las funciones deben estar indentados de la misma forma que el resto del código en donde se encuentran, pudiéndose escribir en una sola línea. Inclusive, si son muy cortos, pueden colocarse al lado del código.

Ejemplos:

```
if (argc > 1)
{
        /*   Se toma el archivo desde la línea de comandos  */
        if (freopen (argv[1] , "r", stdin) == NULL)
        {
        ..........................
```

```
if (a == excepcion)
        b = 1;                  /* Se devuelve el valor VERDADERO */
else
        b = esPrimo(a)     /* solamente se trabaja con números pares  */
```

Tipos de Datos - Constantes y Variables

Introducción

En este apunte se detallan las características del Lenguaje C en cuanto a los tipos de datos que presenta y la posibilidad de trabajar con constantes y variables dentro de los programas.

1. Tipos de Datos

Tipo de un Dato

El tipo de un dato determina el conjunto de valores que puede tomar dicho dato (dominio) y las operaciones que se pueden realizar sobre él.

Las instrucciones de una computadora para operar con números enteros no son las mismas que las utilizadas para reales (recordar Assembler).

Aclaración

Los límites del dominio para un determinado tipo de dato están dados por la arquitectura de la máquina.

Cuando se diseña un lenguaje de computación se tiene en cuenta si se va a exigir que sea tipado o no.

Un lenguaje tipado asegura que las operaciones de un programa se apliquen de manera apropiada.

Un compilador para un lenguaje tipado aprovechará los tipos básicos que ofrece la arquitectura de computadora, pudiendo ofrecer tipos estructurados que se construyen a partir de tipos simples, y pudiendo permitir que los usuarios definan sus propios tipos. Exigir que el programador especifique el tipo de datos con el que va a trabajar permite generar código en forma eficiente y facilita la etapa de compilación.

Los lenguajes tipados pueden ser fuertemente tipados (*strongly typed*) o no, según la efectividad con que evite errores, o sea según cuan estricto es el compilador en el manejo de tipos:

- ***Fuertemente tipado:***
  Sólo acepta expresiones seguras, no se permite asignar un tipo de datos a otro sin una función de conversión previa.
  Ejemplo: Pascal

- ***No fuertemente tipado:***
  Es menos estricto que el anterior, se pueden ver los datos de distintas formas.
  Ejemplo: C

Aclaración

C no es un lenguaje fuertemente tipado.
Su antecesor, el lenguaje B ni siquiera era tipado.

El hecho de que C no sea fuertemente tipado puede producir ciertos problemas (bugs), pero también puede ser considerado como una de sus ventajas.

Ventaja:
A programadores expertos les da mayor flexibilidad.

Desventaja:
Puede ocasionar efectos colaterales debido a la evaluación de una expresión con un valor inesperado.

1.1 Tipos de datos en C

Existen cuatro tipos básicos en C: ***char, int, float, double***

1.1.1 Tipo para enteros

❖ *char*

Utiliza un byte para el almacenamiento. Típicamente sirve para almacenar caracteres (el código correspondiente a los mismos que maneja la máquina, por ejemplo ASCII).

Puede agregársele los cualificadores ***signed/unsigned*** para desviar el rango de lo representable con dicha cantidad de bits.

En el lenguaje C no se estableció un default para el caso de no especificar explícitamente alguno de los dos cualificadores, o sea con un determinado compilador *char* podría ser sinónimo de *unsigned char*, y en otro, *char* podría ser sinónimo de *signed char* .

Importante

Para garantizar portabilidad hay que explícitamente declarar si se lo quiere ***signed*** o ***unsigned*** (nunca omitirlo)

❖ *int*

Su tamaño refleja el tamaño de la palabra de la máquina. Típicamente en arquitecturas de 16 bits es de 16 bits y en las de 32 de bits es de 32 bits.

Se le puede agregar los cualificadores ***short/long*** y ***signed/unsigned.***

El par ***short/long*** se utiliza con la intención de poder trabajar con enteros de diferente tamaño. Lo único que asegura ANSI C es la siguiente relación

16 bits £ tamaño(short int) £ tamaño(int) £ tamaño(long int)

32 bits £ tamaño(long int)

Ejemplo:

Para DOS (FAT 16):	tamaño del char = 1 byte tamaño del short int = 2 bytes tamaño del int = 2 bytes tamaño del long int = 4 bytes
Para Windows NT	tamaño del char = 1 byte tamaño del short int = 2 bytes tamaño del int = 4 bytes tamaño del long int = 4 bytes
Para Linux	tamaño del char = 1 byte tamaño del short int = 2 bytes tamaño del int = 4 bytes tamaño del long int = 4 bytes

El par ***signed/unsigned***, como en el caso del char sólo sirve para desviar el rango de valores posibles. A diferencia de éstos el default para los enteros es signed.

Por último, cuando la palabra int va acompañada por un cualificador puede omitirse la misma

Ejercicio:

Aparear entre sí los tipos de datos que son sinónimos en C

1. int
2. short
3. short int
4. long
5. long int
6. signed int
7. signed short
8. signed short int
9. unsigned
10. unsigned short
11. unsigned int

Respuesta:

1, 6
2, 3, 7, 8
4, 5
9, 11
10

Ejercicio:

Aparear entre sí los tipos de datos que son sinónimos en C

1. char
2. unsigned char
3. signed char

Respuesta:

Ninguno aparea, porque no hay default.

1.1.2 Tipos para Reales

❖ *float*

Sirve para almacenar punto flotante de precisión normal

❖ *double*

Sirve para almacenar punto flotante de doble precisión.
Se le puede agregar el cualificador ***long***, para aumentarle la precisión.

Lo único que asegura ANSI C es la siguiente relación:

tamaño(float) £ tamaño(double) £ tamaño(long double)

Ejemplo:

Para DOS (FAT 16):	tamaño del float = 2 byte tamaño del double = 8 bytes tamaño del long double = 8 bytes
Para Windows NT	tamaño del float = 4 byte tamaño del double = 8 bytes tamaño del long double = 8 bytes
Para Linux	tamaño del float = 4 byte tamaño del double = 8 bytes tamaño del long double = 12 bytes

Nota

Los compiladores tienen en los archivos de encabezado limits.h y float.h los rangos para los tipos aritméticos concretos para dicha implementación de arquitectura de máquina.

Importante

La precisión indica cuantos dígitos significativos están disponibles en la representación. Debido a la precisión, en una máquina dos números reales distintos pueden ser considerados iguales (Ejemplo: con 5 dígitos significativos los números 1.0 y 1.000001 son iguales)

2. Expresiones

Expresión

En lenguaje C, una expresión está compuesta de términos y operadores.

Término

- Una constante (valor explícito de un dato invariante)
- Una variable (zona de memoria que contiene un dato que puede cambiar su valor durante la ejecución)
- Una invocación a una función
- Una expresión (entre paréntesis)

Se llama ***evaluación*** al proceso mediante el cual , en tiempo de ejecución, se calcula el valor de cada uno de las operaciones especificadas en una expresión. Cuando una expresión es evaluada, cada operador es aplicado a los datos que están a su alrededor, produciendo un único valor.

2.1 Constantes

Constantes o Literales

Tienen solamente un atributo: **valor,** que no puede cambiar durante la ejecución y que debe pertenecer al dominio de algún tipo de dato soportado por el lenguaje.

2.1.1 Constantes en C

Recordando los tipos disponibles para C, veremos las constantes correspondientes a cada tipo.

❖ Constantes Enteras

Las ***constantes enteras*** pueden ser escritas en los siguiente sistemas de numeración: decimal, octal y hexadecimal.

Para diferenciarlas se utilizan prefijos (no sensibles): **ninguno** para indicar sistema decimal, **cero** para sistema octal y un **cero seguido de una x** para el sistema hexadecimal.

Para poder especificar el tamaño de los tipos de las constantes enteras se utilizan sufijos (no sensible): **U** ó **u** para indicar ***unsigned,*** y **L ó l** para ***long***. Se pueden combinar, pero siempre la U debe preceder a la L.

Para los enteros del sistema decimal, si no se especifica sufijo, el tipo asumido será el primero de la siguiente lista que sea capaz de representar dicho valor:

int
long
unsigned long

Para los enteros notación octal o hexadecimal, de no especificarse sufijo, el tipo asumido será el primero de la siguiente lista que sea capaz de representar dicho valor:

int
unsigned int
long
unsigned long

Ejemplo: Suponiendo arquitectura de 16 bits

32	⇒	int	
32L	⇒	long	
100000	⇒	long	
0x6	⇒	int	(hexadecimal)
0171060	⇒	unsigned int	(octal, 62000 en decimal)

❖ *Constantes Punto Flotante*

Las ***constantes de punto flotante*** pueden ser escritas con punto decimal y/o exponente (e o E)

Para especificar el tamaño de las constantes reales también se especifican sufijos: **F** ó **f** para ***float,*** y **L** ó **l** para ***long double***. Si no se indica sufijo, se asume el default, que es ***double.***

Ejemplo: Suponiendo arquitectura de 16 bits

6.32	⇒	double
6.32F	⇒	float
6.32L	⇒	long double
632 E-2	⇒	double

❖ *Constantes Caracter*

La ***constante de caracter*** es de tipo **int** y se escribe como un caracter entre comillas simples (su valor corresponde a la representación para la máquina, por ejemplo ASCII).

Existen ciertos caracteres que son representados por medio de ***secuencias de escape***, las cuales se ven como varios caracteres, pero en realidad representan sólo uno:

Algunas secuencias de escape son:

Secuencia	*Significado*
\a	**beep**
\b	**backspace**
\f	**comienzo de nueva página**
\n	**comienzo de nueva línea**
\r	**return (retorno al comienzo de la línea actual, sin avanzar)**
\t	**tabulador (se mueve horizontalmente al próximo tab)**
\0	**caracter nulo (caracter con código ASCII cero)**
****	**el propio caracter barra invertida**
\'	**el caracter comilla simple (sólo se usa en caracteres constantes)**
\"	**el caracter comilla doble (sólo se usa en *strings* constantes)**
\ddd	**el caracter cuyo código ASCII es el número octal ddd**
\xhh	**el caracter cuyo código ASCII es el número hexadecimal hh**

❖ *Constantes Cadena*

La ***constante cadena*** es una secuencia de cero o más caracteres, encerrados entre comillas dobles. Las comillas no son parte de la cadena pero son necesarias para delimitarlas.

Internamente se la representa con un caracter por cada uno de los caracteres que la componen, más el caracter extra nulo '\0'. Su tamaño, en cantidad de bytes, es igual a la cantidad de caracteres que contiene más 1.

De acuerdo a lo explicado, no es lo mismo el caracter '**r'** que la cadena **"r"**. En una arquitectura de 32 bits, el tamaño de la ***constante de caracter* 'r'** es de 4 bytes (lo que ocupa un entero en dicha arquitectura), mientras que el tamaño de la ***constante cadena* "r"** es de 2 bytes (1 + 1).

Atención

Las cadenas pueden contener símbolos que deben ser escapados (como ocurre con los caracteres) y en ese caso el símbolo de escape no es un caracter más de la misma.

Ejemplo:

Constante Cadena	*Tamaño (bytes)*
`""`	**1**
`"h"`	**2**
`"hola"`	**5**
`"'hola'"`	**7**
`"\"hola\""`	**7**
`"\\"`	**2**

❖ *Constantes de Enumeración*

La ***constante de enumeración*** es una lista de valores enteros constantes. Por default el primer nombre tiene el valor 0, y cada elemento tiene el valor anterior aumentado en 1. Si se especifica algún valor, la enumeración aumentada en uno comienza a partir de él.

Sintaxis:

enum nombreIdentificatorio { ListaDeConstantes }

Ejemplos

/* Aquí **LUNES** toma el valor 0, **MARTES** el 1, etc. */
enum dias { LUNES, MARTES, MIERCOLES, JUEVES , VIERNES }

/* Aquí **LUNES** toma el valor 1, **MARTES** el 2, etc. */
enum dias { LUNES= 1, MARTES, MIERCOLES, JUEVES , VIERNES }

/* Aquí **ROJO** toma el valor ASCII de la letra 'r', etc. */
enum colores { ROJO='r', AMARILLO='a', AZUL='z' }

Las enumeraciones proporcionan una manera de asociar valores constantes con nombres, parecida a la definición de una constante simbólica, ofreciendo la ventaja de la generación automática.

Ayudan a darle **claridad** al código. Resulta muy conveniente para tener valores de constantes relacionadas entre sí, pudiendo luego utilizarlas en ciclos.

Reglas para las Constantes de Enumeración

- No conviene tener valores consecutivos con "baches" internos.
- Si se usa un enumerativo, el primer número constante debe ser un valor distinto de cero o debe ser un valor que indique error
  Ejemplo:
  enum estado {STATUS_ERR, STATUS_START, STATUS_OK, STATUS_END}
  enum {LUNES=1, MARTES, MIERCOLES, JUEVES, VIERNES}
- Las constantes enumerativas deben empezar con mayúscula o bien estar todas en mayúsculas.

Regla para las Constantes en General

Las constantes deben ser definidas consistentemente con su uso.

Ejemplo: Para una *float* usar 540.0 en vez de 540, así evitará que luego se realice la conversión (casteo) implícita de *int* a *float*.

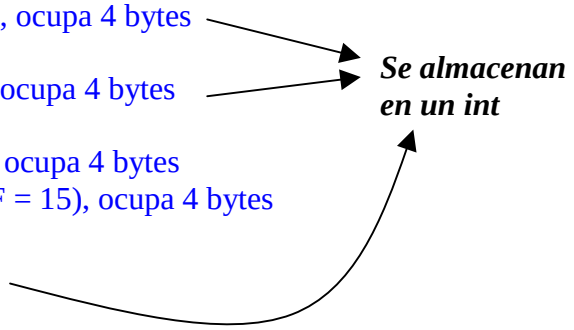
Ejercicio global sobre constantes

Determinar el tipo de las constantes que sean válidas e indicar su tamaño, suponiendo una arquitectura de 32 bits (LINUX):

a) **0.5**
b) **0515**
c) **'\052'**
d) **27,822**
e) **'\xb'**
f) **"x"**
g) **0x1f**
h) **0xful**
i) **"8"**
j) **'8'**
k) **8**

Respuestas:

a) Constante double , ocupa 8 bytes
b) Constante entera (octal 515 = 333), ocupa 4 bytes
c) Constante caracter (ASCII = 42) , ocupa 4 bytes
d) Constante no válida
e) Constante caracter (ASCII = 11), ocupa 4 bytes
f) Constante cadena, ocupa 2 bytes
g) Constante entera (hexa 1F = 31), ocupa 4 bytes
g) Constante unsigned long (hexa F = 15), ocupa 4 bytes
h) Constante cadena, ocupa 2 bytes
i) Constante caracter, ocupa 4 bytes
j) Constante entera, ocupa 4 bytes



2.1.2 Directiva para darle Nombre a la Constantes

Las constantes numéricas, caracter o cadenas aparecen dentro del código formando parte de expresiones.

Existe la posibilidad de asociar un nombre a un valor constante, por medio de la directiva al preprocesador **#define,** que se denomina **definición de constante simbólica**.

En tiempo de preprocesamiento, se produce una sustitución automática en cada lugar donde aparece el nombre dado para la constante, por el valor correspondiente (recordar el ***equ*** de Assembler Z80).

Sintaxis

#define nombre texto

Se puede comenzar con blancos o tabuladores, a los efectos de aplicar indentación (no es necesario comenzar en la primera columna)

Aclaración

Es de buena programación el uso de constantes simbólicas, ya que introducir en el medio del programa literales sueltos suele oscurecer la semántica del mismo y hace a los programas difíciles de mantener.

Ejemplo:

El uso del número 0.21 para el cálculo del I.V.A. en varias partes del programa dificulta la modificación futura para poder agregarle mayor precisión. Es mejor hacer:

#define IVA 0.21

Así, si en otro momento varía el porcentaje del impuesto, sólo se debe cambiar esta línea en vez de todos los lugares donde se la referencia (obviamente hay que recompilar):

#define IVA 0.25

Reglas para las Constantes Simbólicas

Los identificadores para las constantes simbólicas deben estar formados por mayúsculas.

2.2 Variables en C

Variables

Se caracterizan por poseer los siguientes atributos:

- **nombre ®** identificador usado para declararla
- **tipo ®** alguno de los tipos de datos soportados por el lenguaje
- **valor ®** algún valor perteneciente al rango válido de su tipo

Para el atributo ***nombre*** se utiliza un identificador (nombre válido, permitido por el lenguaje).

Identificador

Nombre que se usa para representar variables, constantes, funciones y rótulos del programa.

En lenguaje C, un **identificador** se forma lexicográficamente como una **secuencia de una o más letras, dígitos y guión de subrayado, que no comience con dígito y que no sea una palabra clave (*keyword*)**

El lenguaje C es sensitivo (***case sensitive***): considera que las letras mayúsculas y las minúsculas son distintas.

La longitud de un identificador no está limitada pero sólo los primeros 31 caracteres son tenidos en cuenta por el Compilador (si se resuelven internamente).

Para los nombre resueltos por el Linkeditor sólo se garantiza los primeros 6 caracteres.

Reglas para los Identificadores

- Los identificadores no deben tener *underscore* ni al comienzo ni al final (algunas rutinas y variables del sistema comienzan de esta forma, pudiendo traer problemas y haciendo que los códigos no sean portables).
- Evitar tener identificadores que sólo difieran en una letra, por ser mayúscula en uno y minúscula en otro (Ejemplo: cantidad y Cantidad)
- Evitar identificadores que se parezcan mucho entre sí
- Evitar usar el | y el 1 (en algunas terminales se parecen entre sí)
- Evitar conflictos por usar nombres de la biblioteca estándar
- Los identificadores deben ser nombres significativos
- Si un identificador está formado por más de una palabra, usar mayúscula en el comienzo de cada palabra interna. (Ej.: cantidadDeElementos)

2.2.1 Declaración de Variables en C

Sintaxis de Declaración de una Variable

```
tipo    nombre;                  /* declaración */
ó bien
tipo    nombre = valor inicial;  /* declaración e inicialización */
```

Muy Importante

Antes de utilizar una variable, se debe especificar explícitamente su tipo. Esto se denomina *declaración de la variable.*

Ejemplo:

```
int     cantidad, acumulador = 0,  edad;
float   sueldo;
char    letra;
float   delta = 0.001F;
```

2.2.2 Palabras Clave (Keywords)

Son identificadores predefinidos por el lenguaje. Estas palabras son reservadas y los programas de usuarios no las pueden definir.

Palabras claves establecidas por **ANSI C:**

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

Nota:

Observar qué chico es el lenguaje que, de las 32 palabras reservadas ya conocemos 9 con sólo hablar de tipos de datos (más del 25%).

Operadores y Expresiones

Introducción

En el presente apunte se detallan los tipos de operadores existentes en lenguaje C y se ejemplifica la forma de armar expresiones con los mismos.

1. Operadores en C

Operadores

Símbolos que conectan términos dentro de una expresión e indican un cálculo.

Los operadores pueden ser clasificados según diversos criterios:

- Según el **tipo de operaciones** que realizan (aritméticas, lógicas, etc.)
- Según su **aridad**, o sea la cantidad de operandos sobre los que es aplicado (unarios, binarios, etc.)

En tiempo de ejecución, el proceso de realizar cada una de las operaciones específicas dentro de una expresión se denomina evaluación. Cuando una expresión es evaluada cada operador se aplica a los valores de alrededor. Después de que todos los operadores han sido aplicados, el único valor que se obtiene es el resultado del cálculo de la expresión, llamado valor de dicha expresión.

Todos los operadores tienen una cierta precedencia respecto del resto: cuando aparecen varios en una expresión, los que tienen mayor precedencia son evaluados antes que los de menor precedencia.

Todos los operadores tienen un orden de evaluación: cuando en una expresión aparecen varios operadores de la misma precedencia, dicho orden de evaluación determina dónde empezar a evaluar.

El lenguaje C nos ofrece los siguientes tipos de operadores, de acuerdo a la primera clasificación dada:

- **Operadores Aritméticos y de Manipulación de Bits**
- **Operadores Relacionales y Lógicos**
- **Operadores de Incremento y Decremento**
- **Operadores de Asignación**
- **Operadores de Dirección e Indirección**
- **Operador Condicional**
- **Operador Coma**
- **Operador Sizeof**
- **Operador de Casteo**

1.1 Operadores Aritméticos

Los siguientes operadores son aplicables a operandos de tipo numérico:

Operador	*Significado*	*Aridad*
-	Opuesto	Unario
+	**Idéntico valor**	Unario
*****	**Multiplicación**	**Binario**
/	**División**	**Binario**
+	**Adición**	**Binario**
-	**Sustracción**	**Binario**

El siguiente operador sólo es aplicable a operandos de tipo entero:

Operador	*Significado*	*Aridad*
%	**Módulo (resto)**	**Binario**

La expresión formada a partir de estos operadores aritméticos
devuelve un tipo numérico.

Ejemplos

- Las siguientes tres expresiones al ser evaluadas devuelven el ***double* 4.5**:

  9.0 / 2
  9 / 2.0
  9.0 / 2.0

- En cambio, la siguiente expresión se evalúa como el ***int* 4**:

  9 / 2

- Por último la siguiente expresión al ser evaluada devuelve el ***int* 1**:

  9 % 2

1.2 Operadores Relacionales

Son aplicables a operandos tipo numérico:

Operador	*Significado*	*Aridad*
<	**Menor**	**Binario**
<=	**Menor o igual**	**Binario**
>	**Mayor**	**Binario**
>=	**Mayor o igual**	**Binario**
= =	**Igual**	**Binario**
!=	**Distinto**	**Binario**

La expresión formada a partir de estos operadores relacionales
devuelve un tipo *int*: 0 si la expresión es falsa, **1** si es verdadera

Ejemplo

- Si se tiene una variable ***int x*** con valor **3**, y otra variable ***double z*** con valor **1.5**, al evaluar las siguientes expresiones:

  x != 4 se obtiene el valor ***int* 1**

  x < z se obtiene el valor ***int* 0**

  x <= x se obtiene el valor ***int* 1**

1.3 Operadores Lógicos

A diferencia de otros lenguajes, el lenguaje C no posee un tipo de dato para el dominio { TRUE, FALSE }. Esto dificulta el entendimiento de la naturaleza de las decisiones lógicas.

Para lenguaje C, todo valor **igual a cero** es interpretado como ***FALSE*** y todo valor **distinto de cero** es interpretado como ***TRUE***.

Los operadores lógicos son aplicables a operandos tipo numérico:

Operador	*Significado*	*Aridad*
!	**Not**	**Unario**
&&	**And**	**Binario**
||	**Or**	**Binario**

La expresión formada a partir de estos operadores lógicos
devuelve un tipo *int*: 0 ó **1** según corresponda.

Tablas de Verdad

P	not P
V	F
F	T

P	Q	P and Q
V	V	V
V	F	F
F	V	F
F	F	F

P	Q	P or Q
V	V	V
V	F	V
F	V	V
F	F	F

Leyes de De Morgan

not (P and Q)	es equivalente a	**(not P) or (not Q)**
not (P or Q)	es equivalente a	**(not P) and (not Q)**

Muy Importante

La evaluación de los operadores **&&** y || es **LAZY** (perezosa): deja de evaluar cuando el resultado de la expresión no depende del valor del segundo operando:

En la expresión ***termino1 && termino2*** , sólo se evalúa ***termino2*** si ***termino1*** resulta ser **verdadero**.
En la expresión ***termino1*** || ***termino2*** , sólo se evalúa ***termino2*** si ***termino1*** resulta ser **falso**.

Ejemplo

- Si se tiene una variable ***int x*** con valor **3**, y otra variable ***double z*** con valor **1.5**, al evaluar las siguientes expresiones:

  x < 0 || z >= 10 se obtiene el ***int* 0**

  x && z < 5 se obtiene el ***int* 1**

1.4 Operadores de Manipulación de Bits

Son aplicables a operandos tipo entero:

Operador	*Significado*	*Aridad*
~	**Complemento a 1**	**Unario**
<<	**Decalaje a izquierda (*)**	**Binario**
>>	**Decalaje a derecha (*)**	**Binario**
&	**And a nivel de bit**	**Binario**
|	**Or a nivel de bit**	**Binario**
^	**Xor a nivel de bit**	**Binario**

(*) El operando de la izquierda es decalado la cantidad de bits especificado por el operando de la derecha.

Ejemplo

- Si se tiene una variable ***unsigned x*** con valor **9** y otra variable ***unsigned z*** con valor **1**, al evaluar las siguientes expresiones:

  x & z se obtiene el ***int* 1**

  x | z se obtiene el ***int* 9**

  x ^ z se obtiene el ***int* 8**

1.5 Operadores de Incremento y Decremento

Son aplicables a operandos de tipo numérico:

Operador	*Significado*	*Aridad*
++	**Pre/post incremento en 1**	**Unario**
--	**Pre/post decremento en 1**	**Unario**

El momento en el cual se realiza el incremento o decremento depende de qué lado del operando está colocado el operador:

❖ Si el operador está a la izquierda, el valor de la expresión es el valor del operando ya incrementado/decrementado en 1.

❖ Si el operador está a la derecha, el valor de la expresión es el valor del operando, recién después de usar el valor el operando es incrementado/decrementado en 1.

La expresión formada a partir de estos operadores
devuelve el mismo tipo del operando

Muy Importante

El **operando** al cual se le aplique este operador debe ser un **variable** (***l-value***), es decir un operador que tenga alocado un **almacenamiento en memoria** (no puede ser constante).

Ejemplo

- Si se tiene una variable ***int x*** con valor **9** y otra ***double z*** con el valor **12.5**, al evaluar las siguientes expresiones:

  x++ se obtiene el ***int* 9**. El efecto colateral es que la variable ***x*** cambia su valor por **10**.

  - -z se obtiene el ***double* 11.5**. El efecto colateral es que la variable ***z*** cambia su valor por **11.5**

1.6 Operadores de Asignación

Shorthand Asignment

Operador	*Significado*	*Aridad*
=	**Asignación**	Binario
+=	**Asignación de incremento**	**Binario**
-=	**Asignación de decremento**	**Binario**
*=	**Asignación de multiplicación**	**Binario**
/=	**Asignación de división**	**Binario**
%=	**Asignación de módulo**	**Binario**
<<=	**Asignación de decalaje izq.**	**Binario**
>>=	**Asignación de decalaje der.**	**Binario**
&=	**Asignación de and de bits**	**Binario**
|=	**Asignación de or de bits**	**Binario**
^=	**Asignación de xor de bits**	**Binario**

La expresión formada a partir de estos operadores
devuelve un tipo igual al del operando izquierdo

Nota

Las asignaciones *shorthand* **son una forma abreviada de operar y reasignar:**
Term1 op= Term2 **es equivalente a** Term1 = Term1 op Term2

Muy Importante

El **operando izquierdo** debe ser un **variable** (***l-value***), es decir un operador que tenga alocado un **almacenamiento en memoria** (no puede ser constante).

Los operadores de asignación son una muestra de que el lenguaje C está dentro del paradigma imperativo: se precisa poder cambiar el contenido de memoria.

La asignación es una operación destructiva, por cuanto el valor que posee una variable se pierde en cada nueva asignación.

Ejemplo

- Si se tiene una variable ***double precio*** con valor **120.0** al evaluar la expresión:

  precio *= 2 se obtiene el ***double*** **240.0**

- Como el operador asignación se asocia de derecha a izquierda, se permite la llamada asignación múltiple. Si se tienen las variables ***int x, int y, int z,*** se puede hacer:

  x = y = z = 5 obteniendo como resultado el ***int*** **5**, y como efecto colateral la asignación de dicho valor a las tres variables

1.7 Operador Sizeof

Operador	*Significado*	*Aridad*
sizeof	**Número (**entero sin signo**) de bytes requeridos para almacenar un objeto con el tipo del operando**	**Unario**

Aclaración Importante

El operando puede ser un tipo o una expresión. Cuando se trata de una expresión **no la evalúa** para producir el resultado.

Ejemplo

- Si se tiene una variable ***int x*** con el valor **7** y se evalúa la siguiente expresión:

  sizeof(x = 9) se obtiene el ***int*** **4** en una arquitectura de 32 bits ó el valor ***int*** **2** en una arquitectura de 16 bits, pero en cualquiera de los dos casos el valor de x sigue siendo **7** al finalizar la evaluación

1.8 Operador de Casteo

Operador	*Significado*	*Aridad*
(tipo)	**Convierte el tipo de la expresión al tipo indicado entre paréntesis**	**Unario**

Muy Importante

El operador de casteo NO cambia el tipo del operando sobre el cual está aplicado

1.8.1 Conversión de tipos (promoción/democión)

Es posible combinar valores de distintos tipos numéricos y el lenguaje C los maneja usando "conversión de tipos automática". Dicho proceso consiste en convertir valores de un tipo a otro, compatible, como parte implícita del proceso del cálculo.

Ejemplo 1:

En la expresión **3 + 2.1** se mezclan valores de dos tipos numéricos distintos, ***int*** y ***double***. Para realizar dicha adición se convierte automáticamente el entero **3** a la representación punto flotante **3.0**. Se dice que el valor **3** fue **promovido** (*promoted*)

Ejemplo 2:

Se tiene la variable ***double sueldo***. En la siguiente expresión de asignación también se convierte el valor ***int* 1000** al ***double* 1000.0**. El valor **1000** fue **promovido**.

```
sueldo= 1000;
```

Ejemplo 3:

Se tiene la variable ***int parteEntera***. En la siguiente expresión de asignación el valor **25.3** que es de tipo ***double***, se convierte al tipo ***int* 25**. Se dice que el valor **25.3** fue **demovido** (*demoted*)

```
parteEntera= 25.3;
```

A veces la conversión implícita no alcanza y es preciso especificar la conversión deseada en forma explícita, usando el **operador de casteo** (cast). El operador de casteo consiste en encerrar entre paréntesis el tipo al cual se quiere convertir el operando al cual se le aplica la conversión explícita

Ejemplo:

Se tienen las variables ***int dividendo, int divisor*** inicializadas con los valores **9** y **2** respectivamente. Se tiene también otra variable ***double cociente***.

En la expresión **dividendo / divisor**, como el operador "/ " se aplica a dos operandos del mismo tipo entero, se obtiene el resultado **4** de tipo también entero. Luego se hace la asignación a una variable de tipo ***double***, con lo cual el valor de **cociente** queda en **4.0,** ya que en la asignación **4** es **promovido.** Observar que se obtuvo parte fraccionaria cero.

cociente = dividendo / divisor;

Para poder obtener el resultado **4.5** hay que castear alguno de los dos operandos **antes** de que se evalúe la expresión **dividendo / divisor.**

cociente = (double) dividendo / divisor;

ó bien

cociente = dividendo / (double) divisor;

Notar que no serviría hacer el siguiente casteo:

cociente = (double) (dividendo / divisor);

ya que se estaría casteando el resultado que ya es de tipo ***int* 4** a tipo ***double* 4.0**.

Existen reglas compleja y precisas para la conversión de tipos. Leer primero el resumen de ***Kernighan & Ritchie***, en la **página 49** y luego leer detalladamente las secciones comprendidas entre **A6 y A6.5** (inclusive) del **Apéndice A** del mismo libro.

1.9 Operador Condicional

Operador	*Significado*	*Aridad*
Operando1? Operando2: Operando3	Se evalúa **Operando1,** si el mismo es 0 entonces se evalúa sólo **Operando3**, si es distinto de cero entonces se evalúa sólo **Operando2**.	**Ternaria**

La expresión formada a partir del condicional **devuelve el valor** obtenido en la **segunda evaluación** (Operando2 u Operando3)

Importante

Si el segundo y tercer operando son de distinto tipo, se realizan las conversiones aritméticas usuales para hacerlos de algún tipo común, y ese es el tipo que devuelve el condicional.

Aclaración

Si bien no es necesario un paréntesis, suele usarse para darle más claridad:
(Exp1)? Exp2 : Exp3

Ejemplo

- Si se tiene una variable ***int x*** con el valor 17 y otra variable ***double z*** con el valor **12.0** al evaluar la expresión:

  max= x > z? x: z se obtiene el ***double* 17.0**

- Si se tiene una variable ***int suma*** que contiene la acumulación de los n elementos de un conjunto, y el ***int n*** indica la cantidad de los mismos. Con la siguiente expresión se puede obtener una forma "segura" de calcular el promedio:

  promedio= n = = 0? 0.0: (double) suma/n

1.10 Operador Coma

Operador	*Significado*	*Aridad*
Operando1 , Operando2	**Se evalúa de izquierda a derecha, descartando el valor del Operando1 y devolviendo el valor del Operando2.**	**Binario**

La expresión formada a partir del operador coma
devuelve el tipo y el valor del resultado **del operando de la derecha**

Importante

La coma que separa los parámetros en la invocación de una función NO es el operador coma, y por lo tanto no se garantiza la evaluación de izquierda a derecha de los mismos. Si se tratara del operador coma, ninguna función podría recibir más de un argumento

1.11 Operador Dirección

Operador	*Significado*	*Aridad*
&	**Se verá más adelante**	**Unario**

1.12 Operador Indirección

Operador	*Significado*	*Aridad*
*****	**Se verá más adelante**	**Unario**

2. *Precedencia y Asociatividad de Operadores*

Operadores	***Asociatividad***
() []	Izq. a der.
! ~ ++ -- sizeof + *(unario)* - *(unario)* (casteo) *****(indirección) **&**(dirección)	Der. a izq.
* / %	Izq. a der.
+ - (ambos binarios)	Izq. a der.
<< >>	Izq. a der.
< <= > >=	Izq. a der.
== !=	Izq. a der.
&	Izq. a der.
^	Izq. a der.
|	Izq. a der.
&&	Izq. a der.
||	Izq. a der.
?:	Der. a izq.
Todos los operadores de asignación	Der. a izq.
,	Izq. a der.

La precedencia está dada desde la **mayor precedencia** (primeros renglones) hacia la **menor** (últimos renglones).

Muy Importante

La precedencia y asociatividad de los operandos están especificadas completamente, pero el orden de evaluación de las expresiones es indefinida, salvo para las siguientes excepciones:

- **&&** } *Ya que se asegura que son LAZY*
- || }
- **?:** (operador condicional)
- **,** (operador coma)

Reglas sobre Operadores

- No dejar separación entre un operador unario y el operando sobre el cual se aplica.
- Todo operador binario (excepto "." y "->") debe estar separado por un blanco de los operandos sobre los cuales está aplicado.
- Colocar la primera expresión del operador condicional entre paréntesis, para mejorar la claridad del código.

Reglas sobre Expresiones

- Si se considera que una expresión es muy difícil de leer, separarla en varias líneas, haciendo el corte en el operador de más baja precedencia.
- No utilizar expresiones con demasiados paréntesis anidados.
- No es conveniente utilizar expresiones con operadores ternarios (operador condicional) anidados.
- El operador *coma* debe usarse sólo para inicializaciones múltiples u operaciones tales como las que aparecen en el ***for.***

3. Ejercicios para Afianzar Conceptos

Ejercicio 1

Suponiendo arquitectura de 16 bits, indicar valor y tipo para cada una de las siguientes expresiones:

- **23 / 8** — ***Rta: int* 2**
- **23 % 8** — ***Rta: int* 7**
- **8 % 23** — ***Rta: int* 8**
- **3 * 5.0** — ***Rta: double* 15**
- **3 * 5f** — ***Rta: no compila***
- **3 * 5.0f** — ***Rta: float* 15**
- **9 – 5 / 4 – 2** — ***Rta: int* 6**
- **2 + 2 * (2 * 2 – 2) % 2 / 2** — ***Rta: int* 2**
- **10 + 9 * ((8 + 7) % 6)** — ***Rta: int* 37**
- **sizeof(char)** — ***Rta: int* 1**
- **sizeof('a')** — ***Rta: int* 2**
- **4 * ((5 * 6 % 7 * 8) – 9) – 10** — ***Rta: int* 18**

Ejercicio 2

Si la variables ***i, j, k*** son declaradas de tipo int, indicar para cada una de las siguientes expresiones su valor y tipo

- **k = 3.14159** — ***Rta: int* 3 y el efecto colateral es que la variable k queda con el valor 3.**
- **i = (j = 4) * (k = 16)** — ***Rta: int* 64 y el efecto colateral es que las variables i, j, k quedan con los valores 64, 4 y 16 respectivamente.**

Ejercicio 3

Se tiene una arquitectura de 16 bits (tamaño del int: 16 bits).

Explicar detalladamente por qué al evaluar la siguiente expresión NO se obtiene el valor 1000000, o sea el código no funciona en la forma esperada. Proponer una forma se solucionarlo:

long rta = 1000 * 1000

Rta:

Este código no funciona porque la multiplicación se realiza con aritmética de enteros (el tipo de la constante 1000 es ***int***), y el resultado sobrepasa el rango, ciclando antes de ser promovido al tipo ***long*** debido a la asignación. Una solución hubiera sido usar la expresión:

long rta = (long) 1000 * 1000;

Nota:

Tampoco hubiera servido **long c = (long) (1000 * 1000)** porque es lo mismo que esperar a la conversión implícita.

Ejercicio 4

Se tienen dos variables de tipo ***double***: ***gradosCelcious*** y ***gradosFarenheit.*** Se inicializa la variable ***gradosFarenheit*** con algún valor válido que represente una temperatura en grados Farenheit.

La fórmula física para la conversión de una temperatura a otra es la siguiente:

$$C = \frac{5}{9} * (F - 32^{o})$$

Se la codificó por medio de la siguiente expresión, pero al evaluarla siempre se obtiene el valor 0. Explicar cuál es el error y proponer una solución.

gradosCelcious = 5 / 9 * (gradosFarenheit -32)

Rta:

Esta expresión siempre se evalúa como cero porque la división entre 5 y 9 es entera, y el valor que se obtiene de la misma es el ***int 0***. La forma correcta hubiera sido usar la expresión:

gradosCelcious = (double) 5 / 9 * (gradosFarenheit -32)

o bien

gradosCelcious = 5.0 / 9 * (gradosFarenheit -32)

Ejercicio 5

Se tienen declaradas dos variables de tipo ***unsigned char***: ***lower*** *y* ***upper***. La variable ***lower*** se inicializa con la constante **'q'**. Citar todas la conversiones que se realizan implícitamente al evaluar la siguiente expresión:

upper = (lower – 'a') + 'A'

Rta:

Se convierten todos los tipos ***char*** a ***int*** y se los opera con aritmética de enteros. Finalmente el resultado que es de tipo ***int*** se lo castea a ***char*** (demoted) para asignarlo a la variable upper. Siempre que en una expresión tiene lugar un tipo ***char*** o ***short*** se pasa a ***int*** para poder aplicarle operadores.

Ejercicio 6

Se tienen dos variables ***x*** y ***z*** declaradas ***unsigned ,*** inicializadas con los valores **1** y **12** respectivamente.

¿ Cuál es el valor y el tipo de la siguiente expresión?

(x <<= 1) + (z >> x)

Rta:

El operador + no garantiza cuál de los dos operandos será evaluado primero.

Si se evaluara primero la sub-expresión de la izquierda, el resultado de la expresión completa sería *int* 5 (2 + 3) y el efecto colateral sería que la variable x quedaría con el valor 2.

Si se evaluara primero la sub-expresión de la derecha, el resultado de la expresión completa sería *int* 8 (2 + 6) y el efecto colateral sería que la variable x quedaría con el valor 2.

Consejo:

Usar expresiones seguras. No asumir nada que no esté garantizado, porque deja de ser portable.

Ejercicio 7

Se tienen las variables ***unsigned x, z*** inicializadas con algún valor. Explicar en palabras qué quiere calcular la siguiente expresión:

(x != 0) && (z % x == 0)

Rta:

Si ***z*** es múltiplo de ***x*** en forma segura, o sea sin correr los riesgos de una división por cero.

Ejercicio 8

Se tienen la variable ***unsigned year*** inicializada con un valor que representa cierto año. Explicar en palabras qué quiere calcular la siguiente expresión:

((year % 4 = = 0) && (year % 100 != 0)) || (year % 400 = = 0)

Rta:

Si el valor que contiene la variable ***year*** representa un año bisiesto.

Ejercicio 9

Se ha declarado la variable de tipo ***int* valor**, inicializada con el valor **15**. Indicar si la siguiente expresión sirve para evaluar si dicho valor está comprendido entre 1 y 10 inclusive:

0 < valor < 11

Rta:

No sirve para testear si ***valor*** está entre 1 y 10, ya que al evaluar esta expresión formada por dos operadores de igual precedencia, se aplicaría la asociatividad de izquierda a derecha: (0 < valor) < 11. La sub-expresión izquierda se evalúa como 1, y la expresión 1 < 11 se evalúa como 1. Pero obviamente 15 NO está en el rango pedido y debió haberse obtenido como resultado de dicha expresión el valor 0 (interpretado como FALSE).

La forma correcta hubiera sido:

0 < valor && valor < 11

Ejercicio 10

Decir si son equivalentes las siguientes expresiones (x variable de tipo int):

- **!(x == 2 || x ==5)**
- **x != 2 || x != 5**

Rta:

NO. Las leyes de De Morgan aplicadas a la primera expresión nos hacen obtener la expresión:

! (x == 2) && ! (x = = 5) ó sea **x != 2 && x != 5**

Ejercicio 11

Indicar para qué valores la variable ***int x*** hacen que la expresión sea interpretadas como verdadera:

(x != 4) || (x != 17)

Rta:

Esta expresión siempre se evalúa verdadera ya que hay dos casos a considerar

a) si el valor de ***x*** fuera distinto de 4, ya daría verdadera la expresión, por ser lazy la evaluación del ||
b) si el valor de ***x*** fuera 4, evaluaría la segunda sub-expresión, pero 4 != 17 se evalúa como 1 y por lo tanto se interpreta como verdadera.

Atención:

Si lo que se quería era evaluar como verdadera cuando x no fuera ni 4 ni 17, debió usarse la expresión **(x != 4) && (x != 17)**

No MENOSPRECIAR este tipo de errores: Por medio de caja blanca estos errores pueden ser corregidos, tenemos pues todas las herramientas necesarias para no dejar que se produzcan.

Ejercicio 12

Se tiene la siguiente definición de constante simbólica:
#define MASK 0x80

Indicar en palabras qué intentan hacer cada una de las siguientes expresiones, considerando la variable ***unsigned* x** inicializada con algún valor y la variable **unsigned nroBit** inicializada con un valor entre 0 y 7 . (Nota: consideraremos el bit 0 como el de más a la derecha).

- **x |= MASK** — ***Rta:*** **Enciende el bit 7**
- **x &= ~ MASK** — ***Rta:*** **Apaga el bit 7**
- **x & MASK** — ***Rta:*** **Devuelve un int con los bits encendidos en los lugares donde la variable y la máscara tienen encendidos los bits en 1**
- **1 << nroBit** — ***Rta:*** **Genera una máscara con el bit deseado encendido**

Ejercicio 13

Indicar qué valor queda almacenado en la variable letra después de la siguiente inicialización:

unsigned char letra= 257

Rta:

Queda en el byte el valor 1 (fijarse en la tabla ASCII). El compilador sólo indica un WARNING sobre el truncamiento de la constante debido a que el lenguaje C es "typed" pero NO "strongly typed"

257 = 1 0000 0001 (9 bits) Þ 0000 0001 (8 bits) = 1

Ejercicio 14

Dado el siguiente programa indicar paso a paso, los valores que va tomando la variable ***x:***

```
#define OPERANDO1    'a'
#define OPERANDO2    0

int
main(void)
{
        int  x = OPERANDO1 && OPERANDO2;
        x = OPERANDO1 || OPERANDO2;
        x =  ! OPERANDO1;
        x =  ! OPERANDO2;
        return 0;
}
```

Rta: **0, 1, 0, 1**

NOTA: No con confundir el operando ! con el ~

Ejercicio 15

Dado el siguiente programa, mostrar la salida del preprocesador e indicar si el resultado obtenido es el esperado:

```
/*  Este programa intenta calcular el volumen de una esfera  */

#define PI      3.14159

int
main(void )
{
        double radio;
        double volumen;
        radio = 5;
        volumen= 4/3 * PI * radio * radio * radio;
        return 0;
}
```

Rta:

a) Salida del preprocesador

```
# 1      "nombreArchivo.c"

int
main(void )
{
        double radio;
        double volumen;
        radio = 5;
        volumen= 4/3 * 3.14159 * radio * radio * radio;
        return 0;
}
```

b) El programa compila pero no se obtiene el resultado esperado debido a que la sub-expresión 4 / 3 se evalúa como el ***int* 1**. Habría que castearla para que sea ***double***.

Ejercicio 16

Dado el siguiente programa, mostrar la salida del preprocesador e indicar si el resultado obtenido es el esperado:

```
#define IVA_A   19
#define IVA_B   1.5
#define IVA_TOTAL      IVA_A + IVA_B


int
main(void)
{
        float precio= 100.75;
        float incremento= precio * IVA_TOTAL / 100;

        precio+= incremento;

        return 0;
}
```

Rta:

a) Salida del preprocesador

```
# 1      "nombreArchivo.c"

int
main(void)
{
        float precio= 100.75;
        float incremento= precio * 19 + 1.5 / 100;

        precio+= incremento;

        return 0;
}
```

b) El programa compila pero no se obtiene el resultado esperado debido a que el preprocesador realiza un reemplazo textual, o sea, en el lugar donde aparece ***IVA_TOTAL*** lo reemplaza por ***19 + 1.5***, pero no hace dicha adición.

Ejercicio 17

Arreglar el programa anterior para obtener el resultado esperado.

Rta:

```
#define IVA_A   19
#define IVA_B   1.5
#define IVA_TOTAL      (IVA_A + IVA_B)

int
main(void)
{
        float precio= 100.75;
        float incremento= precio * IVA_TOTAL / 100;

        precio+= incremento;
        return 0;
}
```

Entrada y Salida de Datos

Introducción

En este apunte se muestran formas básicas para el manejo de los datos de entrada y salida de un programa en Lenguaje C.

1. Entrada y Salida de Datos

En todo programa de computación orientado a usuarios, resulta ser muy importante la entrada y salida de datos.

Todos los lenguajes deben ofrecer instrucciones o funciones para realizar dichas operaciones.

La **entrada de datos** puede realizarse desde:
- Teclado
- Sensores conectados a puertos
- Archivo residente en disco o cinta
- Lectora de tarjetas magnéticas
- Lectora de código de barras
- Etc.

La **salida de datos** puede dirigirse hacia:
- Pantalla
- Impresora
- Plotter
- Etc.

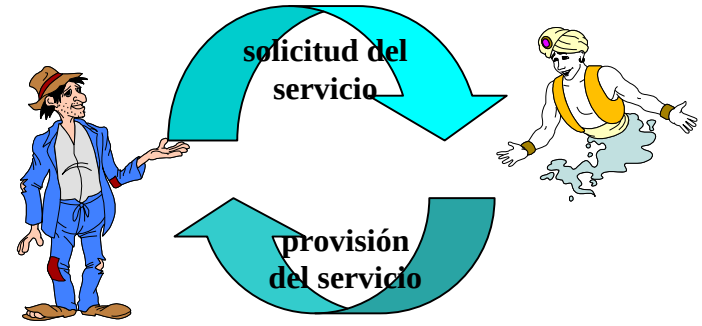
Un programa normalmente está compuesto de dos partes: la que contiene la lógica (procesamiento de cálculo, ***back-end***) y la que ofrece una interfaz amigable al usuario (provee el ingreso y salida de datos al/del sistema, ***front–end***).

En todas las aplicaciones es deseable que el ***back-end*** sea multiplataforma (por lo tanto si están desarrolladas en el lenguaje C seguirán rigurosamente la especificación ANSI C), en cambio el ***front-end*** puede ser desarrollado en cualquier lenguaje, pudiendo inclusive ser Visual (C, Java, Delphi, PowerBuilder, Basic, etc.) con el fin de ofrecer una interfaz amigable y fácil de utilizar.

En las materias ***Programación I*** y ***Estructura de Datos y Algoritmos*** estudiamos los conceptos para poder construir aplicaciones funcionalmente eficientes y portables. Pronto comenzaremos a construir nuestras propias bibliotecas que contendrán la lógica de funcionamiento para dichas aplicaciones, motivo por el cual seguimos **estrictamente ANSI C**.

La ventaja de mantener separada la capacidad de procesamiento y la interfaz reside en:

- El ***back-end*** puede ser multiplataforma, por lo tanto se puede ofrecer la misma aplicación para varias arquitecturas de computadora, porque lo único que habría que re-escribir sería la delgada capa ***front-end***
- Se puede pasar fácilmente a una arquitectura **cliente/servidor** donde la capa ***back-end*** se ejecute en una computadora (***server***) y las capas ***front-end*** se ejecuten en ***workstations.*** Sólo habría que agregar algunas funciones para el protocolo de red que comunique dichas máquinas.



cliente (front-end) ***servidor (back-end)***

1.1 Redireccionamiento de Entrada y Salida

Los sistemas operativos modernos consideran al **teclado** y a la **pantalla** como archivos. Esto es razonable ya que el sistema puede leer desde el teclado de la misma forma que lo hace desde un archivo que yace en disco o cinta. Análogamente, la escritura en pantalla se hace del mismo modo que la escritura en un archivo de disco o cinta.

Redireccionamiento de la entrada o la salida

Acción de cambiar la entrada estándar o la salida estándar, respectivamente

Supongamos ahora un programa llamado ***proof*** que lee desde teclado y escribe en pantalla. Si quisiéramos que los datos entraran desde un cierto archivo llamado ***datos.txt***, podríamos usar el mismo programa siempre y cuando le indiquemos al sistema operativo que reemplace la entrada estándar (teclado) considerada como un archivo, por otro archivo llamado ***datos.txt***.

Se podría hacer algo parecido si se quisiera que la salida de datos no fueran a la salida estándar (pantalla) sino a un cierto archivo llamado ***out.dat***.

Los sistemas operativos que permiten hacer redireccionamiento proveen una forma de indicar al intérprete de comandos sobre qué programa se efectúa el mismo.

En particular, en DOS y en LINUX el redireccionamiento resulta ser muy sencillo:

- Para redireccionar la entrada:

  nombre_del_programa < nombre_del_archivo_de_entrada

- Para redireccionar la salida:

  nombre_del_programa > nombre_del_archivo_de_salida

Ejemplo

En nuestro ejemplo anterior, para redireccionar la entrada de datos, tendríamos que indicar en la línea de comandos:

En DOS: C:> proof < datos.txt
En UNIX: $ proof < datos.txt

Para redireccionar la salida deberíamos indicar:

En DOS: C:> proof > out.dat
En UNIX: $ proof > out.dat

Si quisiéramos redireccionar a ambos, se debe hacer simultáneamente:

En DOS: C:> proof < datos.txt > out.dat
En UNIX: $ proof < datos.txt > out.dat

También se puede conectar directamente la salida estándar de un programa a la entrada estándar de otro vinculando a través de una **tubería** (pipe). Esto se logra a través del siguiente comando:

nombre_del_ programa1 | nombre_del_programa2

siendo programa1 el que posee la salida estándar que debe ser conectada con la entrada estándar del programa2 El sistema operativo manejará todos los detalles para lograr el efecto deseado.

Ejemplo

Si quisiéramos que la salida estándar del programa ***inicial*** pase a ser la entrada estándar del proceso ***final***, deberíamos escribir:

En DOS: **C:> inicial | final**
En UNIX: **$ inicial | final**

Con esto se logra que todos lo datos que el programa ***inicial*** envíe a su salida pasen como entrada al programa ***final.***

Más Ejemplos

Se tiene el archivo ***telefono*** que contiene en cada línea un número de teléfono en cualquier orden.

- Para obtener en la salida estándar los teléfonos ordenados ascendentemente, en la línea de comandos hay que colocar:

En UNIX: **$ cat telefono | sort**
En DOS: **C:\> type telefono | sort**

- Si se quisiera que dicho listado de teléfonos ordenados ascendentemente quedara en un archivo llamados ***orden.txt:***

En UNIX: **$ cat telefono | sort > orden.txt**
En DOS: **C:\> type telefono | sort > orden.txt**

- Para obtener en la salida estándar los teléfonos ordenados descendentemente debería escribirse en la línea de comandos:

En UNIX: **$ cat telefono | sort –r**
En DOS: **c:\> type telefono | sort /r**

Ejercicio 1

Indicar qué se obtendría al escribir desde el intérprete de comandos

En UNIX: **$ sort < telefono | more**
En DOS: **c:\> sort < telefono | more**

Rta:
Se obtiene por pantalla, el contenido del archivo ***telefono*** ordenado ascendentemente y mostrado de a páginas.

Ejercicio 2

Indicar qué se obtendría al escribir desde el intérprete de comandos:

En UNIX: **$ ls | sort | more**
En DOS: **c:\> dir | sort | more**

Rta:
El listado de directorio ordenado ascendentemente y mostrado de a páginas.

1.2 Bufferización

Típicamente en un bajo nivel, existen rutinas dependientes del sistema operativo (no multiplataforma) que manejan los dispositivos particulares de teclado, discos, etc.

Pero los programas ANSI C manejan en forma transparente los archivos, teclado, impresora, etc. Usan las funciones estándares de biblioteca que son independientes de la arquitectura donde se ejecutan. Por ejemplo: cuando un programa intenta leer algún dato de la entrada estándar, el sistema operativo lo hace por medio de un editor orientado a líneas, o sea le deja al usuario borrar caracteres, insertar otros, etc. hasta que presiona la tecla **Return** o **Enter**. Recién ahí el sistema operativo hace disponible dichos datos al programa que intentaba realizar lectura de datos. Aunque el programa quiera leer de a un caracter, el sistema operativo realiza una bufferización, y desde dicho buffer (cierta zona de memoria) le envía los valores al programa a medida que este los solicite.

No hay forma portable de leer un carácter desde el teclado sin quedarse a la espera de que el usuario presione la tecla **Return**. Tampoco hay forma estándar de hacer que los caracteres que se tipean en el teclado no sean mostrados en la pantalla (***no echo***).

Esto parecería ser una dificultad, pero en realidad no lo es porque nadie haría un ***front-end*** con pantallas deslizables y manejo de ***mouse*** en ANSI C. Por lo tanto la necesidad de responder inmediatamente cuando se presiona una tecla (sin esperar a que se presione Return) no es nuestro problema ya que no se trata de una implementación multiplataforma. Cuando hablamos de multiplataforma (alcance de esta materia) nos centramos en el ***back-end***.

Por este motivo, no dedicamos tiempo a desarrollar ***front-end*** espectaculares, ya que eso se puede hacer fácilmente en una tarde, una vez que se tiene un verdadero ***back-end*** portable.

1.3 Flujos de Datos

Un flujo de datos (***stream*****)** es una fuente o destino de datos . Existen **flujos de texto** y **flujos binarios.** Los primeros son interpretados como secuencia de caracteres separadas por '\n' y los segundos como secuencia de bytes (no hay formato)

Las arquitecturas no manejan de manera uniforme el caracter de fin de línea. La librería estándar de C traduce el manejo interno de fin de línea al único carácter '\n' del lenguaje C. Así es como cualquier programa C que lea o escriba en un flujo de texto no debe preocuparse por la implementación de dicha computadora, ya que recibe o envía un '\n' cuando quiere una nueva línea.

El problema reside cuando se leen caracteres como secuencia de bytes (en forma binaria, sin traducción). Por ejemplo en **DOS** al leer la secuencia ***13 10*** como flujo binario se obtienen ambos valores, pero si se lo lee como flujo de texto se obtiene un único carácter '\n'. Así es como el tamaño de un archivo no necesariamente se corresponde con la cantidad de bytes leídos por medio de la función ***getchar*** (ver 1.4.1).

En **UNIX** la secuencia ***13 10*** siempre se devuelve de esta forma no importa si el flujo se interpreta como binario o de texto.

Cuando un programa comienza su ejecución, se abren en forma automática, los siguientes **flujos (*streams*)**

- **stdin:** entrada estándar (por defecto → teclado)
- **stdout:** salida estándar (por defecto → pantalla)
- **stderr:** errores estándar (por defecto → pantalla)

Normalmente una aplicación debe leer datos de la entradas estándar y enviar respuestas a la salida estándar, ofreciendo la posibilidad de redireccionamiento.

Ahora bien, la pregunta es dónde enviar la salida de errores que se producen durante la ejecución y no requieren interacción del usuario. Normalmente son errores que han sido contemplados correctamente en la aplicación como probables y se quiere tener registro de los mismos para su posterior análisis. Los errores son enviados al flujo ***stderr***, pero salvo que se especifique lo contrario, este coincide con la pantalla. Es decir un usuario recibiría los mensajes normales de la aplicación mezclados con los anormales, produciendo un desconcierto. Los mensajes de errores que recibe un usuario común deben ser claros, pero quizás el nivel de detalle del error debe dejarse para ser analizado por alguien del área de sistemas (usuario avanzado).

Gracias al redireccionamiento se puede cambiar la salida estándar de errores inclusive desde el intérprete de comandos.

Los siguientes números corresponden a los descriptores de archivos (handles) que se abren para un proceso en ejecución y corresponden a los tres flujos antes comentados:

handle	*flujo (stream)*
0	entrada estándar
1	salida estándar
2	errores estándar

Ejemplo

Si se tiene el siguiente programa que envía en forma discriminada mensajes comunes a la salida estándar y mensajes más explícitos al flujo de errores estándar

```
#include <stdio.h>

int
main(void)
{
        int dividendo= 12;
        int divisor= 0;

        if (divisor == 0)
                fprintf(stderr, "división por cero\n");      /* Error estándar */
        else
                printf("%d", dividendo / divisor);          /* Salida estándar */
        printf("Gracias por usar este software\n");         /* Salida estándar */

        return 0;
}
```

En DOS se puede redireccionar

C:\> proof.exe 2>errores.log

En UNIX se puede redireccionar

$ proof 2>errores.log

Nota: también se podría redireccionar la salida estándar a un archivo llamado salida.txt de la siguiente forma

En DOS se puede redireccionar

C:\> proof.exe 2>errores.log 1>salida.txt

En UNIX se puede redireccionar

$ proof 2>errores.log 1>salida.txt

1.4 Entrada y Salida en Lenguaje C

Las funciones que sirven para E/S no son en sí parte del lenguaje C. Para poder realizar operaciones, el lenguaje ofrece una serie de funciones de biblioteca estándar para trabajar las entradas y salidas de datos.

Cada código fuente que haga referencia a alguna función (o macro) de la biblioteca de E/S, debe contener la directiva al preprocesador:

#include <stdio.h>

para incluir el archivo de encabezado (*header*) de dicha biblioteca. Explicaremos más adelante qué información brindan estos archivos y por qué se los debe incluir.

ANSI define en forma precisa las funciones que conforman la biblioteca, de modo de que sean compatibles en cualquier sistema que trabaje con lenguaje C.

Las entradas y salidas en lenguaje C se manejan como *streams* o flujo de caracteres. La biblioteca estándar conforma un modelo simple de entradas y salidas de flujo de texto.

1.4.1 Entrada de Datos Usando getchar

La entrada de datos más simple es la que consiste en leer un solo caracter por vez desde la "entrada estándar" (normalmente teclado) y se realiza a través de **getchar**. Cada invocación de getchar devuelve el siguiente caracter de la entrada.

Aclaración

getchar puede estar implementado como una macro.

La expresión **c = getchar() ,** luego de la asignación, deja en la variable c el valor leído desde la entrada estándar.

Para saber cuando finaliza la entrada de datos, se trabaja con un valor distintivo, que obviamente no debe coincidir con el código de máquina de ningún caracter (no debe estar comprendido entre 0 y 255). Para normalizar dicha marca se utiliza la constante simbólica EOF (end of file) que generalmente vale –1, pero que cada sistema puede definirla con un valor diferente (el valor de dicha constante se encuentra en el archivo **stdio.h**, pero las aplicaciones, para que sean portables, usarán la constante EOF en vez de su valor).

Debido a que **getchar** devuelve un número entero (por el EOF), la variable que reciba el valor devuelto por **getchar** debe ser de un tipo suficientemente grande para poder almacenarlo: ***int*** en lugar de ***char.***

Muy Importante

Para **ANSI,** la constante **EOF** representa cualquier valor negativo, no necesariamente el valor **–1**.

Ejemplo

Indicar por qué no funciona el siguiente programa que intenta dejar en la variable ***c*** el valor leído desde la entrada estándar y además pretende que en la variable ***final*** quede valor 1 si se leyó EOF y 1 en caso contrario.

```
final= ( c = getchar() != EOF )? 0: 1;
```

Rta: Como **!=** tiene mayor precedencia que la asignación, primero se evalúa **getchar() != EOF** y el valor 0 o 1 resultante se asigna a **c**, que no se lleva el caracter leído.

Ejemplo

Supongamos el siguiente fragmento de código

```
int c1, c2, c3;
.......
c1= getchar();
c2= getchar();
c3= getchar();
.......
```

- Si en la entrada estándar se recibe **ABCD¿**
  las variables quedarán con los siguientes contenidos:
  c1 ¬ A c2 ¬ B c3 ¬ C

  y en el **buffer** quedará **D¿**

- Si en la entrada estándar se recibe **AB¿**
  las variables quedarán con los siguientes contenidos:
  c1 ¬ A c2 ¬ B c3 ¬ ¿

  y el **buffer** quedará **vacío.**

- Si en la entrada estándar se recibe **A¿ D¿**
  las variables quedarán con los siguientes contenidos:
  c1 ¬ A c2 ¬ ¿ c3 ¬ D

  y el **buffer** quedará **¿**

La cátedra provee las funciones ***getint***, ***getfloat y getdouble*** para la lectura de números enteros, de números reales de simple precisión y de números reales de doble precisión desde la entrada estándar.

Para **LINUX**, en **/home/prog1** se encuentran la librería ***getnum.a*** y el archivo de encabezado correspondiente ***getnum.h.***

1.4.2 Salida de Datos Usando putchar

La salida más simple que se puede realizar es con **putchar(int)**, que escribe un caracter por vez en la salida estándar. El caracter que escribe es el que corresponde al código de máquina indicado entre paréntesis. Si ocurre algún error, putchar devuelve el valor EOF.

Aclaración

***putchar* puede estar implementado como macro.**

Ejemplo:

El siguiente fragmento coloca en la salida el flujo **PA¿**

```
.....
putchar ('P');
putchar (65);
putchar ('\n');
.....
```

1.4.3 Salida con Formato Usando printf

No existe función estándar en ANSI C para limpiar la pantalla, ni para imprimir caracteres en video reverso, ni para mover el cursor a un determinado punto de la pantalla. Este tipo de cosas no son estándares. Tener en cuenta que un manejo estándar del teclado o de la pantalla (manejar los dispositivos como archivos) hace que no existan funciones portables para tales fines. Estos detalles son sólo necesarios en un ***front-end*** y como ya se explicó antes NO se van a usar por el momento, ya que cualquier lenguaje Visual las posee de forma sencilla de incorporar al final del desarrollo de un proyecto.

Lo que sí nos permite la biblioteca estándar es lograr una salida formateada a través de ***printf***, que provee un poderoso mecanismo para mostrar información.

La función **printf** traduce valores internos a caracteres y los envía a la "salida estándar".

Sintaxis de la invocación:

printf("string de control", expresion1, expresion2, ...);

El primer parámetro es una cadena de caracteres llamada **cadena de formato** y a continuación la lista de datos a imprimir (separados por comas).

La **cadena de formato** contiene:

- **caracteres ordinarios:** son copiados tal cual a la salida estándar
- **especificaciones de conversión:** causan la conversión e impresión de los argumentos que le siguen a la cadena de formato, en forma sucesiva (por orden), reemplazándolos en el lugar correspondiente dentro de la cadena.

La función opera de la siguiente forma: analiza el string de control ***(control string)*** caracter por caracter, e imprime cada uno de sus caracteres en la salida estándar, pero cada vez que aparece un carácter % no lo imprime en la salida estándar sino que lo utilizar para darle formato a la salida. El caracter de porcentaje es tratado en forma especial porque indica el comienzo de un código de formato **(*format code*)**. El código de formato es reemplazado por la primera expresión no utilizada todavía. Dicha expresión es presentada con un formato especial que es indicado precisamente por el caracter que finaliza dicho código de formato. El código de formato también puede contener información sobre el ancho, precisión y alineamiento a utilizar.

Es responsabilidad del programador hacer que el número de porcentajes tengan su correspondiente expresión para ser sustituidos. El compilador NO tiene forma de chequear la correspondencia, y si esto no ocurre el resultado de la ejecución del programa es IMPREDECIBLE

Las especificaciones de conversión comienza siempre con % y terminan con un caracter que indica la conversión deseada, de acuerdo al siguiente cuadro:

Símbolo	***Tipo de dato recibido***	***Conversión de salida***
d, i	int	Número entero con signo en base 10
o	int	Número octal sin signo ni 0 inicial
x, X	int	Número hexadecimal sin signo
u	int	Número entero sin signo en base 10
c	int	Caracter simple
s	char *	Cadena de caracteres
f	double	Número decimal en punto fijo
e, E	double	Número decimal en notación exponencial
g, G	double	Con exponente menor a –4 o mayor a la precisión usa %e, caso contrario usa %f

Entre el % y el caracter de conversión pueden especificarse los símbolos mostrados en la siguiente tabla (colocándolos por orden de aparición):

Símbolo	***Formato***
-	Alineamiento a izquierda
0	Para rellenar con ceros
Un número (ancho)	Se imprime en un campo de al menos este ancho y se completa con blancos a izquierda o a derecha según corresponda. Si el dato supera este ancho, se ignora la especificación.
.	Separa el ancho de la parte entera de la precisión
Un número (precisión)	Nro. de dígitos decimales (para punto flotante), nro. máximo de caracteres a imprimir (para cadenas) o nro. mínimo de dígitos a imprimir (para enteros)

También se pueden aplicar **modificadores de longitud**:

- **h**: para indicar ***short***
- **l**: para indicar ***long***
- **L**: para indicar ***long double***

Ejemplo

Se quiere obtener la siguiente impresión tabular en la salida estándar, correspondiente a **Estado, Área, Cantidad de Forestación y Porcentaje** de la misma:

```
Alabama          50750     33945    66.9%
Alaska          591000    201632    34.1%
Arizona         114000     30287    26.6%
```

Esta salida podría obtenerse por medio de:

```
printf("%-14.14s\t%6d\t%6d\t%4.1f%%\n", "Alabama", 50750, 33945,
                                         33945.0 / 50750 * 100);

printf("%-14.14s\t%6d\t%6d\t%4.1f%%\n", "Alaska", 591000, 201632,
                                         201632.0 / 591000 * 100);

printf("%-14.14s\t%6d\t%6d\t%4.1f%%\n", "Arizona", 114000, 30287,
                                         30287.0 / 114000 * 100);
```

Como se observa, para poder colocar un caracter **%** dentro de la cadena de formato, sin que lo confunda como inicio de código de formato, hay que "escaparlo" con otro porcentaje.

Recordar que la barra invertida es una carácter de escape para que el compilador interprete caracteres en **tiempo de compilación**. En este caso dicho carácter no tiene por qué servir ya que lo tiene que interpretar la función printf en **tiempo de ejecución**.

Para implementar un ancho de campo indicado por una variable (no una constante como "%8d"), y poder especificarlo en tiempo de ejecución, hay que usar **printf("%*d", ancho, valor),** donde el asterisco indica que un valor ***int*** de la lista de argumentos variable debe ser usado para el ancho de campo.

Recomendamos leer el uso de ***printf*** en las secciones **7.2** y **B1.2** del texto de ***Kernighan & Ritchie.***

Ejemplo

```
int  num= 65;

printf("num vale %d*\n", num);
printf("num vale %5d*\n", num);
printf("num vale %-5d*\n", num);
printf("num vale %c*\n", num);
printf("num vale %05d*\n", num);
printf("num vale %f*\n", num);
```

imprimiría:

```
num vale 65*
num vale    65*
num vale 65   *
num vale A*
num vale 00065*
IMPREDECIBLE !!!
```

Ejemplo

```
float  num= 18.49;

printf("num vale %f*\n", num);
printf("num vale %5.2f*\n", num);
printf("num vale %5.1f*\n", num);
printf("num vale %-5.1f*\n", num);
printf("num vale %5.0f*\n", num);
printf("num vale %d*\n", num);
```

imprimiría:

```
num vale 18.490000*
num vale 18.49*
num vale  18.5*
num vale 18.5 *
num vale    18*
num vale -1610612736*
```

Ejemplo

```
#define cadena  "es una Prueba\0de otra línea"

printf("Esto %s.\n", cadena);
printf("Esto %20s.\n", cadena);
printf("Esto %-20s.\n", cadena);
printf("Esto %0.5s.\n", cadena);
printf("Esto %20.5s.\n", cadena);
printf("Esto  %-20.5s.\n", cadena);
```

imprimiría:

```
Esto es una Prueba.
Esto        es una Prueba.
Esto es una Prueba       .
Esto es un.
Esto                es un.
Esto es un               .
```

Instrucciones de Decisión

Introducción

En todo lenguaje del paradigma imperativo existen instrucciones de control para cambiar el flujo de control secuencial hacia otra secuencia. En el presente documento se detallará la forma de cambiar el control en base a la evaluación de una condición.

1. Proposiciones y Bloques

Proposición

Cualquier expresión seguida de un punto y coma, que indica una acción a realizar.

En C el **punto y coma** no es un simple separador sino que indica la finalización de una proposición.

Cabe aclarar que, aunque una proposición sea legal en C, puede resultar inútil si se pierde el valor de su respuesta.

Ejemplo

```
x + y;            /*  proposición legal, pero inservible  */
n = x + y;        /* proposición legal y útil  */
```

Bloque (proposición compleja)

Conjunto de declaraciones y proposiciones encerradas entre llaves, que forman parte de una unidad coherente.

Se usa un bloque cuando existe una acción que consiste en varios pasos.
Como ya hemos visto anteriormente, la propia función ***main*** está constituida por un bloque.

Sintácticamente, una proposición compleja equivale a una proposición sencilla: en cualquier sintaxis donde se indique ***"proposición"*** puede también usarse un **"bloque".**

Aclaración Importante

No se coloca punto y coma después de una llave que cierra.

Ejemplos

```
a += 4;     /* proposición simple */

{           /* bloque o proposición compleja */
    a= 4;
    b++;
}
```

2. Control de Flujo

En un lenguaje imperativo para una arquitectura de Von Neumann, el flujo de control para la ejecución de un proceso es secuencial: las instrucciones se ejecutan una a una en el orden en que aparecen (recordar el ciclo de CPU, en el cual el fetch ya preparaba el registro PC con la dirección de la próxima instrucción a levantar).

Para cambiar el orden secuencial del procesamiento existen las siguientes proposiciones de control de flujo, que cambian la secuencia actual hacia otra secuencia:

- ***Proposiciones de decisión (o condicionales)***
- ***Proposiciones de repetición (o iterativas)***
- ***Proposiciones de salto***

Cada instrucción de control en C consta de dos partes bien diferenciadas: la ***línea de control***, que comienza con una palabra clave que identifica la naturaleza de la proposición (condicional o iterativa) y que típicamente contiene información adicional que define la operación de control como un todo, y el ***cuerpo,*** que consiste en las proposiciones a efectuar de acuerdo al resultado de la condición de la **línea de control.**

2.1. Proposiciones de Decisión

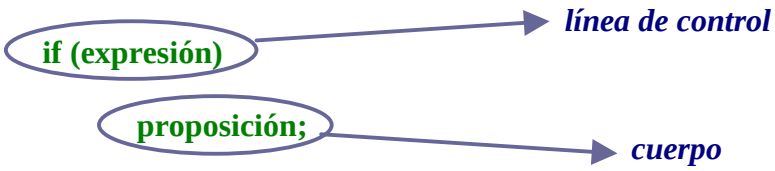
Varían el flujo de ejecución según se cumpla o no una determinada condición (recordar el salto condicional ***jp F, rotulo*** de Assembler)

El lenguaje C ofrece las siguiente proposiciones de decisión:

- ***if***
- ***if- else***
- ***switch***

2.1.1 Proposición if

Sintaxis



Se evalúa la ***expresión de control***, si resulta distinta de cero (verdadera) se ejecuta el cuerpo, en caso contrario no.

La **expresión** de la **línea de control** debe estar siempre entre paréntesis.

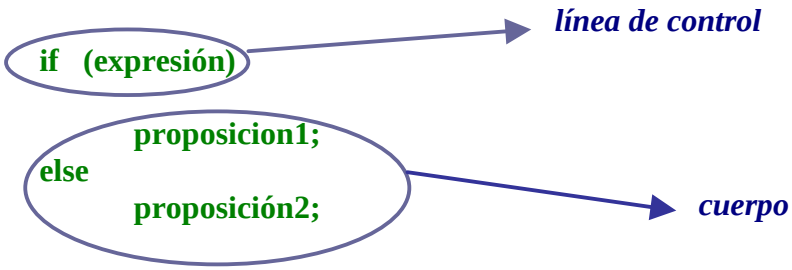
Ejemplo

El siguiente fragmento de código indica si un número es positivo

```
if ( a > 0 )
      printf("Numero positivo\n");
```

2.1.2 Proposición if- else

Sintaxis



Se evalúa la ***expresión de control***, si resulta distinta de cero (verdadera) se ejecuta la ***proposición1***, caso contrario se ejecuta la ***proposición2***.

La **expresión** de la **línea de control** debe estar siempre entre paréntesis.

Ejemplo

El siguiente fragmento de código indica si un número es par o impar

```
if (a%2 == 0)
     printf("Numero par\n");
else
     printf("Numero impar\n");
```

Las proposiciones ***if / if- else*** se pueden **anidar** logrando decisiones múltiples. En este caso se ejecutará la proposición que corresponda a la expresión que resulta verdadera.

En el caso de anidar sucesivos ***if / if-else*** , para evitar ambigüedades, el compilador siempre asocia el ***else*** con el ***if*** anterior más cercano que no posea ***else.***

Ejemplo

En este caso el ***else*** está asociado al ***if interno:***

```
if ( a > 0 )
      if (b > 1)
            x= 5;
      else x= 4;
```

Pregunta:

¿ Cómo lograr asociar el ***else*** con el ***if externo***?

Rta: Utilizando un bloque

```
if ( a > 0 )
{
      if (b > 1)
            x= 5;
}
else x= 4;
```

Muy Importante

Resulta de **pésimo estilo** expresar:

```
if ( a > 0 )
      if (b > 1)
            x= 5;
      else;                    HORRIBLE!!!
else x= 4;
```

ya que se considera de **mala programación** por oscurecer la semántica del programa al indicar que se esta queriendo tomar alguna acción en el caso de que sea a > 0 y b <= 1.

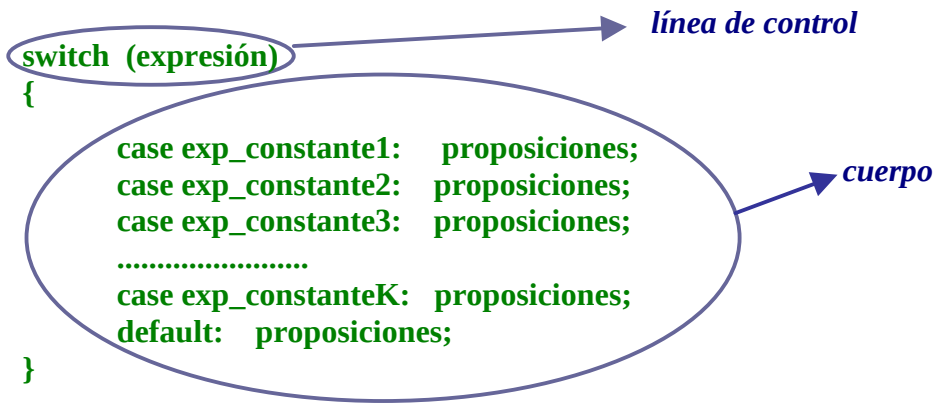
Ejemplo de Anidamiento

El siguiente fragmento de código indica si un número es positivo, negativo o cero:

```
if (a > 0)
      printf("Numero positivo\n");
else  if (a < 0 )
            printf ("Numero negativo\n");
      else
            printf("Numero cero\n");
```

2.1.3 Proposición switch

Sintaxis



La proposición ***switch*** es una decisión múltiple que prueba si la evaluación de una expresión coincide con alguno de los valores constantes enteros especificados por las cláusulas ***case***, y ejecuta **a partir de la primera** proposición rotulada por el ***case*** que se encontró coincidente. Si ninguna de la constantes coincide con el valor de la expresión, se ejecuta las proposiciones rotuladas por la cláusula ***default***, que es opcional.

Todas las expresiones constantes de las cláusulas case deben ser distintas entre sí.

Importante

Las expresiones constantes utilizadas para las cláusulas ***case*** pueden contener operaciones, pero éstas deben tener un valor entero constante (resuelto en tiempo de compilación). No pueden ser variables, ni tipos distintos de enteros.

Consejo

Aunque la cláusula ***default*** es optativa, para evitar la posibilidad de ignorar algún caso inesperado, resulta de buena programación incluirla en toda proposición ***switch,*** aunque se crea haber contemplado todas las posibilidades,

De acuerdo a la limitación en el uso de las expresiones constantes utilizadas en las cláusulas ***case***, si bien la proposición ***switch*** puede hacer los programas más claros, no siempre puede ser utilizada: no sirve para cadenas de caracteres, ni para números reales. En estos casos hay que utilizar proposiciones ***if*** anidadas.

Ejemplo

El siguiente código imprime el nombre de un color identificado entre 0 y 7, de acuerdo a su descomposición en potencias de 2, donde los colores primarios son 1,2 y 4.

```
enum coloresPrimarios {RED = 1, YELLOW, BLUE = 4};
int  color;
.........................
switch (color)
{
      case RED:
            printf("rojo\n");
      case YELLOW:
            printf("amarillo\n");
      case BLUE:
            printf("azul\n");
      case RED + YELLOW:
            printf("naranja\n");
      case BLUE + YELLOW:
            printf("verde\n");
      case BLUE + RED:
            printf("violeta\n");
      case RED + BLUE + YELLOW:
            printf("negro\n");
      default:
            printf("blanco\n");
}
```

Supongamos ahora que el contenido de la variable ***color*** es **5**, ¿qué ejecutaría el fragmento de código anterior?

Rta:

Aparecería en pantalla

```
violeta
negro
blanco
```

Sin duda no es el resultado deseado

¿Cómo hacer para salir del switch, una vez que se ha pasado por la cláusula deseada?

2.2 Proposiciones de Salto Incondicional

Varían el flujo de ejecución en forma incondicional (recordar el salto incondicional ***jp rotulo*** de Assembler)

El lenguaje C ofrece las siguiente proposiciones de salto incondicional:

- ***break***
- ***continue***
- ***goto***

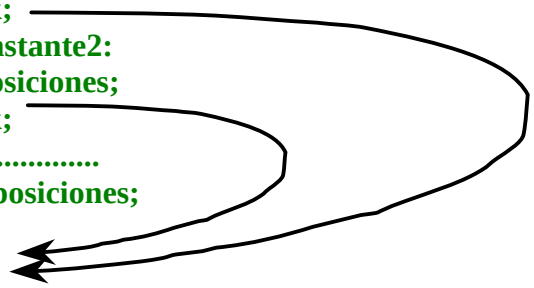
2.2.1 Proposición break

Provoca una salida inmediata (anticipada) de la proposición ***switch*** y de los ciclos ***while***, ***for*** y ***do-while*** (que veremos más adelante)

En un ***switch*** el ***break*** resulta **muy útil** para salir una vez que se ha llegado al ***case*** correcto, si es que no se quiere seguir ejecutando las proposiciones restantes. Con el ***break***, el programa continúa con la proposición siguiente al ***switch.***

Para un switch, el break es un salto de control incondicional deseable

```
.....................
switch (expresión)
{
        case exp_constante1:
                proposiciones;
                break;
        case exp_constante2:
                proposiciones;
                break;
        ..................................
        default: proposiciones;
}
.....................
```



Ejemplo

Mejoramos nuestro ejemplo para que sólo imprima el color correspondiente:

```
enum coloresPrimarios {RED = 1, YELLOW, BLUE = 4};
.........................
switch (color)
{
        case RED:
              printf("rojo\n"); break;
        case YELLOW:
              printf("amarillo\n"); break;
        case BLUE:
              printf("azul\n"); break;
        case RED + YELLOW:
              printf("naranja\n"); break;
        case BLUE + YELLOW:
              printf("verde\n"); break;
        case BLUE + RED:
              printf("violeta\n");  break;
        case RED + BLUE + YELLOW:
              printf("negro\n");  break;
        default:
              printf("blanco\n");
}
```

De esta forma si la variable ***color*** tuviera contenido **5** sólo se imprimiría el color deseado:

```
violeta
```

Ejemplo

Imprimir la cantidad de días que posee un mes numerado.

```
switch (mes)
{
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
            printf("31 dias\n");
            break;
      case 4:
      case 6:
      case 9:
      case 11:
            printf("30 dias\n");
            break;
      case 2:
            printf("28 o 29 dias\n");
            break
      default:
            printf("El numero de mes es incorrecto\n");
}
```

Ejemplo

El siguiente fragmento de código imprime el número o la figura de un naipe, según corresponda a su numeración.

```
int naipe;
switch (naipe)
{
      case 1:  printf("as\n"); break;
      case 11: printf("jack\n"); break;
      case 12: printf("reina\n"); break;
      case 13: printf("rey\n"); break;
      default: printf("%d\n", naipe); break;
}
```

3. Consejos sobre Instrucciones de Decisión

Consejo 1

Para preguntar por variables booleanas no hace falta indicar la igualdad, lo cual sería una redundancia.

Ejemplo:

En lugar de preguntar

```
if (flag != 0 )
      printf("es verdadero");
else
      printf("es falso");
```

sería mejor preguntar directamente

```
if (flag )
      printf("es verdadero");
else
      printf("es falso");
```

Consejo 2

No siempre es necesario usar una instrucción ***if else*** para asignarle a una variable un resultado booleano, es mucho más elegante hacerlo directamente usando los operadores correspondientes

Ejemplo:

En vez de

```
if (tecla == 'S' || tecla == 's')
      salir= 1;
else
      salir= 0;
```

es mejor asignar directamente

```
salir= tecla == 'S' || tecla == 's';
```

Consejo 3

Para condiciones excluyentes no usar if paralelos porque es ineficiente (se pierde tiempo evaluando condiciones que ya se saben falsas).

Ejemplo:

No hacer

```
if (sueldo < 300 )
      printf("sueldo inicial");
if (sueldo >= 300 && sueldo < 900 )
      printf("sueldo medio");
if (sueldo >= 900 )
      printf("sueldo aceptable");
```

sino

```
if (sueldo < 300 )
      printf("sueldo inicial");
else        if  (sueldo < 900 )
            printf("sueldo medio");
      else
            printf("sueldo aceptable");
```

Consejo 4

Cuando se trabaja con números reales hay que introducir un **rango de tolerancia** y en vez de utilizar una igualdad o no igualdad, usar un delta (en el sentido matemático propiamente dicho)

Ejemplo:

Si tenemos una variable punto flotante sobre la que hemos estando realizando operaciones, posiblemente no posea el valor que nosotros creemos que tiene. Tal vez pensamos que almacena un 5 y en realidad se guardó un 4.9999999999999999 o un 5.000000000001. Si más tarde queremos tenemos que tomar una decisión en base a si la misma contiene el valor 5 o no, la comparación

```
if (a == 5)
```

será impredecible

En estos casos hay que usar una tolerancia en el sentido matemático: $|a - 5| < 10^{-12}$

```
if ( (a - 5) < 1E-12  &&  (a – 5) > - 1E-12 )
```

4. Ejercicios usando Instrucciones de Decisión

Problema

Al ejecutar el siguiente fragmento de código, aparece en la salida el cartel "**a es positivo**", ¿dónde se encuentra el error ?

```
int a = -3;
if (a > 0 );
      printf( "a es positivo");
```

Rta:

El problema es que la proposición ***if*** no tiene ***cuerpo***, por lo tanto el ***printf*** está fuera de su alcance.

Ejercicio

Indicar cuál es el error de compilación (sintaxis) de las siguientes instrucciones:

- `if a > b { c = 0;}`
- `if (a > b) c = 0 else b = 0;`

Rta:

En la primera se debe usar **(a >b)** y en la segunda el "**;**" en la proposición **c = 0;**

Ejercicio

Explicar por qué el siguiente fragmento de código siempre imprime "mujer", aunque los valores para representar los sexos sean 1 para la mujer y 2 para el hombre:

```
int  sexo;
.............................
sexo = getint("\nIngrese el sexo (1: femenino, 2: masculino)");
if ( sexo = 1)
      printf("\nEs mujer\n");
else
      printf("\nEs hombre\n");
```

Rta:

Porque ***sexo*** siempre vale 1 (hay una asignación en lugar de una comparación)

Ejercicio

Indicar qué es lo que imprime el siguiente fragmento de código:

```
int x = 7, y = 8, z = 2;

if ((z > x) && (z + x / y - y * z == x))
      printf("La primera expresion es VERDADERA.\n");
else
      if ((y > z) || (y % z > x * z))
            printf("La segunda expresion es VERDADERA.\n");
      else
            printf("Ninguna expresion es VERDADERA.\n");
```

Rta:

Se imprime: ***La segunda expresion es VERDADERA.***
Cabe destacar que sólo realiza las comparaciones **(z>x)** y luego **(y>z),** resolviendo ambos casos con dicho resultado (es evaluación LAZY)

Operador condicional vs. if

En muchos casos es más conciso usar el operador condicional en vez de una sentencia ***if***.

Un ejemplo típico es el uso del operador condicional dentro de un ***printf***, para ajustar un cartel de acuerdo al valor de la variable a imprimir, como el caso del singular y del plural:

```
printf ("Tenemos %d libro%s en stock. \n", cant, (cant ==1)? "":"s");
```

obteniéndose para ***cant*** igual a **1** el cartel:

Tenemos 1 libro

y para otros valores:

Tenemos 5 libros

Ejercicio

Escribir un programa que calcule el sueldo semanal de los empleados de una fábrica. Los mismos deben cumplir normalmente 8 horas diarias, de lunes a viernes, a $10.30 la hora, pudiendo trabajar horas extras que se pagan a $17.20 cada una. No se hacen descuentos por trabajar menos horas semanales de las estipuladas.

El programa contiene una variable ***totalHoras*** en la cual se almacena la cantidad total de horas trabajadas en la semana para un determinado empleado, que se ingresa desde la entrada estándar.

Este programa debe imprimir el sueldo indicando por separado sueldo básico y sueldo extra, si lo hubiere.

Rta:

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

#define     PRECIO_HORA_COMUN 10.30
#define     PRECIO_HORA_EXTRA 17.20
#define     HS_DIARIAS        8
#define     DIAS_LABORABLES   5

int
main(void)
{
      int  horasExtra, totalHoras;

      totalHoras = getint("\nTotal de horas:");
      horasExtra = totalHoras - HS_DIARIAS * DIAS_LABORABLES;

      printf("Sueldo basico:\t %.2f\n",
            HS_DIARIAS * DIAS_LABORABLES * PRECIO_HORA_COMUN);

      if (horasExtra > 0)
            printf("Sueldo extra:\t %.2f\n", horasExtra *
                                          PRECIO_HORA_EXTRA);

      return 0;
}
```

Realizaremos a continuación el Testeo de Caja Blanca para la unidad ***main*** del código anterior:

Se calcula a través del enunciado

Se calcula con un seguimiento en papel o ejecutándolo en una computadora

	Rango de Valores	***Ti***	***Valor Esperado***	***Valor Obtenido***	***Conva-lidación***
Cobertura de Sentencias	horasExtra > 0	**T1={45}**	S Base= 412 S Extra= 86	S Base= 412 S Extra= 86	✓
Cobertura de Decisiones	horasExtra>0 ***verdadero***	**T1**	✓	✓	✓
	horasExtra>0 ***falso***	**T2={40}**	S Base= 412	S Base= 412	✓
Cobertura de Límites	horasExtra= 0, 1, 2	**T2**	✓	✓	✓
		T3={41}	S Base= 412 S Extra= 17.20	S Base= 412 S Extra= 17.20	✓
		T4={42}	S Base= 412 S Extra= 34.40	S Base= 412 S Extra= 34.40	✓

Pasó el testeo de caja blanca exitosamente!

Ejercicio

Escribir un programa que, dada una temperatura entera, imprima el deporte apropiado para la misma, según el siguiente cuadro:

Deporte	***Temperatura***
Ajedrez	Menor o igual a -20° C
Ski	Mayor a -20°C y menor o igual a 10°C
Tenis	Mayor a 10°C y menor o igual a 25°C
Golf	Mayor a 25°C y menor o igual que 30°C
Natación	Mayor a 30°C

Rta:

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

int
main(void)
{
      int temperatura;

      temperatura = getint("Ingrese una temperatura entera: ");

      printf("Con esta temperatura le aconsejo jugar: ");

      if (temperatura <= -20 )
            printf("Ajedrez\n");
      else
            if (temperatura <= 10)
                  printf("Ski\n");
            else
                  if (temperatura <= 25)
                        printf("Tenis\n");
                  else
                        if (temperatura <= 30)
                              printf("Golf\n");
                        else
                              printf("Natacion\n");

      return 0;
}
```

¿No sería conveniente usar switch para que no haya tanto anidamiento?

Sería ridículo. Habría que usar **case0:case1:case2.case3: .....case20**:, etc.

Realizaremos a continuación el Testeo de Caja Blanca para la unidad ***main*** del código anterior:

Se calcula a través del enunciado

Se calcula con un seguimiento en papel o ejecutándolo en una

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convali-dación*
Cobertura de Sentencias	t <= -20	**T1={-22}**	Ajedrez	Ajedrez	✓
	-20 < t <= 10	**T2={0}**	Ski	Ski	✓
	10 < t <= 25	**T3={15}**	Tenis	Tenis	✓
	25 < t <= 30	**T4={27}**	Golf	Golf	✓
	30 < t	**T5={35}**	Natacion	Natacion	✓
Cobertura de Decisiones	t <= -20 ***verdadero***	**T1**	✓	✓	✓
	t <= -20 ***falso***	**T2**	✓	✓	✓
	t <= 10 ***verdadero***	**T2**	✓	✓	✓
	t <= 10 ***falso***	**T3**	✓	✓	✓
	t <= 25 ***verdadero***	**T2**	✓	✓	✓
	t <= 25 ***falso***	**T4**	✓	✓	✓
	t <= 30 ***verdadero***	**T2**	✓	✓	✓
	t <= 30 ***falso***	**T5**	✓	✓	✓
Cobertura de Límites	t <= -20 ⇒ t = -21, -20, -19	**T6={-21}**	Ajedrez	Ajedrez	✓
		T7={-20}	Ajedrez	Ajedrez	✓
		T8={-19}	Ski	Ski	✓
	t <= 10⇒ t = 9, 10, 11	**T10={9}**	Ski	Ski	✓
		T11={10}	Ski	Ski	✓
		T12={11}	Tenis	Tenis	✓
	t <= 25⇒ t = 24, 25, 26	**T14={24}**	Tenis	Tenis	✓
		T15={25}	Tenis	Tenis	✓
		T16={26}	Golf	Golf	✓
	t <= 30⇒ t = 29, 30, 31	**T18={29}**	Golf	Golf	✓
		T19={30}	Golf	Golf	✓
		T20={31}	Natación	Natación	✓
	t > 30 ⇒ t >= 31 ⇒ t = 30, 31, 32	**T19**	✓	✓	✓
		T20	✓	✓	✓
		T21={32}	Natación	Natación	✓

pasó el testeo de caja blanca exitosamente

Reglas para Espacios y Tabulaciones

- Usar blancos verticales y horizontales en forma generosa.

- La indentación y el espaciado deben reflejar la estructura del bloque del código. Es aconsejable dejar dos líneas en blanco entre el final del bloque que define una función y el comentario de la siguiente.

- Un cadena de caracteres larga conteniendo operadores booleanos debe dividirse en líneas separadas, cortando antes de un operador booleano.

  Ejemplo:

  En vez de

```
if (a == 5 && total< tope && tope <= MAX && isDigit(n))
.....
```

  usar

```
if (a == 5 && total< tope
     && tope <= MAX && isDigit(n))
   .....
```

- Las palabras clave que son seguidas de expresiones entre paréntesis deben separase del parámetro izquierdo por medio de un espacio, excepto en el caso del operador ***sizeof.***

- Debe existir un espacio después de cada coma, en la lista de parámetros y argumentos.

Reglas para Proposiciones Simples y Bloques

- Debe haber sólo una proposición por línea, al menos que las proposiciones estén altamente relacionadas (como el caso del ***break*** en las cláusulas ***case*** del ***switch***).

- En toda condición de control, cuando el nombre de la variable o función utilizado es claro respecto de su valor de verdad o falsedad, no hace falta usar el operador relacional.

  Ejemplo:

  Usar `if (esNroValido(n))`

  en vez de `if (esNroValido(n) != 0)`

  ó en vez de `if (esNroValido(n) == 1)`

Justamente para poder hacer este uso de variables booleanas, es muy importante que los nombre de las mismas sean significativos y representen sin ambigüedad cuándo resultan verdadera y cuándo falsas.

Ejemplo:
Si una función hace la validación de un número, no usar como nombre para la misma ***numero***, o ***darNumero***, sino ***esNumeroValido.***

- Aunque el lenguaje permite resumir varias acciones en una sola expresión, lo que se debe priorizar es la claridad y la mantenibilidad del código

  Ejemplo:

  Usar
  ```
  a = b + c;
  d = a + r;
  ```

  es mucho más claro que

  ```
  d = (a = b + c) + r;
  ```

- Las llaves de un bloque deben estar siempre en una **línea separada**, y cada proposición del mismo debe estar en línea aparte, e indentada respecto de la llave.

Instrucciones de Repetición

Introducción

En todo lenguaje del paradigma imperativo existen instrucciones de control para cambiar el flujo de control secuencial hacia otra secuencia. En el presente documento se detallará la manera de ejecutar repetidamente una misma secuencia en base a la evaluación de cierta condición que la controla, hasta salir de ella iniciando la ejecución de una nueva secuencia.

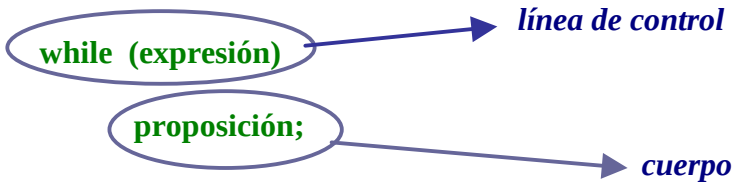
1. Estructuras de Repetición

Permiten cambiar el flujo de control de un programa para repetir una cierta cantidad de veces una proposición determinada. Una vez terminado el ciclo repetitivo se continúa nuevamente con otra secuencia. El lenguaje C ofrece tres estructuras de repetición:

- ***while***
- ***for***
- ***do while***

1.1 Ciclo while

Sintaxis



La proposición se ejecuta repetidamente mientras la ***expresión de control*** resulta ser diferente de 0 (verdadera).

La **evaluación de la expresión de la línea de control** se realiza siempre **antes** de la ejecución de la proposición.

Notas:

- Puede ocurrir que no se ejecute ni una sola vez
- Es responsabilidad del programador lograr que el ciclo se ejecute una cierta cantidad **finita de veces,** ya que de lo contrario no sería un algoritmo. Es importante asegurarse de que, una vez que se haya entrado al ciclo, alguna ejecución del cuerpo cambie la expresión de control al valor cero, para salir del mismo.

Ejemplo

El siguiente programa toma los caracteres de la entrada estándar y los copia a la salida estándar, reemplazando las letras minúsculas por mayúsculas.

```
#include <stdio.h>
#define  DELTA  'a' - 'A'

int
main(void)
{
      int caracter;

      while ((caracter= getchar()) != EOF )
      {
            if ( caracter>='a' &&  caracter<='z' )
                  caracter -= DELTA;
            putchar(caracter);
      }

      return 0;
}
```

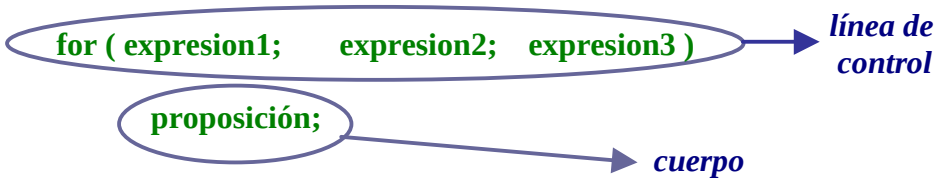
Ejemplo

El siguiente programa cuenta la cantidad de espacios en blancos que hay en un texto de entrada

```
#include  <stdio.h>
#define   BLANCO   ' '

int
main(void)
{
      int caracter, cantidadBlancos=0;

      while ((caracter= getchar()) != EOF )
            if (caracter == BLANCO)
                  cantidadBlancos++;

      printf("Cantidad de blancos en la entrada: %d\n",
                                                cantidadBlancos);
      return 0;
}
```

1.2 Ciclo for

Sintaxis



La ***expresión1*** se evalúa una sola vez al comienzo del ciclo (no hay restricción en cuanto a su tipo), la ***expresión2*** se evalúa en cada iteración y, análogamente al ***while,*** cuando se evalúa como cero termina el ciclo. La ***expresión3*** se evalúa justo antes de volver a comenzar el ciclo.

Cualquiera de las tres expresiones que figuran en el ***for*** pueden estar ausentes, pero los puntos y comas son obligatorios.

De no figurar la ***expresion2*** se evalúa siempre como verdadera.

En el caso de que la ***expresión2*** exista, el ***for*** resulta equivalente a:

```
expresion1;
while (expresion2)
{
        proposición;
        expresion3;
}
```

Pregunta:

De no existir la ***expresión2*** en un ***for***, ¿cuál podría ser el ***while*** equivalente?

Rta:

```
expresion1;
while (1)
{
        proposición;
        expresion3;
}
```

Pregunta:

De no existir ninguna de las tres expresiones del **for**, ¿a qué sería equivalente?

Rta:

```
for ( ; ; )
{
        ..................
}
```

es equivalente a

```
while (1)
{
        ..................
}
```

El ***operador coma*** resulta muy útil cuando se desean asignar valores a diferentes variables en la primera parte del ciclo ***for.***

Recordar que los operandos del operador coma se evalúan de izquierda a derecha y el resultado es del tipo y valor del operando de la derecha.

```
for ( recorrido= 1, cantidad= 0;   cantidad < tope;  recorrido++ )
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^
        expresion1
```

Generalmente, aunque no tiene por qué ser así:

- la ***expresión1*** es una asignación inicial
- la ***expresión2*** es una expresión relacional
- la ***expresión3*** es una actualización de alguna variable contadora.

Ejemplo

Este fragmento de código coloca en la salida la letra **'A'** 6 veces:

```
int
main(void)
{
      int cant = 20;

      for (cant = 4; cant <10; ++cant )
            putchar( 'A');

      return 0;
}
```

Ejemplo

Esto es un ciclo infinito. Inadmisible, pues ***cant*** nunca cambia su valor:

```
int
main(void)
{
      int cant = 20;

      for (cant = 4; cant <10;  )
            putchar( 'A');

      return 0;
}
```

Ejemplo

Esto es un ciclo infinito. Inadmisible, ya que la ***expresión2*** ausente se toma como verdadera:

```
int
main(void)
{
      int cant = 20;

      for (cant = 4; ; ++cant)
            putchar('A');

      return 0;
}
```

Ciclo for vs. Ciclo while

La instrucción ***for*** es similar a la instrucción ***while***, y en muchos casos es cuestión de preferencia del programador cuando usar una u otra.

Aunque todo ciclo ***for*** puede ser re-escrito con un ciclo ***while***, en la línea de control del ***for*** está toda la información necesaria para saber exactamente cuantos ciclos se ejecutarán.

Ejemplo:

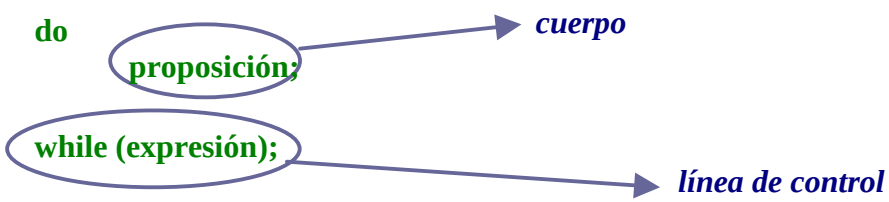
En ambos casos el ciclo se ejecuta 10 veces, pero eso se ve mucho más claro en el ciclo ***for***.

```
for ( i= 0; i<10; i++)
{
      ...
}
```

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

1.3 Ciclo do while

Sintaxis



Es similar al ***while*** con la única diferencia que la evaluación de la expresión para saber si se ejecuta el ciclo o no se realiza al final. Esto implica que seguro se ejecuta por lo menos 1 vez (la primera).

Ejemplo

```
do
{
      printf ("pulse S para abandonar este ciclo \n");
}
while  ( (c=getchar()) != EOF  &&  c != 'S'   &&  c != 's' );
```

2. *Algo más sobre Saltos Incondicionales.*

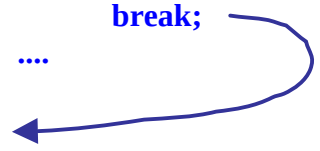
Veremos a continuación el efecto del uso de ***break*** y ***continue*** en las estructuras de repetición.

2.1 *Salto break*

Termina la ejecución del ciclo más anidado que encierre directamente dicha proposición.

Ejemplo 1

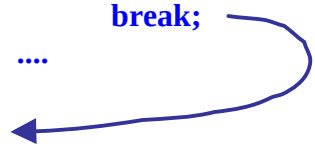
```
while ( expresion1)
{
      if ( expresion2 )
            break;
      ....
}
....
```



De verificarse expresion2, el break haría continuar el flujo de control a lo que sigue al while (corta la ejecución del while)

Ejemplo 2

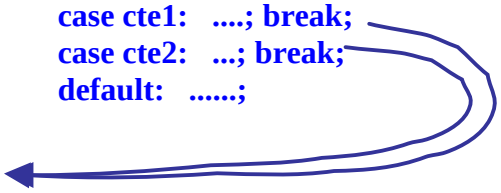
```
while  (expresion1)
{
      for(  ;  ;  )
      {
            if ( expresion2 )
                  break;
            ....
      }
 .....
 }
```



De verificarse expresion2, el break continuaría con el flujo de control que le sigue al for. Sólo corta la ejecución más anidada (sigue dentro del while)

Ejemplo 3

```
while ( expresion1)
{
      switch (expresion2)
      {
            case cte1:   ....; break;
            case cte2:   ...; break;
            default:   ......;
      }
.......
}
```



De ejecutarse algún break, sólo se saldría fuera del switch, pero continuaría dentro del while, ya que estos breaks no están contenidos directamente en el while (sólo sale del más anidado)

2.2 Salto continue

A diferencia del anterior, sólo puede ser usado en iteraciones. Ocasiona que se salte directamente a la ***línea de control*** del ciclo que lo contenga, para poder continuar con la ejecución de dicho ciclo

Esquemáticamente

```
while (expresión)
{
        ....
        continue;
        ....
        ....
}
```

Si se alcanzara este continue, se saltaría a la línea de control y las proposiciones que le siguen al continue serían salteadas

```
do
{
....
....
        continue;
....

} while (expresión)
```

Si se alcanzara este continue, se saltaría a la línea de control del ciclo que lo contiene directamente, salteando las proposiciones que le siguen

```
for ( expresion1; expresion2; expresion3 )
{
        .....
        ....
        continue;
        ....
        ....
}
```

Si se alcanzara este continue, se saltaría a la parte final de la línea de control, se evaluaría la expresión3, y luego se evaluaría la expresión2 para saber si se continúa dentro del ciclo o no

Como se pudo observar, el uso del ***continue*** hace que los ciclos ***for*** *y* ***while*** no funcionen en forma equivalente, ya que en el ***while*** no se evaluaría la ***expresión3*** (si forma parte del cuerpo del mismo y está más abajo que el ***continue*****)** y se pasaría directamente a la evaluación de la ***expresión***, para saber si se continúa dentro del ciclo o no.

Aclaración

El uso de ***continue*** y ***break*** debe ser **prudente.**

Obviamente el uso de ***break*** y ***continue*** no favorece a la programación estructurada, sin embargo pueden ser usados siempre que esto implique una **mejora en el estilo del programación** y no enturbien la semántica del programa, sino que por el contrario la favorezcan.

Existe un esquema básico que sirve para realizar un ciclo, que consiste en:

1) Pedir al usuario el ingreso de un dato
2) Si el dato coincide con un cierto "centinela" prefijado, termina el ingreso de datos, caso contrario se procesa el dato y se sigue en el ciclo.

Si bien dicho esquema podría solucionarse con la siguiente estructura de control:

```
while (1)
{
      printf( "Ingrese el dato [se termina con ... ]" );
      valor = lecturaDato();
      if (valor == centinela)
            break;
      procesarDato();
}
```

claramente, el uso del salto incondicional ***break*** enturbia la semántica del código, y sería mejor reemplazarla por una salida manejada por la ***línea de control*** del ciclo:

```
printf( "Ingrese el dato [se termina con ...]"  );
valor = lecturaDato();

while (valor != centinela)
{
      procesarDato();
      printf( "Ingrese el dato [se termina con ...]"  );
      valor = lecturaDato();
}
```

2.3 Salto goto

Sintaxis

goto rotulo;

donde rotulo es un identificador al estilo

rotulo: proposición;

Todo lenguaje imperativo ofrece una forma de cambiar el flujo de control hacia otra parte del programa en forma incondicional

Aunque el lenguaje C ofrece su uso por medio del goto, **no alienta** su utilización porque los programas que los utilizan dejan de tener una metodología estructurada y oscurecen notablemente su semántica

Muy Importante

Queda terminantemente prohibido el uso de ***goto*** en el dictado de esta materia

3. Problema de la Inicialización de Variables

Todas las variables que estamos usando en los ejemplos y ejercicios de la teoría/práctica y laboratorio son declaradas al comienzo de bloque. Las mismas contienen cualquier información, hasta que en el código se las inicialice o se les asigne algún valor.

Es un ***error muy grave*** asumir que las variables así declaradas empiezan teniendo automáticamente un valor por default: No se les asigna directamente ni cero, ni ningún otro valor especial por el solo hecho de haber sido declaradas. Son creadas en el ***stack*** y contienen cualquier información. Por lo tanto si se las va usar como acumuladores, contadores, etc., habrá que asignarles el valor correspondiente antes de usarlas para tal fin

Ejemplo: *Error muy muy grave*

```
int
main(void)
{
      int cantidadDeLetras;
      int letra;

      while((letra= getchar() ) != EOF)
            cantidadDeLetras++;
      printf("Ud. ha ingresado %d simbolos\n", cantidadDeLetras);
      return 0;
}
```

La forma correcta hubiera sido:

```
int
main(void)
{
      int cantidadDeLetras= 0;
      int letra;

      while((letra= getchar() ) != EOF)
            cantidadDeLetras++;
      printf("Ud. ha ingresado %d simbolos\n", cantidadDeLetras);
      return 0;
}
```

ó bien

```
int
main(void)
{
      int cantidadDeLetras;
      int letra;

      cantidadDeLetras=0;
      while((letra= getchar() ) != EOF)
            cantidadDeLetras++;
      printf("Ud. ha ingresado %d simbolos\n", cantidadDeLetras);
      return 0;
}
```

¿Por qué no hizo falta inicializar o asignarle valor alguno a la variable ***letra***?
Porque ***letra*** toma un valor desde la entrada estándar ANTES de ser usada.

4. Ejercicios usando Instrucciones de Repetición

Ejercicio

Identificar y corregir los errores en los siguientes fragmentos de programas

a)
```
int a=0;
while ( a <= 4)
      resultado += a;
      a++;
```

b)
```
while ( 'Y'  != (c= getchar()) ) ;
      printf ("Pulse Y para abandonar este ciclo \n");
```

c)
```
/* Este código debería imprimir los números del 1 al 10  */
n = 1;
while ( n<10 )
      printf("%d  ", n++);
```

d)
```
/* Este código debería imprimir los números del 1 al 10  */
n = 1;
while ( n <= 10)
      printf ("%d  ", ++n );
```

Rta:

a) Es un ciclo infinito (a++ está fuera del ciclo ***while***)

b) El cuerpo del ***while*** es vacío: cuando se pulsa "Y" aparece el cartel, porque el ***printf*** está fuera del ciclo

c) Imprime: **1 2 3 4 5 6 7 8 9** (se debería preguntar por **n <= 10)**

d) Imprime: **2 3 4 5 6 7 8 9 10 11** (se debería postincrementar **n++)**

Ejercicio

Escribir un programa que imprima los múltiplos de 3 menores o iguales a un cierto número fijo dado.

- *Por medio de un **for***

```
#include <stdio.h>
#include  “getnum.h”
#define MULTIPLO       3

int
main(void)
{
      int tope, recorrido;

      tope = getint(“Tope: “);

      for (recorrido= MULTIPLO; recorrido <= tope; recorrido++ )
            if (recorrido % MULTIPLO == 0 )
                  printf("%d\n", recorrido );

      return 0;
 }
```

- *Una variante más eficiente sería:*

```
#include <stdio.h>
#include “getnum.h”
#define  MULTIPLO  3

int
main(void)
{
      int tope, recorrido;

      tope = getint(“Tope: “) / MULTIPLO;

      for (recorrido= 1; recorrido <= tope; recorrido++ )
                  printf("%d\n", recorrido * MULTIPLO );

      return 0;
 }
```

- *Otra alternativa, utilizando un ciclo **while**:*

```
#include <stdio.h>
#include "getnum.h"
#define  MULTIPLO   3

int
main(void)
{
      int tope, recorrido = MULTIPLO;

      tope = getint("Tope: ");
      while (recorrido <= tope)
      {
            printf("%d\n", recorrido);
            recorrido+= MULTIPLO;
      }

      return 0;
 }
```

✍ *Reglas para Proposiciones y Bloques*

- Si la preposición ***while*** tiene cuerpo vacío, éste debe estar en una línea separada y comentado para que quede claro que no se trata de una omisión.
- Es conveniente que el cuerpo de la proposición ***do-while*** siempre se encuentre entre llaves.
- En las expresiones de control de ciclo NO usar los operadores == ó != para comparar números reales. Utilizar los operadores relacionales <= ó >=, o tratar en caso de que sea posible, de trabajar con número enteros.
- Jamás usar la proposición ***goto***

Ejercicios Globales con Control de Flujo

Introducción

En este documento se presentan distintos ejercicios, aplicando las proposiciones de control de flujo vistas hasta el momento. Creemos de suma importancia observar códigos ya realizados por programadores con experiencia, antes de comenzar a programar los propios. En algunos casos se efectúa el testeo correspondiente, quedando a cargo del alumno realizarlo en el resto de los ejercicios.

Ejercicio 1

Escribir un programa que sume los dígitos de un entero positivo, ingresado desde la entrada estándar.

```
#include <stdio.h>
#include “getnum.h”

int
main(void)
{
      int nro, total= 0;

      do
      {
            nro = getint("\nIngrese un numero entero positivo:");
      }
      while (nro <= 0);

      while (nro > 0)
      {
            total+= nro % 10;
            nro /= 10;
      }
      printf("La suma de sus digitos es %d\n\n", total);

      return 0;
}
```

A continuación, proponemos el testeo con caja blanca:

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convalidación*
Cobertura de sentencias	Con un númro mayor a 0 se pasa por todas las sentecias	**T1={15 }**	6	6	✓
Cobertura de decisiones/ condiciones	(nro<= 0) ***verdadero***	**T2={ -1 }**	entrada inválida	sigue esperando un entero positivo	✓
	(nro<= 0) ***falso***	**T3={ 235}**	10	10	✓
	(nro > 0) en el ciclo while siempre empieza ***falso*** y termina ***verdadero***	**T3**	✓	✓	✓
Cobertura de límites	(nro<= 0) ⇒ tomar –1, 0, 1	**T2**	-	-	-
		T4={0}	entrada inválida	sigue esperando un entero positivo	✓
		T5={1}	1	1	✓

Pasó el testeo de caja blanca exitosamente!

Ejercicio 2

Escribir un programa que imprima en la salida estándar la tabla de multiplicación de los números hexadecimales entre 0 y F, de acuerdo al siguiente formato:

```
   |   1     2     3     4     5     6          .....          E     F
---|-------------------------------------------------------------------------------------------
1  |   1     2     ...
2  |   2     4     6     ...
3  |   3     6     9     C     ...
........ ..... ........ .

F  |   F     1E    2D    ...
```

```
#include <stdio.h>

int
main(void)
{
      int fila, columna;

      printf("   |");
      for(columna= 1; columna <= 0xF; columna++)
            printf("%3X ", columna);

      printf("\n");
      printf("---|");

      for(columna= 1; columna <= 0xF; columna++)
            printf("----");
      printf("\n");


      for(fila= 1; fila <= 0xF; fila++)
      {
            printf("%3X|", fila);
            for(columna= 1; columna <= 0xF; columna++)
                  printf("%3X ", fila * columna);
            printf("\n");
      }

      return 0;
}
```

Ejercicio 3

Escribir un programa que lea frases desde la entrada estándar y las procese de acuerdo al siguiente criterio:

En las frases que se encuentren en las líneas pares se deberá reemplazar cada uno de sus blancos por tabuladores, permaneciendo sin cambios aquellas frases que se encuentren en las líneas impares.

Asumir que las líneas se comienzan a numerar desde 1.

Ejemplo:

Si la entrada fuera

Esta es una
prueba de lo que
se puede
hacer con el procesador pedido

debería salir

Esta es una
prueba de lo que
se puede
hacer con el procesador pedido

```
#include <stdio.h>

int
main(void)
{
      int letra, linea;

      for( linea= 1; (letra=  getchar()) != EOF; putchar(letra) )
            switch ( letra )
            {
                  case '\n': linea++;
                              break;

                  case ' ' : if ( !(linea % 2) )    /* linea par */
                                    letra= '\t';
                             break;
            }
      return 0;
}
```

Ejercicio 4

Escribir un programa que determine si un numero positivo entero, ingresado desde la entrada estándar, es o no primo.

```
#include <stdio.h>
#include “getnum.h”


int
main(void)
{
      int nro, rec;

      do
      {
            nro = getint("\nIngrese un numero entero positivo:");
      }
      while (nro <= 0);


      switch( nro )
      {
            case 2:
            case 3:
                  printf("El numero %d es primo\n", nro);
                  break;

            default:
                  for (rec= 2; rec< nro; rec++)
                      if ( !(nro % rec) )
                            break;

                  printf("El nro %d %s primo\n", nro,
                                          (rec==nro)?"es":"no es");
      }

      return 0;

}
```

Ejercicio 5

Rehacer el ejercicio anterior utilizando la propiedad de la raíz cuadrada (si un número no tiene divisores menores o iguales a su raíz cuadrada y distintos de 1, entonces es primo.

```
#include <stdio.h>
#include “getnum.h”


int
main(void)
{
      int nro, rec;


      do
            nro = getint("\nIngrese un numero entero positivo:");
      while (nro <= 0);


      switch( nro )
      {
            case 2:
            case 3:
                  printf("El numero %d es primo\n", nro);
                  break;

            default:
                  for (rec= 2; rec * rec <= nro; rec++)
                     if ( !(nro % rec) )
                           break;

                  printf("El nro %d %s primo\n", nro,
                                 (rec * rec > nro)?"es":"no es");
      }

      return 0;

}
```

Ejercicio 6

Escribir un programa que imprima en la salida la secuencia de números de ***Fibonacci***, hasta un cierto orden ingresado desde la entrada estándar.

La ***secuencia de Fibonacci*** es un ejemplo de sucesión recurrente, en la cual se fijan dos valores iniciales (0 para el orden 0 y 1 para el orden 1) y el resto se obtiene de como la suma de los valores correspondientes a los dos órdenes anteriores:

F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
.....

```
#include <stdio.h>
#include “getnum.h”

int
main(void)
{
      int orden, rec;
      int fiboAnt= 0, fibo= 1, aux;

      do
      {
         orden= getint("Ingrese el orden de Fibonacci deseado:");
      }
      while (orden < 0);

      for (rec= 0; rec <= orden; rec++)
      {
            switch(rec)
            {
                  case 0: fibo= 0; break;
                  case 1: fibo= 1; break;
                  default:
                           aux= fiboAnt + fibo;
                           fiboAnt= fibo;
                           fibo= aux;
            }
            printf("F(%d)= %d\n", rec, fibo);
      }
      return 0;
}
```

A continuación, proponemos el testeo con caja blanca:

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convalidación*
Cobertura de sentencias	rec = 0 ⎫ rec = 1 ⎬ orden=3 rec = 3 ⎭	**T1={ 3 }**	F(0) = 0 F(1) = 1 F(2) = 1 F(3) = 2	F(0) = 0 F(1) = 1 F(2) = 1 F(3) = 2	✓
Cobertura de decisiones/ condiciones	(orden < 0) ***verdadero***	**T2={-5}**	orden inválido	sigue esperando un orden positivo	✓
	(orden < 0) ***falso***	**T1**	✓	✓	✓
	verdadero y ***falso*** para cada case del switch y para el default	**T1**	✓	✓	✓
Cobertura de límites	(orden < 0) ⇒ (orden <= -1) ⇒ tomar -2, -1, 0	**T4={-2}**	orden inválido	sigue esperando un orden positivo	✓
		T5={-1}	orden inválido	sigue esperando un orden positivo	✓
		T6={0}	F(0) = 0	F(0) = 0	✓

Pasó el testeo de caja blanca exitosamente!

Ejercicio 7

Escribir un programa que lea caracteres desde la entrada estándar (hasta llegar al EOF) y escriba en la salida estándar el mismo texto, pero habiendo removido los excesos de ***espacios en blanco*** y dejando todo el texto en una sola línea, sin líneas en blanco.

Consideramos ***espacio en blanco*** a los espacios, tabuladores y caracteres de nueva línea. Exceso es cuando hay más de uno de ellos seguidos, en cuyo caso se deja sólo el primero, excepto en el caso de fin de línea, que se reemplaza por un blanco y se sigue en la misma línea. Definimos como palabra a toda secuencia de caracteres sin blancos en el medio. A su vez, las palabras se separan unas de otras por medio de secuencias de uno ó más ***espacios en blanco***.

Ejemplo:

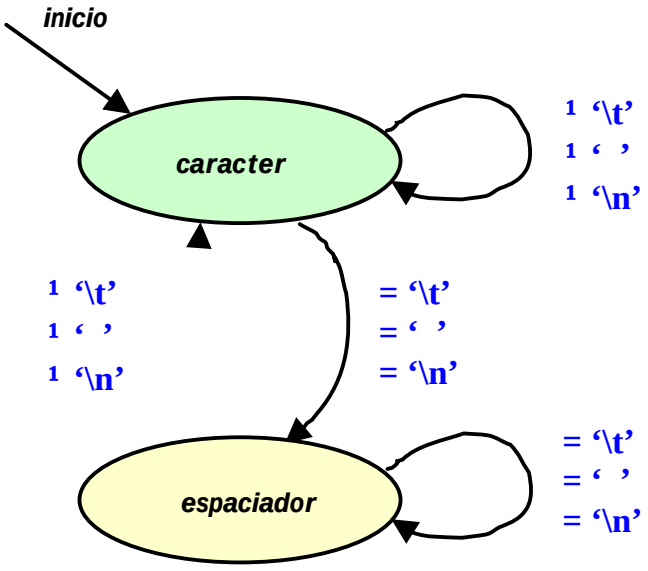
Si entra el texto

```
Esta  es una prueba para           mostrar.
Un mensaje
        que  no                 se


        merece.   Ahora.
```

debe salir

```
Esta  es una prueba para mostrar. Un mensaje que no se merece. Ahora.
```

Para determinar los pasos a seguir en el procesamiento de textos, suele ser muy útil plantear un ***autómata de estados finitos***, en el cual se estipulen los estados posibles y las transiciones entre ellos:



```
#include <stdio.h>

int
main(void)
{
      int estado= 1, letra;


      while ( (letra= getchar()) != EOF)
            switch( estado )
            {
                  case 1:                    /* estado carácter */

                        switch (letra)
                        {
                              case ' ':
                              case '\t':
                                          estado= 2;
                                          break;
                              case '\n':
                                          letra= ' ';
                                          estado= 2;
                                          break;
                        }
                        putchar(letra);
                        break;


                  case 2:               /* estado espaciador */

                        if (letra != ' ' && letra != '\t'
                            && letra != '\n' )
                        {
                              estado= 1;
                              putchar(letra);
                        }
                        break;
            }

      return 0;

}
```

¿Cómo hubiera codificado el ***switch*** interno (dentro del ***case 1***) usando sentencias ***if*** o ***if-else*** ?

Preprocesador - Parte I

Introducción

Muchas de las características útiles del lenguaje C no son implementadas por el compilador sino por el preprocesador. En esta primera parte sobre el preprocesador de C se desarrollan los usos básicos del mismo.

1. Preprocesador

El preprocesador es un módulo que lee un archivo fuente, realiza ciertas acciones y genera una salida que es usada por el compilador de C. Las acciones que el preprocesador realiza son aquellas que están indicadas por directivas comenzando con el símbolo #.

1.1 Macro de Sustitución simple (sin Parámetros) o Definición de Constantes Simbólicas

Sintaxis

#define IDENTIFICADOR textoDeReemplazo

El preprocesador busca toda ocurrencia que aparezca como unidad léxica completa (token) no encerrada entre comillas, desde la definición de la macro hasta el final del lote fuente, y la reemplaza por el texto de reemplazo.

El uso de macros sin parámetros permite obtener códigos fáciles de entender y mantener. Resulta difícil ***descifrar*** la semántica de cierto valor embebido en el medio de un código fuente, ya que el mismo valor en distintos lugares puede tener significados muy diferentes. Cambiar la definición de una constante simbólica implica sólo cambiar una línea de código y recompilar.

Muy Importante

Si una macro se define pero NO SE EXPANDE (no es invocada), entonces NO OCUPA LUGAR.

Separación en líneas

No es bueno usar más de 80 columnas en un código fuente, por lo tanto, si se quiere que el texto de reemplazo ocupe más de una línea, se debe colocar la barra invertida para indicar que el texto continúa en la línea siguiente.

Aclaración

La constante simbólica reemplazada NO puede ser cambiada durante la ejecución del programa. Para cambiarla hay que editar el archivo fuente y recompilarlo nuevamente.

Ejemplo:

Sea el código fuente

```
#define PI      3.1416

int
main(void)
{
        printf("PI vale %g\n", PI);
        return 0;
}
```

esta aparición no será reemplazada porque está entre comillas

esta ocurrencia si será sustituida en forma textual por la definición dada

La salida del preprocesador es

```
int
main(void)
{
        printf("PI vale %g\n", 3.1416);
        return 0;
}
```

Ahora el compilador tomará la salida anterior y generará código objeto, que luego de la linkedición, si todas las fases resultaron exitosas, estará listo para ser ejecutado.

El preprocesador ***no chequea la sintaxis*** que resulta de la sustitución realizada, esa tarea es llevada a cabo por el compilador

Ejemplo:

Sea el código fuente

```
#define PI      = 3.1416

int
main(void)
{
        printf("El valor de PI es %g\n", PI);
        return 0;
}
```

La salida del preprocesador es

```
int
main(void)
{
        printf("El valor de PI es %g\n", = 3.1416);
        return 0;
}
```

Ahora el compilador tomará la salida anterior y NO generará código objeto, por encontrar errores sintácticos.

1.2 Macro con Parámetros

En el inciso anterior analizamos cómo definir nombres que son siempre reemplazados por el *mismo texto*. La macro con parámetros actúa como una ***plantilla*** que se completa en forma diferente cada vez que se la invoca.

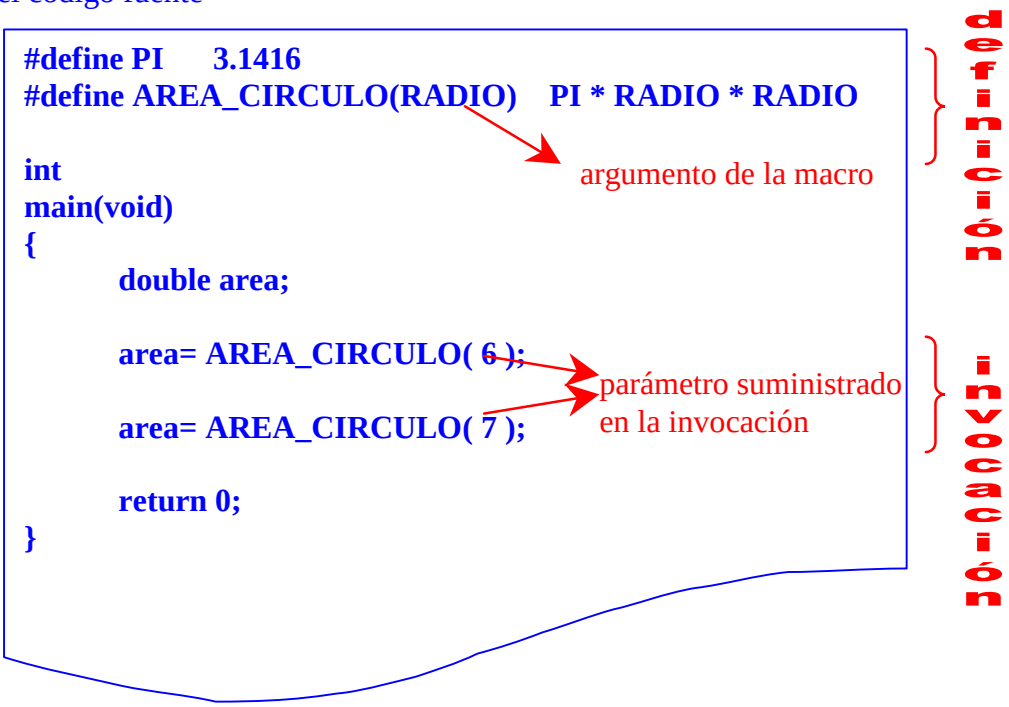
Sintaxis

```
#define IDENTIFICADOR(ARG1, ...ARGN)      textoDeReemplazo
```

El preprocesador busca toda invocación de macro con parámetro que aparece a partir del lugar donde se la definió y hasta el final del lote fuente. Cuando encuentra una *invocación* de macro procede a *macroexpandirla*, o sea, reemplaza la invocación de macro con el texto de definición, y sustituye los argumentos de la macro por los parámetros que se suministraron en la invocación.

Ejemplo

Sea el código fuente



La salida del preprocesador es

```
int
main(void)
{
        double area;

        area=  3.1416 * 6 * 6;

        area= 3.1416 * 7 * 7;

        return 0;
}
```

El programa estará listo para la compilación.

Ejemplo

```
#include <stdio.h>
#define  Leo_Numero_No_EOF(c)   (( (c = getchar()) != EOF) &&  \
                                        (c >= '0') && (c<= '9') )
.....
while  (Leo_Numero_No_EOF (num))
        putchar (num);
.....
```

La salida del preprocesador será:

```
.....
while ((( (num = getchar()) != EOF) &&  (num >= '0') && (num<= '9') ))
    putchar (num);
.....
```

Ejercicio 1

a) Dado el siguiente código fuente, indicar cuál sería la salida del preprocesador
b) Indicar si la compilación resulta exitosa
c) Indicar si el programa ejecuta correctamente

```
#define PI      3.1416
#define AREA_CIRCULO(RADIO)                PI * RADIO * RADIO

int
main(void)
{
        int radioInicial= 3, incremento= 1;
        double area;

        area= AREA_CIRCULO( radioInicial + incremento);

        return 0;
}
```

Rta:

a)

```
int
main(void)
{
        int radioInicial= 3, incremento= 1;
        double area;

        area= 3.1416 * radioInicial + incremento * radioInicial + incremento

        return 0;
}
```

b) Sí, compila exitosamente

c) Ejecuta incorrectamente por la precedencia de operadores
La macro debió definirse

```
#define AREA_CIRCULO(RADIO)            PI *  (RADIO) * (RADIO)
```

Ejercicio 2

Escribir la macro **AREA_RECTANGULO(BASE, ALTURA)** para calcular el área de un rectángulo

Rta:

```
#define AREA_RECTANGULO(BASE, ALTURA)      (BASE) * (ALTURA)
```

Ejercicio 3

Si la macro anterior es invocada en un programa con los parámetros del siguiente código, escribir la salida que deja el preprocesador, e indicar qué se mostraría en la salida estándar en tiempo de ejecución.

```
#define AREA_RECTANGULO( BASE, ALTURA)     (BASE)* (ALTURA)
#define BASE          10

int
main(void)
{
        int lado1= 5, lado2= BASE;

        int superficie= AREA_RECTANGULO( lado1 + 10, lado2);
        printf("La superficie es de %d\n", superficie);

        return 0;
}
```

Rta: La salida del preprocesador sería

```
int
main(void)
{
        int lado1= 5,  lado2= 10;

        int superficie=  ( lado1 + 10) * (lado2) ;
        printf("La superficie es de %d\n", superficie);

        return 0;
}
```

Al ejecutar el programa, se obtendría: **La superficie es de 150**

Ejercicio 4

Escribir la salida del preprocesador del siguiente código fuente

```
#define SUMATORIA(INIT, TOPE, SUMA) {                    \
                                        int i;           \
                                        SUMA = 0;        \
                                        for(i= INIT; i<=TOPE; i++)   \
                                                SUMA+=i;             \
                                    }

int
main(void)
{
        int acumulador= 0;
        SUMATORIA (1, 5, acumulador);
        printf ("Sumatoria de los primeros 5 enteros: %d \n", acumulador);

        return 0;
}
```

Rta:

```
int
main(void)
{
        int acumulador= 0;
        {
                int i;
                acumulador = 0;
                for(i= 1; i<=5; i++)
                        acumulador+=i;
        }

        printf ("Sumatoria de los primeros 5 enteros: %d \n", acumulador);

        return 0;
}
```

¿ Se podría haber invocado a la macro dada en este ejemplo como **SUMATORIA(1, 5, 0)** ?

Por supuesto que NO, ya que el **0,** al ser una constante**, NO es un *l-value* y no tiene dirección de memoria asociada.**

1.2.1 Operadores para Macros

En ANSI C existen dos operadores que pueden aparecer en macrodefiniciones:

- **Operador #**: es unario y hace que una división léxica de texto de reemplazo se convierta en una cadena encerrada entre comillas, es decir, encierra entre comillas dobles el argumento de la macro que es precedido por dicho operador. Es de gran utilidad porque el *compilador* concatena las cadenas separadas por blancos en una única cadena.

- **Operador ##**: es binario y concatena los dos componentes léxicos a los cuales se aplica.

Ejemplo

Sea la macro definición

#define SALUDO(nombre) printf ("\tHola " #nombre ", como estas? \n")

este token será reemplazado por el argumento correspondiente y encerrado entre comillas dobles

Si en un código se invocará la macro como

SALUDO (Ana);

La macroexpansión que hace el preprocesador generaría:

printf ("\tHola " "Ana" ", como estas?" \n");

El compilador concatenaría todos esos strings y en tiempo de ejecución se obtendría

Hola Ana, como estas?

Ejemplo

Si se definen las siguientes macros para debuggear valores de variables:

```
#define DUMP_INT(X)          printf(  #X "=%d\n", X )
#define DUMP_DOUBLE(X)       printf(  #X "=%g\n", X )
```

Al invocar esta macro con

```
DUMP_INT( i)
```

el preprocesador la macroexpandiría como

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

Ejemplo

¿Para qué serviría la siguiente macro?

```
#define DUMP(PREFIJO, SUFIJO)        PREFIJO ## SUFIJO
```

Rta:

Para unir dos tokens léxicos

Ejemplo

Indicar qué se obtendría en cada macroexpansión:

```
#define  INGRESAR_DATO(x,y)          x = get  ##  y( "")
INGRESAR_DATO(edad,int);
INGRESAR_DATO(sueldo,float);
```

Rta:

```
edad = getint( "");
sueldo = getfloat("" );
```

2. Directiva #include

El preprocesador reemplaza la línea con la directiva #include con el contenido del archivo especificado

Existen dos formas para esta directiva, que sólo se diferencian por el lugar en donde el preprocesador busca el archivo a incluir:

- **#include <nombre_de_archivo>**
  Busca el archivo en directorios reservados por el sistema. Generalmente se utiliza para incluir archivos de la biblioteca estándar.

- **#include "nombre_de_archivo"**
  Busca el archivo en el directorio actual de trabajo y si no lo encuentra lo sigue buscando en los directorios reservados por el sistema. Generalmente se utiliza para incluir archivos escritos por el mismo programador.

@ ***Reglas para Macros***

- Los nombres de las macros deben estar en letras **mayúsculas**.

- Suele ser útil definir las siguientes macros para clarificar los valores booleanos:
  ```
  #define  FALSE   0
  #define  TRUE    1
  ```

- Todas las ocurrencias de los argumentos en el texto de reemplazo de una macro deben estar entre paréntesis (para evitar problemas de precedencias después del reemplazo)

- Si el texto de una macro contiene operaciones matemáticas, conviene encerrarlo entre paréntesis, para asegurar la precedencia de sus operadores frente a otros operadores externos.
  Ejemplo:
  ```
  #define  AREA_RECT(a, b)          ( (a) * (b) )
  .......
  calculo = 1 / AREA_RECT(5, 2)
  ```

- Si el texto de una macro contiene varias proposiciones, es conveniente encerrarlo entre llaves.

- No cambiar la sintaxis del lenguaje C usando macros.

Funciones

Introducción

Las funciones son una herramienta que sirve para simplificar la estructura de los programas. Se las puede analizar desde dos puntos de vista:

❖ **Holístico**: Sólo se presta atención a "qué hace" una función. Esta perspectiva sirve para utilizar las funciones en la construcción de bloques de complejidad mayor.

❖ **Reduccionista**: Sólo se estudia "cómo hace" una función su tarea. Esta perspectiva sirve para analizar la implementación de una función.

1. Funciones

Una función es un conjunto de sentencias que reciben un nombre.

Gracias al uso de funciones es que podemos programar nuestras aplicaciones en forma modular y estructurada, por más complejas que éstas sean. El lenguaje C es reducido pero está potenciado por un conjuntos de funciones para diversos fines (entrada/salida, matemáticas, etc.). El ANSI C especificó las funciones que forman parte de la Biblioteca Estándar, para garantizar que las mismas estén disponibles en cualquier compilador ANSI C.

Cada programa C está formado por funciones, no pudiéndoselas definir en forma anidada (no se puede definir una función dentro de otra). Recordar que la propia ejecución de un programa comienza dándole el control a la función llamada ***main***.

1.1 Definición de una función

Sintaxis de Definición

```
tipoQueDevuelve      nombreDeLaFuncion( listaDeParametrosFormales)
{
        /* proposiciones */
}
```

La lista de parámetros, separados por coma, especifica el tipo y el nombre para cada uno de los parámetros. Gracias a la especificación del nombre de un parámetro es que se le puede referenciar para realizar los cálculos correspondientes en el bloque de definición (cuerpo de la función).

Los parámetros que aparecen en la definición de la función se denominan parámetros formales.

Importante

El tipo de los parámetros o el valor que devuelve una función puede ser cualquiera de los ya vistos.

- Si se omite la especificación del tipo, tanto para algún parámetro como para el valor que retorna la función, el compilador asume para el mismo el tipo ***int***.
- Para que una función reciba una lista vacía de parámetros se debe especificar explícitamente la palabra ***void*** como especificación de argumento.
- Análogamente, si se desea que una función no devuelva valor se debe explícitamente especificar que el tipo de retorno es ***void.***

Ejemplo 1:

A continuación se define la función ***sumaDigitos*** que calcula la suma de los dígitos que componen un número entero no negativo.


```
unsigned
sumaDigitos( unsigned numero )
{
      unsigned suma= 0;

      while( numero )
      {
            suma += numero % 10;
            numero /= 10;
      }

      return suma;
}
```

Ejemplo 2:

A continuación se define la función ***dibujaTriangulo*** que muestra en la salida estándar un triángulo invertido generado por asteriscos, donde cada línea contiene dos asteriscos menos que la línea anterior (uno menos a cada lado). La cantidad de líneas a dibujar es el parámetro de tipo ***unsigned*** que recibirá la función.

tipo que se debe especificar para indicar que no retornará valor alguno

lista de un solo parámetro formal

```
void
dibujarTriangulo( unsigned altura )
{
      unsigned ancho = 2 * altura -1;

      for( ; altura; altura--)
      {
            int linea;
            /* dibujo de los blancos en la linea */
            for( linea= ancho/2 - altura + 1 ; linea;  linea--)
                  putchar(' ');

            /* dibujo de los asteriscos en la linea */
            for( linea= 2 * altura - 1; linea;  linea--)
                  putchar('*');

            /* dibujo de los blancos en la linea */
            for( linea= ancho/2 - altura +1 ; linea;  linea--)
                  putchar(' ');

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

¿Es necesario el último ciclo for interno?

Ejemplo 3:

La función ***main*** es un ejemplo de función que no recibe parámetros (En Estructura de Datos y Algoritmos veremos otra variante).

tipo que retorna la función

lista vacía

```
int
main( void)
{
      .....
      return 0;
}
```

1.2 Invocación de una función

Sintaxis de Invocación

- Si la función tiene parámetros:

  nombreDeLaFuncion(listaDeParámetrosActuales)

- Si la función NO tiene parámetros (lista vacía):

  nombreDeLaFuncion( )

Invocar una función consiste en ejecutar las proposiciones que la definen. Para ello se escribe el nombre de la misma, seguida por una lista de expresiones (que puede ser vacía) encerrada entre paréntesis. Dichas expresiones de invocación reciben el nombre de ***parámetros actuales*** o argumentos , y permiten el intercambio aceptable de información entre el módulo llamador y la función invocada.

Debe existir una correspondencia en **tipo y cantidad** entre los parámetros formales y los actuales.

Existe una forma de definir funciones con una cantidad de parámetros variables pero la veremos en Estructura de Datos y Algoritmos por ser de mayor complejidad (pensar el caso de la función ***printf***).

Importante

La coma que aparece entre los parámetros actuales NO es el operador coma. Si lo fuera, sólo se podrían definir funciones con un único parámetro formal (¿por qué?)

Por este motivo, ANSI C **no especifica el orden en que van a ser evaluados los parámetros en el momento de la invocación.**

Una vez invocada la función se le transfiere el flujo de control y ejecuta las proposiciones que la definen (se cambia el flujo de control desde una secuencia hacia otra) y luego el flujo de control retorna al módulo invocador.

La acción que ocurre cuando una función devuelve el flujo de control al módulo invocador se denomina ***retornar***. Si la función además de retornar el flujo de control, devuelve un valor al módulo invocador, se dice que ***retorna un valor***.

Para que la función retorne al módulo invocador, se utiliza la proposición ***return***.

Sintaxis para Retornar

- Si sólo se quiere retornar al módulo invocador:

  return;

- Si se quiere retorna al módulo invocador un valor:

  return (expresión);

Recordar en Z80 la instrucciones ***call*** y ***ret*** que se utilizaban para transferir el flujo de control hacia una subrutina y retornar al módulo invocador respectivamente, liberando correctamente el stack.

Aclaración

La proposición ***return*** es un salto incondicional, pero absolutamente necesario para liberar stack y retornar el control al módulo llamador.

Si se omite la proposición ***return*** se sigue ejecutando hasta el final del bloque de definición de la función. Los compiladores suelen generar un **return automático** al final del bloque, si no lo encuentran o avisar con un ***warning*** cuando se definió que la función debía devolver un tipo distinto de ***void***.

Importante

Cuando se alcanza el ***return*** se regresa al módulo invocador.

Si bien el lenguaje C permite escribir varias proposiciones ***return*** en cualquier lugar del bloque de la función desde donde se desee regresar, el uso de esta forma sintáctica estaría VIOLANDO el principio de programación estructurada (debe haber un único punto de entrada y un único punto de retorno para una función).

Como queremos no sólo programar sino hacerlo con un muy buen estilo de programación sólo admitiremos más de un return en una función cuando sea estrictamente necesario, o sea cuando realmente se ponga en evidencia que el uso del mismo clarifica el código.

La invocación de una función es uno de los términos válidos de una expresión (ver Clase 4, pág. 6).

Una invocación de función puede aparecer en cualquier contexto donde pueda aparecer una expresión. El invocador de una función es libre de hacer lo que desee con el valor retornado.

Aclaración

Las funciones también suelen recibir el nombre de subprogramas porque son los bloques de construcción de los programas. Notar la analogía de una función con la de un programa:

- La entrada y salida de un programa permite la comunicación entre éste y el usuario del mismo.
- Los argumentos y el valor retornado por una función permiten la comunicación entre ésta y su invocador

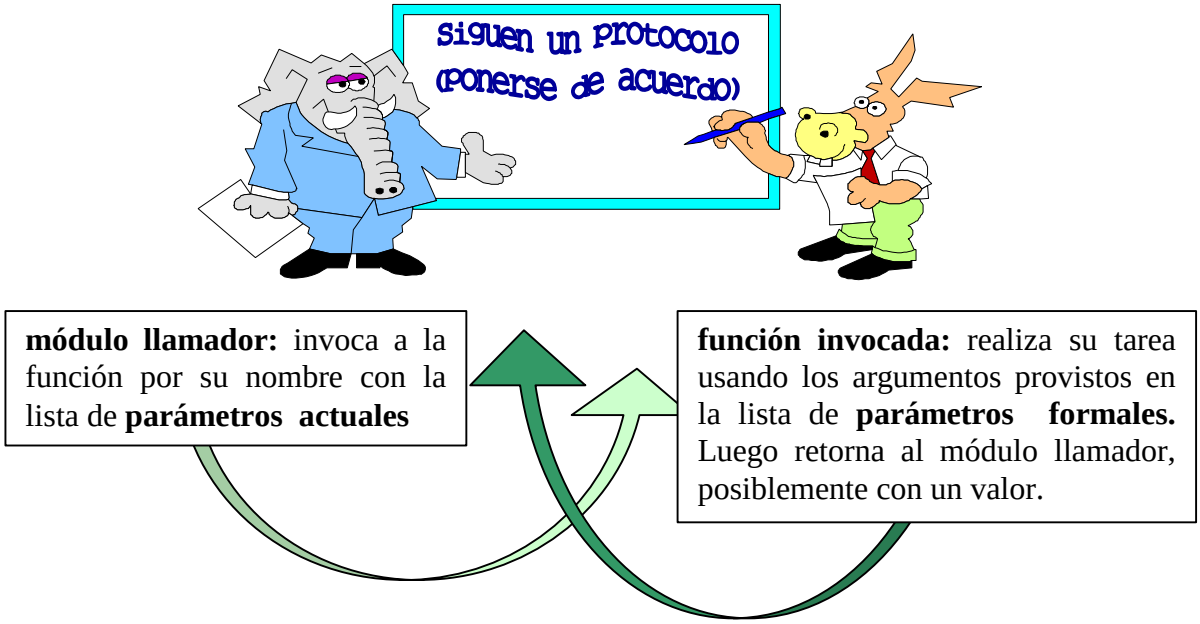
En Assembler para realizar el pasaje de parámetros había que especificar diversas acciones. Si se deseaba pasar un parámetro por **valor** (sólo de entrada), había que salvarlo en algún lugar para garantizar que su contenido NO se modificaba. Si se deseaba pasar un parámetro por **referencia** (de salida o bien de entrada/salida) se debía permitir el cambio del contenido del mismo. Los parámetros se pasaban por medio de su dirección (en el stack) o por medio de un registro en particular.

En C, la generación del código para transformar la simple línea de invocación de una función en todas las instrucciones de bajo nivel para llevar a cabo dicho pasaje está realizado por el compilador.

Los programadores sólo tienen que conocer que la ÚNICA forma de pasaje de parámetros en C es por **valor** (o sea la información es de entrada a la función). Si se quiere que la función pase información de vuelta al módulo invocador se lo debe hacer permitiendo que la función ***retorne un valor***. Obviamente NO se usarán variables globales como intercambio de información entre una función y el módulo invocador para evitar acoplamiento y efectos colaterales indeseables (Recordar que en Z80 las subrutinas no debían usar rótulos del programa principal)

Muy Importante

El pasaje de parámetros en C es siempre por valor



1.3 Declaración de Funciones o Prototipación

ANSI C exige que todas las funciones sean *declaradas* antes de ser usadas. La declaración de una función es análoga, en cierta forma, a la declaración de las variables.

La declaración de una función se denomina ***prototipación***.

Sintaxis de Declaración o Prototipación

```
tipoQueDevuelve      nombreDeLaFuncion( listaDeParametrosFormales);
```

La lista de parámetros, separados por coma, especifica el tipo y *opcionalmente* el nombre para cada uno de los parámetros. Si se incluye el nombre de los parámetros es para que el programador entienda cual es la semántica de los mismos para su invocación (obviamente los nombres deben ser significativos).

Notar que el prototipo de la función sólo le muestra al compilador ***el protocolo de la interface*** y no cómo hace la acción para la cual fue diseñada.

Ejemplo:

La funciones que venimos usando getfloat(), getint() deben prototiparse antes de ser usadas en nuestro código. Una opción sería colocar sus prototipos al comienzo del programa fuente que las utiliza:

```
float getfloat(char mensaje[], ... );
int getint(char mensaje[], ... )


int
main(void)
{
      float sueldo;
      int cantEmpleados;

      ....
      sueldo= getfloat("\nSueldo: ");
      cantEmpleados= getint("\nEmpleados: ");
      .....
      return 0;
}
```

Como se notará el hecho de tener que incluir en cada programa fuente el prototipo de todas las funciones que usemos es bastante tedioso, y se corre el riesgo de omitir alguno.

Además, si la implementación de las funciones es realizada por terceros (como en el caso de las funciones provistas por la cátedra), y resulta que en algún momento el implementador decide cambiar algún tipo en los parámetros y/o el tipo retornado debe recordar darle un aviso ***a todos los usuarios*** de las mismas para que editen ***todos los módulos fuentes*** que usen dichas funciones y cambien sus prototipos.

Una forma mucho más prolija de trabajar con funciones consiste en:

❖ escribir los prototipos de las funciones afines en un archivo llamado de encabezamiento (header) y que tiene la extensión .h

❖ escribir las implementaciones de las funciones afines en un archivo con extensión .c (pudiéndose inclusive más adelante armar bibliotecas)

❖ distribuir por un lado los ***archivos de encabezamiento*** y por otro lado ***los archivos .c o bien los .obj o bien las bibliotecas,*** dependiendo de si se quiere o no que los que utilicen las funciones conozcan o no las implementaciones.

De esta forma, los programadores usuarios de nuestras funciones, en vez de tener que escribir uno por uno los prototipos de las funciones que utilizan, harán uso de

la directiva al preprocesador **#include** con el nombre del archivo de encabezamiento (el que contiene los prototipos) al comienzo del lote fuente. Recordar que esta directiva se encarga de reemplazar la directiva por el contenido del archivo solicitado, por lo tanto el efecto será obtener, a la salida del preprocesador, los prototipos de las funciones al comienzo del lote fuente. El compilador tomaría dicha salida, sin distinguir si los prototipos de las funciones fueron escritos uno por uno por el programador, o aparecieron gracias a la directiva #include

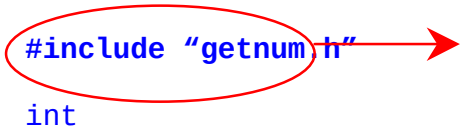
Ejemplo

El archivo de encabezamiento **getnum.h** contiene sólo los prototipos de las funciones y no la implementación de las mismas:

```
/* archivo de encabezamiento getnum.h */

float getfloat(char mensaje[], ... );
int getint(char mensaje[], ... );
```

En el código fuente hacemos un ***include*** del encabezamiento:

```
#include "getnum.h"
int
main(void)
{
      float sueldo;
      int cantEmpleados;

      ....
      sueldo= getfloat("\nSueldo: ");
      cantEmpleados= getint("\nEmpleados: ");
      .....
      return 0;
}
```



1.3.1 ¿Por qué Prototipar Funciones?

Cuando el compilador no encuentra el tipo para los parámetros y/o el tipo de retorno de una función asume que el tipo es ***int***.

El compilador, cuando encuentra una invocación de función, genera el código assembler necesario para pasar los parámetros hacia dicha función y, dependiendo del tamaño del tipo de datos, deja en el stack la información que debe tomar la función cuando se le transfiera el control en tiempo de ejecución.

A su vez cuando el compilador encuentra la definición de la función, por el tipo de los datos (que es obligatorio colocar en la lista de parámetros y en el tipo a retornar) levanta del stack la cantidad de bytes necesaria según el tamaño de los mismos. Obviamente, si los bytes dejados en el stack no se corresponden con los que después levanta la función, se produce un grave error debido a que la función levanta otra información que la que le mandaron.

Para evitar estos errores es que ANSI C exige que todas las funciones sean prototipadas, y agrega la palabra reservada **void** para poder especificar cuando la lista de parámetros es vacía, y cuando no se desea retornar valor alguno al módulo invocador.

Importante

ANSI C exige prototipar todas las funciones que se van a usar

Ejemplo

Supongamos que estamos en una **arquitectura de 32 bits**, y decidimos escribir una función que calcule el cuadrado de un número dado. Sea el archivo matem.c que contiene su código:

```
/* Archivo matem.c  */

double cuadrado( double nro )
{
      return nro * nro;
}
```

Sea ahora el programa que usa dicha función, en el cual hemos **omitido la prototipación**, y hemos olvidado la declaración de la función cuadrado antes de su invocación:

```
/* Archivo program.c  */


int
main(void)
{
      int valor = 5;
      printf("El cuadrado de %d es %g\n",
                        valor, cuadrado( valor ));
      return 0;
}
```

Al compilar y linkeditar dichos módulos, se obtiene un archivo ejecutable:

```
$ gcc  program.c  matem.c  -oproof
```

```
$ El cuadrado de 5 es  6.64587e-316
```

Como se puede observar, TODO MAL !!!

¿A qué se debe el mal funcionamiento anterior?

En el momento de la compilación del módulo ***program.c***, como la función ***cuadrado*** no está prototipada (no se conoce su tipo ni el de sus parámetros), se asumen tipos enteros, por lo cual la invocación que se realiza de ella envía al stack el contenido 5 de la variable ***valor*** en formato entero (32 bits).

Cuando se compila el módulo ***matem.c*** la función ***cuadrado*** está declarada de tipo *double* y con un parámetro tipo *double*, por lo cual, en la traducción de su código, el acceso para levantar el parámetro involucra 64 bits.

Evidentemente, durante la ejecución, no hay concordancia entre el tipo de información que coloca (push) en el stack quien invoca a la función ***cuadrado*** y el tipo de información que levanta (pop) del stack la misma para realizar su acción (al levantar 64 bits se lleva los 32 bits del entero 5 más otros 32 bits de *"datos basura"*).

1.4 Mecanismo en el Proceso de Invocación

Cuando el **compilador** encuentra la invocación de una función genera las instrucciones en Assembler necesarias para que en **tiempo de ejecución** ocurran los siguientes pasos:

- Se evalúe cada argumento actual proporcionado en la lista de parámetros. Los argumentos son expresiones que a su vez pueden consistir en invocaciones a otras funciones y pueden contener operadores para aplicar. Recordar que NO se garantiza ningún orden en que estos parámetros serán evaluados.

- Cada valor se pasa en el stack para que su correspondiente parámetro formal lo levante de allí. Antes de poner en el stack, si es preciso se realiza alguna promoción o democión al tipo especificado en el prototipo, por ejemplo si el prototipo de la función anunció que el tipo de parámetro debía ser double, y el valor que pasamos es un entero, el compilador promueve el entero a double y luego lo deja en el stack. Pero si hubiéramos omitido la prototipación el compilador no hubiera adivinado que deseábamos realizar esta conversión y hubiera generado código incorrecto. Cuando omitimos la prototipación el compilador realiza conversión, pero siempre a int (que es el tipo por omisión).

- Cuando se le transfiere el flujo de control a la función, ésta levanta del stack los valores que se le pasaron en el mismo, almacenándolos en los correspondientes parámetros formales. O sea con la invocación de una función se crea un conjunto de variables (parámetros formales) nuevo cuyo contenido es una copia de los valores de los parámetros actuales, por eso es que en C el pasaje de parámetros es sólo **por valor** (por ser una copia cualquier modificación del valor de un parámetro formal NO afecta el valor que posee el parámetro actual correspondiente.

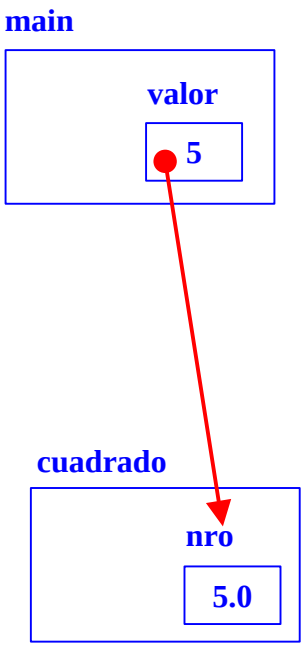
- El flujo de control de la función retorna al módulo invocador recién cuando alcanza la proposición return.

Para poder realizar **seguimientos** sobre funciones vamos a asumir la siguiente **representación**:

- Cada función se representa gráficamente por un recuadro rotulado con el nombre de la misma.

- Cada variable local y parámetro formal de la misma se la representa con otro recuadro, rotulado por su nombre, ubicado dentro de la caja que representa la función que la define. Dentro de cada recuadro de variable y parámetro se indica el valor que posee el mismo a medida que transcurre la ejecución.

Ejemplo:



Los parámetros actuales y los parámetros formales están en correspondencia en tipo y cantidad.

El pasaje de parámetros es sólo por valor en C, por lo tanto cualquier cambio realizado por la función ***cuadrado*** sobre la variable ***nro*** que ella definió, no afecta el valor de la variable ***valor*** que se encuentra definido en la función ***main.***

El compilador no confunde las variables ni aún cuando se llamaran igual. El alcance de las mismas es sólo el lugar donde fueron definidas.

2. Reglas para Tener en Cuenta

Reglas sobre Funciones

- Los nombres de las funciones (como también el de las variables y constantes) **deben ser significativos** para que leyendo sólo el nombre se pueda saber qué es lo que hacen (no cómo lo hacen). Los nombres deben surgir naturalmente por la funcionalidad del mismo, caso contrario o bien es porque no llega a hacer nada, o bien porque hace demasiadas cosas (cada función debe realizar una **única tarea)**.

- Cada prototipo de función debe ser precedida por un prólogo (comentario del bloque) que describa en forma resumida qué hace la función y cómo usarla. Una función debe ser pensada para que pueda ser utilizada por distintos programas y usuarios, por lo tanto debe quedar perfectamente documentado qué es lo que hace, qué devuelve y que parámetros espera recibir. Esto permitirá que sea utilizada correctamente y que sea fácil su modificación en el futuro

- El valor de retorno de la función debe estar separado en una línea, antes del nombre de la función, que debe encontrarse en la línea siguiente.

- No omitir la definición del tipo de la función o de sus argumentos, por más que sea ***int***. Cuando no devuelve dato o no tiene parámetros usar ***void*** explícitamente**.**

- El cuerpo de la función debe estar tabulado respecto de sus llaves, las cuales deben colocarse en la primera columna.

- Debe haber una separación de por lo menos una línea en blanco entre las declaraciones de variables locales y sentencias del bloque.

Reglas sobre Archivos de Encabezamiento (Headers Files)

- Nunca usar path absoluto en los archivos de encabezamiento
  La opción "include path", generalmente –I, es la mejor forma de poder indicar la ubicación de los archivos headers.

  Ejemplo:
  No usar **#include"c:\jobs\borlandc\include\getnum.h"**
  sino solamente **#include "getnum.h"**

- Cualquier encabezamiento que declare funciones o variables externas debe ser incluido en el propio archivo fuente que las define. De esta forma el compilador puede hacer el chequeo de tipos correcto.

- **Definir variables** en un archivo de encabezamiento es muy mala idea y es un síntoma de mala modularización.

Reglas de Ingeniería de Software para los módulos (funciones)

- No pretender crear siempre algoritmos inéditos, es muy útil prestar atención a algoritmos realizados por expertos programadores C para captar el buen estilo inherente al lenguaje

- No quedarse con el primer algoritmo que se nos ocurra. Pensar varias alternativas, evaluarlas y quedarse con la mejor, en términos de claridad y eficiencia (en ese orden).

- Utilizar **programación defensiva**, o sea nunca presuponer que algo jamás va a ocurrir (por Ley de Murphy sabemos que siempre ocurrirá)

- **Aplicar siempre a cada función los tests de software** explicados: caja blanca y caja negra (obviamente si la función tuviera muchas líneas de código o salto incondicionales innecesarios el test de caja blanca sería prácticamente imposible)

Aplicación de Funciones

Introducción

En este documento se ponen en práctica las reglas básicas para la prototipación y definición de funciones, mostrando sus ventajas.

En un primer ejemplo simple de modularización, se muestra la ventaja de reutilizar código. A través de un segundo ejemplo mostramos que, contando con una buena documentación de los módulos y de sus interfaces, es posible desarrollar sólo parte de un sistema total, combinándolo con el resto, que puede haber sido implementado por otra persona.

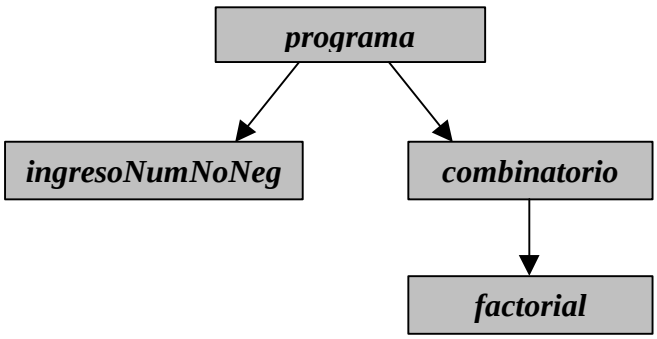
También se estipula un método de representación para el seguimiento de programas con invocaciones de funciones, que se utilizará en el resto del curso.

1. Primer Ejemplo de Modularización

Vamos a escribir un programa para calcular números combinatorios, solicitando numerador y denominador desde la entrada estándar.

$$\text{Combinatorio}(m, n) = \frac{m!}{n!\ (m-n)!} \qquad \text{con } n \geq 0 \text{ y } m \geq n$$

Representación Gráfica de la Modularización:



Documentación de la Interfaz:

Nombre	*Descripción*	*Parámetros*	
Principal	Calcula un número combinatorio para un numerador y un denominador ingresados desde la entrada estándar	No tiene	
ingresoNumNoNeg	Devuelve un número entero no negativo	No tiene	
combinatorio	Devuelve el cálculo del número combinatorio de numerador **m** y denominador **n**	**m**	Tipo: entero
			De entrada
			Valor inicial: entero mayor ó igual a **n**
		n	Tipo: entero
			De entrada
			Valor inicial: entero mayor ó igual a 0
factorial	Devuelve el factorial de un número	**nro**	Tipo: entero
			De Entrada
			Valor inicial: numero entero mayor ó igual a 0

1.1 Una Posible Versión

```
/* Archivo combin.h
** Autores: G & G
** -----------------
*/

/* funcion que devuelve un entero no negativo  */
unsigned ingresoNumNoNeg(void);

/* funcion que recibe dos enteros no negativos de entrada,
** considerados como numerador y denominador, y devuelve
** el numero combinatorio correspondiente a los mismos
*/
unsigned long combinatorio(unsigned m, unsigned n);

/* funcion que recibe un numero entero no negativo y
** devuelve su factorial
*/
unsigned long factorial(unsigned nro);
```

```
/* Archivo combin.c
** El siguiente programa calcula el numero combinatorio,
** una vez solicitados el numerador y el denominador.
*/

#include <stdio.h>
#include "getnum.h"
#include "combin.h"

int
main(void)
{
      int numerador, denominador;

      printf (“Ingrese el numerador (no negativo)” );
      numerador = ingresoNumNoNeg();

      printf (“Ingrese el denominador (no negativo)” );
      denominador = ingresoNumNoNeg();

      if (numerador  >= denominador)
            printf (“C(%u, %u)= %lu \n”, numerador, denominador,
                        combinatorio(numerador,denominador) );
      else
            printf (“El numerador debe ser mayor o igual al
                     denominador\n” );
      return 0;
}


/* La siguiente función NO es una buena versión de validación,
** ya que los datos pueden ingresarse mal y sin embargo ciclar
** quedando como correctos. A lo largo del curso se iran viendo
** versiones mejoradas, con el uso de strings
*/
unsigned
ingresoNumNoNeg(void)
{
      int  nro;

      do
      {
            nro = getint(“”);
      }
      while (nro < 0);

      return nro;
}



unsigned long
combinatorio(unsigned m, unsigned n)
{
      return (factorial(m) /  ( factorial(n) * factorial(m - n)) );
}
```

```
unsigned long
factorial(unsigned nro)
{
      unsigned long producto;

      for (producto = 1; nro > 1; nro--)
            producto *= nro;

      return (producto);
}
```

1.2 Testeo de Caja Blanca

Testeo de Caja Blanca para ***ingresoNumNoNeg***

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convalidación*
Cobertura de sentencias	cualquier **nro**	**T1={ 5 }**	5	5	✓
Cobertura de decisiones	para **do-while:** nro<= -1	**T2={ -2 }**	inválido	inválido	✓
		T3={ -1}	inválido	inválido	✓
		T4={ 0 }	0	0	✓
Cobertura de límites	-	**-**	-	-	-

O.K.

Testeo de Caja Blanca para ***factorial***

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convalidación*
Cobertura de sentencias	**nro > 1**	**T1={ 3 }**	6	6	✓
Cobertura de decisiones	límite para **for:** nro >= 2	**T2={ 1 }**	1	1	✓
		T3={ 2 }	2	2	✓
		T4 =T1	✓	✓	✓
Cobertura de límites	-	**-**	-	-	-

O.K.

Testeo de Caja Blanca para ***main***

	Rango de Valores	*Ti*	*Valor Esperado*	*Valor Obtenido*	*Convalidación*
Cobertura de sentencias	para el **if**: 5 y 2 para el **else**: 2 y 5	**T1={5,2}**	10	10	✓
		T2={2,5}	inválido	inválido	✓
Cobertura de decisiones	(num >= den) ***Verdadero***	**T3 =T1**	✓	✓	✓
	(num >= den) ***Falso***	**T4 =T2**	✓	✓	✓
Cobertura de límites	-	-	-	-	-

O.K.

1.3 Ejemplo de Seguimiento

main

numerador

denominador

ingresoNumNoNeg

nro

combinatorio

m

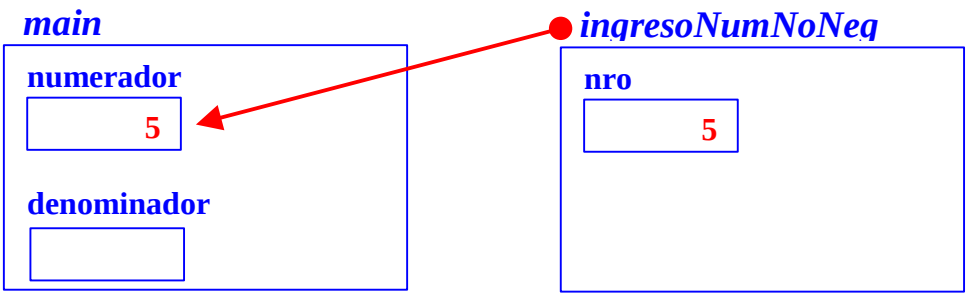
n

factorial

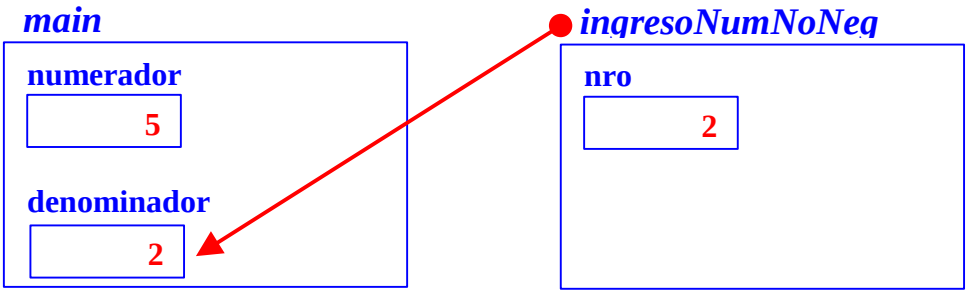
nro

producto

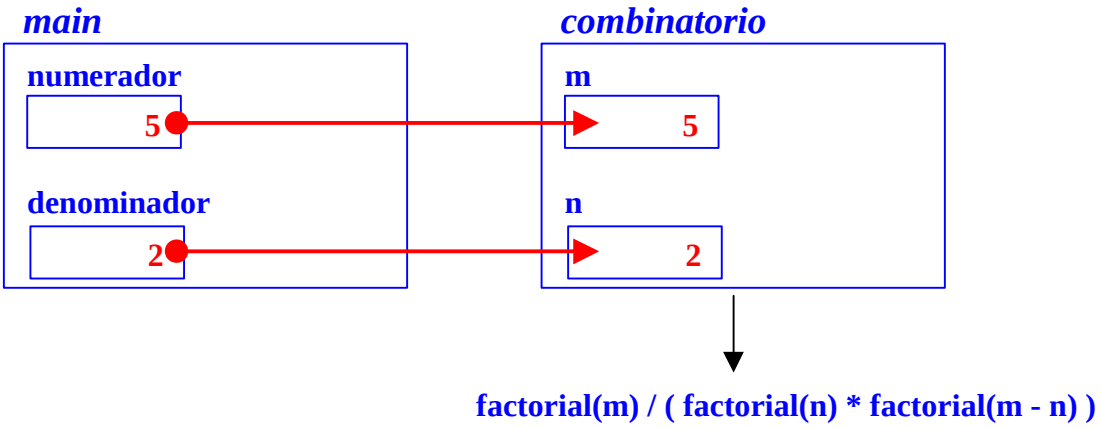
1) En la invocación **numerador = ingresoNumNoNeg**



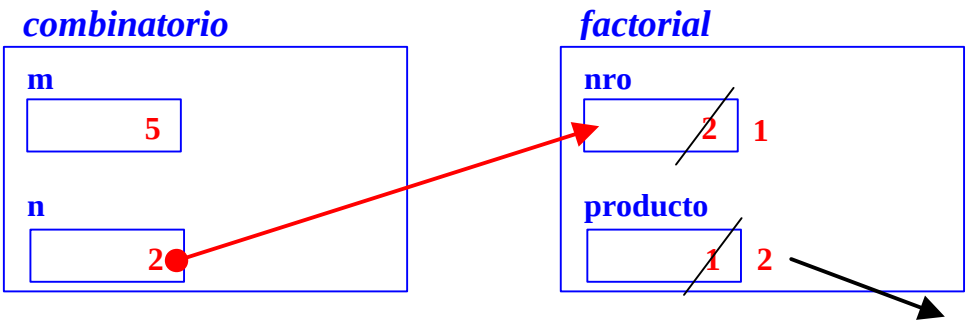
2) En la invocación **denominador = ingresoNumNoNeg**



3) Como 5 > 2, se invoca **combinatorio (numerador, denominador)**

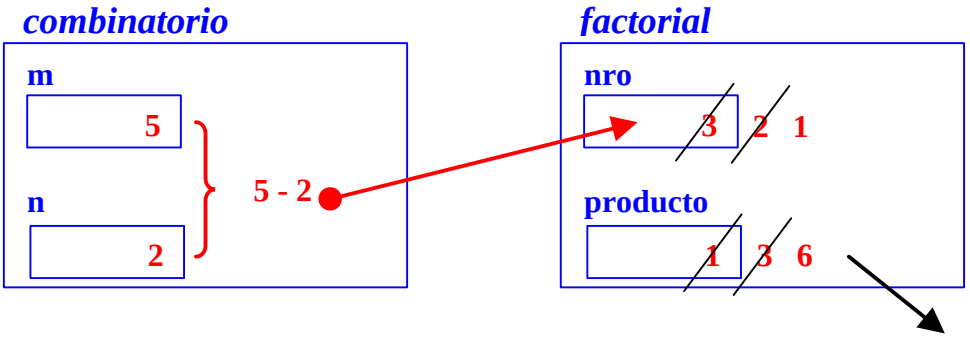


4) Cuando se invoque, en algún momento, **factorial(n)**



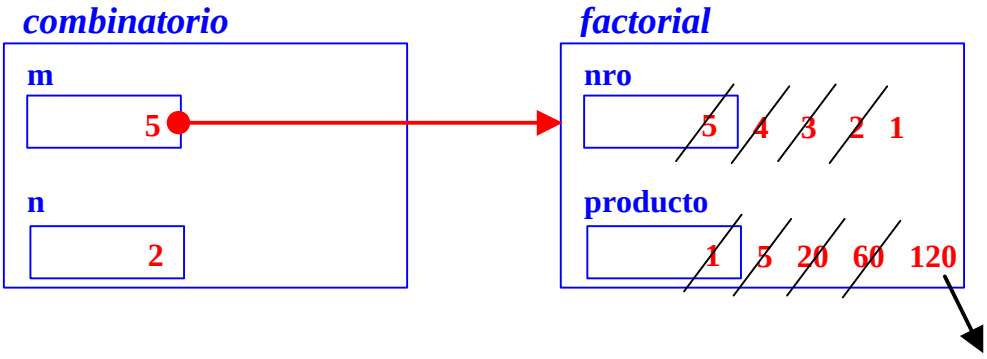
devuelve **2**

5) Cuando se invoque, en algún momento, **factorial(m - n)**



devuelve **6**

6) Cuando se invoque, en algún momento, **factorial(m)**



devuelve **120**

7) Finalmente la función **combinatorio** devuelve:

120 / (6 * 2) = 120 / 12 = 10

y el programa principal imprime:

C (5, 2) = 10

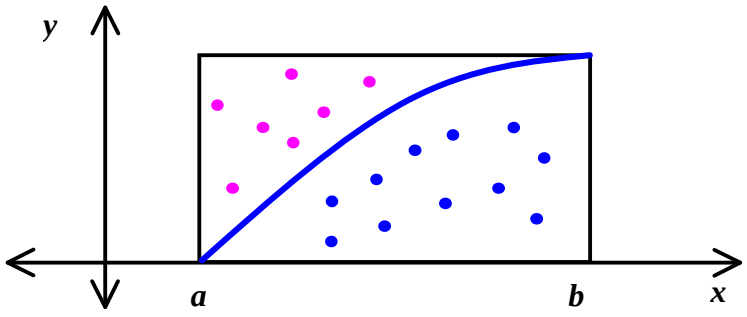
2. Segundo Ejemplo de Modularización

Se busca calcular el área de una función en un cierto intervalo [a,b], tal que en el mismo la función es positiva monótona y no tiene raíces. Para esto aplicaremos el método de Montecarlo.

Para aplicar dicho método se forma un rectángulo R cuya base está formada por el segmento de abscisa entre los extremos del intervalo, y cuya altura es igual al máximo absoluto de la función en dicho intervalo.

Luego se bombardea dicho rectángulo, generando puntos aleatorios (x,y). Cuando el valor del **y aleatorio** se encuentra entre la imagen del **x aleatorio** correspondiente y el eje de abscisas, se lo cuenta como acierto:

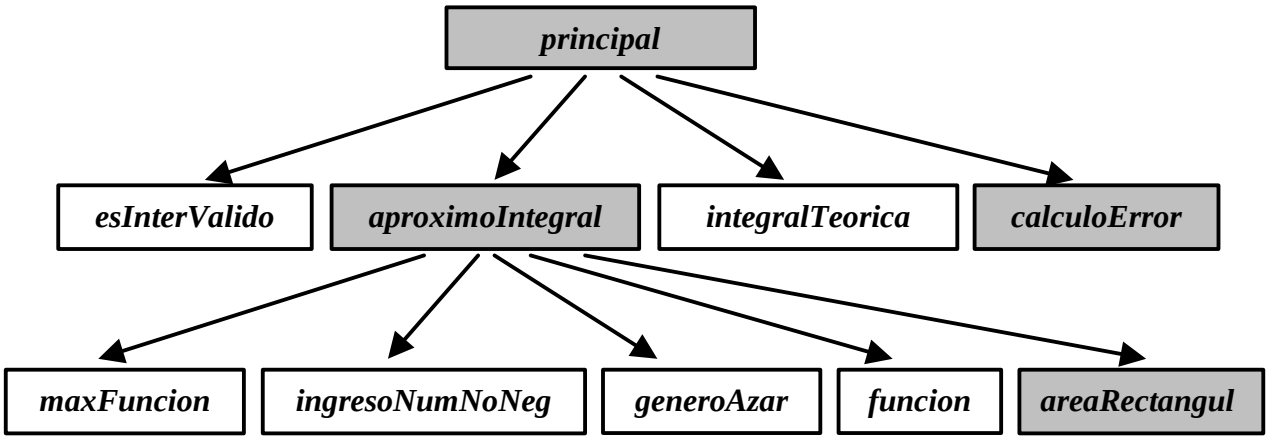
yAleatorio <= F(xAleatorio) Þ acierto



Después de N cantidad de tiros, la integral se calcula como:

$$\frac{\text{Cantidad de aciertos} \;*\; \text{Área del Rectángulo}}{N}$$

Representación Gráfica:



Documentación de la Interfaz:

Nombre	*Descripción*	*Parámetros*	
principal	Calcula la integral de una función positiva monótona en un intervalo dado, tanto en forma teórica como en forma aproximada (mediante el Método de Montecarlo), indicando el porcentaje de error cometido con la aproximación.	No tiene	
esInterValido	Devuelve 1 si los extremos pertenecen al un intervalo real válido y 0 en caso contrario	**izq**	real E v.i.: número real
		der	real E v.i.: número real
aproximoIntegral	Realiza el cálculo de la integral de la función positiva monótona, mediante el método de Montecarlo, solicitando la cantidad de tiros desde entrada estándar.	**izq**	real E v.i.: **izq** <= **der**
		der	real E v.i.: número real
integral Teorica	Devuelve el cálculo de la integral teórica de la función, en el intervalo [izq, der], mediante la Regla de Barrow	**izq**	real E v.i.: **izq** <= **der**
		der	real E v.i.: número real

Nombre	*Descripción*	*Parámetros*	
calculoError	Devuelve el error relativo porcentual entre el valor calculado y el teórico esperado	**valCal-culado**	real
			E
			v.i.: número real
		valTeo-rico	real
			E
			.i.: número real
maxFuncion	Devuelve el máximo valor de la función monótona positiva dentro del intervalo [izq, der]	**izq**	real
			E
			v.i.: **izq** <= **der**
		der	real
			E
			v.i.: número real
ingresoNumNoNeg	Devuelve un número entero no negativo	No tiene.	
generoAzar	Devuelve un número real aleatorio perteneciente al intervalo recibido	**izq**	real
			E
			v.i.: **izq** <= **der**
		der	real
			E
			v.i.: número real
funcion	Devuelve la imagen para la función matemática positiva en el punto indicado	**x**	real
			E
			v.i: número real
areaReactangulo	Devuelve el área del rectángulo cartesiano definido por las abscisas y ordenadas recibidas	**xIzq**	real
			E
			v.i.: **izq** <= **xDer**
		xDer	real
			E
			v.i.: número real
		yArriba	real
			E
			v.i.: **yArriba** <= **yAbajo**
		yAbajo	real
			E
			v.i.: número real

2.1 Una Posible Versión

```
/*
**   Archivo monte.h
**   Autores: G & G
*/



/*
**  La siguiente funcion recibe los extremos de un intervalo real y
**  determina si es valido (1) o no (0)
*/
int esInterValido(float izq, float der);


/*
**  La siguiente funcion calcula la integral de la funcion matematica
**  positiva monótona en un cierto intervalo [izq, der] mediante el
**  Metodo  de  Montecarlo, solicitando cantidad de tiros desde la
**  entrada estándar
*/
float aproximoIntegral(float izq, float der);


/*
**  La siguiente funcion calcula la integral de la funcion en forma
**  teorica, aplicando Regla de Barrow
*/
float integralTeorica(float a, float b);


/*
**  La siguiente funcion calcula el error relativo porcentual cometido
**  en un calculo practico, respecto del valor teorico esperado.
**  Respeta el signo de la diferencia para indicar si el error
**  cometido fue por exceso o por defecto
*/
float calculoError(float integCalculada, float integTeorica);


/*
**  La siguiente funcion devuelve el maximo de la funcion matematica
**  positiva monótona en un cierto intervalo [a,b]
*/
float maxFuncion(float a, float b);


/*
**  La siguiente funcion permite ingresar desde la entrada estandar un
**  numero entero no negativo
*/
int ingresoEntNoNeg(void);
```

```
/*
**  La siguiente funcion devuelve un numero aleatorio real,
**  perteneciente al intervalo [izq, der]
*/
float generoAzar(float izq, float der);


/*
**  La siguiente funcion representa una funcion matematica real de una
**  sola variable real
*/
float funcion(float x);


/*
**  La siguiente funcion calcula el area de un rectangulo cartesiano,
**  definido por las abscisas de sus lados verticales y las ordenadas
**  de sus lados horizontales
*/
float areaRectang(float xIzq, float xDer,
                                  float yArriba, float yAbajo);
```

```
/*
**    Archivo:  monte.c
**    Autores:    G & G
**
**    Este programa calcula la integral de una funcion positiva
**    monótona en un intervalo real ingresado desde la entrada
**    estandar, mediante el Metodo de Montecarlo.
**    Tambien indica el porcentaje de error cometido con dicho
**    calculo, respecto del valor teorico.
**    En este código la función a integrar es F(x)= x*x
*/


#include <stdio.h>
#include <stdlib.h>
#include “getnum.h”
#include “monte.h”

#define  VALOR_ABSOLUTO(n)  ( (n) >= 0?(n):(-(n)) )


int
main(void)
{
      float izq, der;
      float simulacion, integral;

      izq= getfloat("Ingresar el extremo izquierdo del intervalo: ");
      der= getfloat("Ingresar el extremo derecho del intervalo: ");
```

```
      if ( esInterValido(izq,der) )
      {
            simulacion = aproximoIntegral(izq, der);
            integral = integralTeorica(izq, der);

            printf("Integral en [%.5f, %.5f]\n", izq, der);
            printf("\tSimulada: %.2f\n", simulacion);
            printf("\tTeorica : %.2f\n", integral);

            printf("Error de la simulacion: %.2f%%\n",
                              calculoError(simulacion, integral) );
      }
      else
            printf("Los limites del rectangulo son incorrectos");

      return 0;
}



int
esInterValido(float izq, float der)
{
      return (der >= izq);
}


float
aproximoIntegral(float izq, float der)
{
      float x, y, tope;
      int i, acum, tiros;

      tope= maxFuncion(izq, der);

      printf("Ingrese la cantidad de tiros a efectuar: ");
      tiros = ingresoEntNoNeg();

      for (acum=0, i=tiros ; i; i--)
      {
            x= generoAzar(izq, der);
            y= generoAzar(0, tope);
            if (y <= funcion(x) )
                  acum++;
      }

      return  ( acum * areaRectang(izq, der, tope, 0) / tiros );
}


float
integralTeorica(float a, float b)
{
      return ( (b * b * b / 3) - (a * a * a / 3) );
}
```

```
float
calculoError(float integCalcu, float integTeo)
{
      return ( VALOR_ABSOLUTO(integCalcu-integTeo) / integTeo * 100 );
}


float
maxFuncion(float a, float b)
{
      return (funcion(a) > funcion(b))?funcion(a):funcion(b);
}


int
ingresoEntNoNeg(void)
{
      int n;
      do
            n= getint("");
      while (n <0);

      return n;
}


float
generoAzar(float izq, float der)
{
      float n;
      n= izq + (der - izq) * rand() / RAND_MAX;

      return n;
}


float
funcion(float x)
{
      return (x * x);
}


float
areaRectang(float xIzq, float xDer, float yArriba, float yAbajo)
{
      return ( (xDer - xIzq) * (yArriba - yAbajo));
}
```

Gestión de la Memoria

Introducción

En este documento se describe la relación entre el formato de un objeto binario y la zona de memoria que se le asignará cuando se solicite la ejecución del mismo. Asimismo se verá la manera que tiene el programador para gestionar la memoria en lenguaje C.

1. Run Time Environment

Los archivos objeto y los ejecutables poseen distintos formatos que dependen de la arquitectura del computador en la cual fueron generados, y obviamente a los últimos se los podrá ejecutar en dicho entorno. Esta diversidad de formatos es la que hace que un programa ejecutable no sea portable de una arquitectura a otra (obviamente si se programa en ANSI C la portabilidad se garantiza para los programas fuentes, ya que sólo faltaría recompilarlos en las distintas arquitecturas).

Los distintos formatos de archivos binarios manejan el concepto de secciones o segmentos (no confundir con la palabra segmento que se usa en Sistemas Operativos): áreas en un archivo binario que corresponden a cierta información relacionada.

Cuando en C el **compilador** y el **linkeditor** producen un archivo binario, justamente generan tres secciones: ***Text***, ***Data*** y ***BSS***. Cada una de ellas con una función específica, que veremos al final de esta sección.

En Unix para obtener la información de los tres segmentos que integran un modulo binario se debe ejecutar:

$ size nombreDelArchivoBinario

Ejemplo 1:

Para obtener la información de los segmentos en un archivo objeto llamado pepe.o se debe realizar

$ size pepe.o

obteniendo

text	data	bss	dec	hex	filename
31	0	0	31	1f	pepe.o

Ejemplo 2:

Para obtener la información de los segmentos en un archivo ejecutable llamado pepe.out se debe realizar

$ size pepe.out

obteniendo

text	data	bss	dec	hex	filename
1037	196	4	1237	4d5	pepe.out

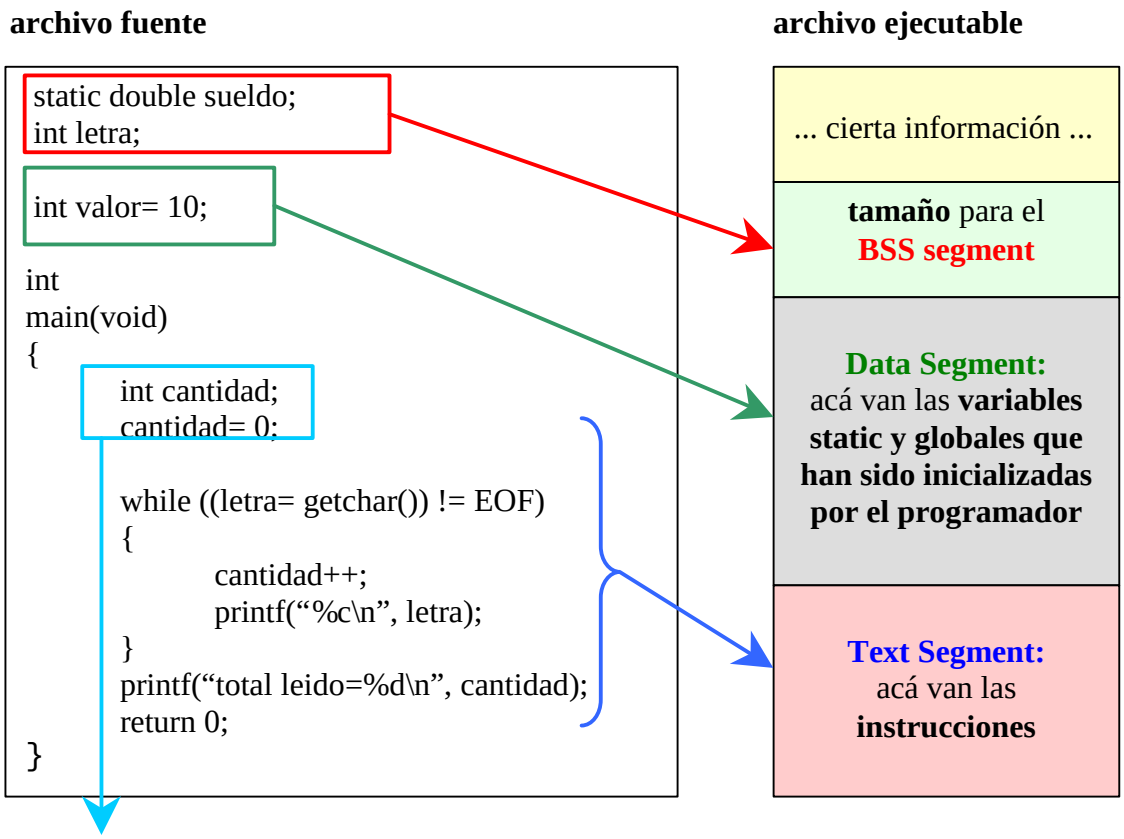
Ejemplo 3:

Si se quiere obtener información de los segmentos en un archivo fuente llamado pepe.c se obtendrá un error porque no es binario

$ size pepe.c

se obtiene: **File format not recognized**

A continuación se muestra el formato de un archivo binario genérico (depende de cada sistema operativo) que surge de un cierto programa fuente:



Las variables locales no estáticas no van al archivo binario porque son creadas en tiempo de ejecución

Así es como un archivo binario es organizado en segmentos en forma tal que cuando luego se pida su ejecución, el cargador del sistema operativo pueda colocarlo en memoria directamente.

El cargador tomará el archivo binario y lo colocará en memoria listo para que pueda ejecutar cuando se le dé el control.

El propósito para el cual sirve cada zona mostrada en la próxima figura y los pasos que el cargador realizará típicamente cuando tenga que cargar el programa en memoria son:

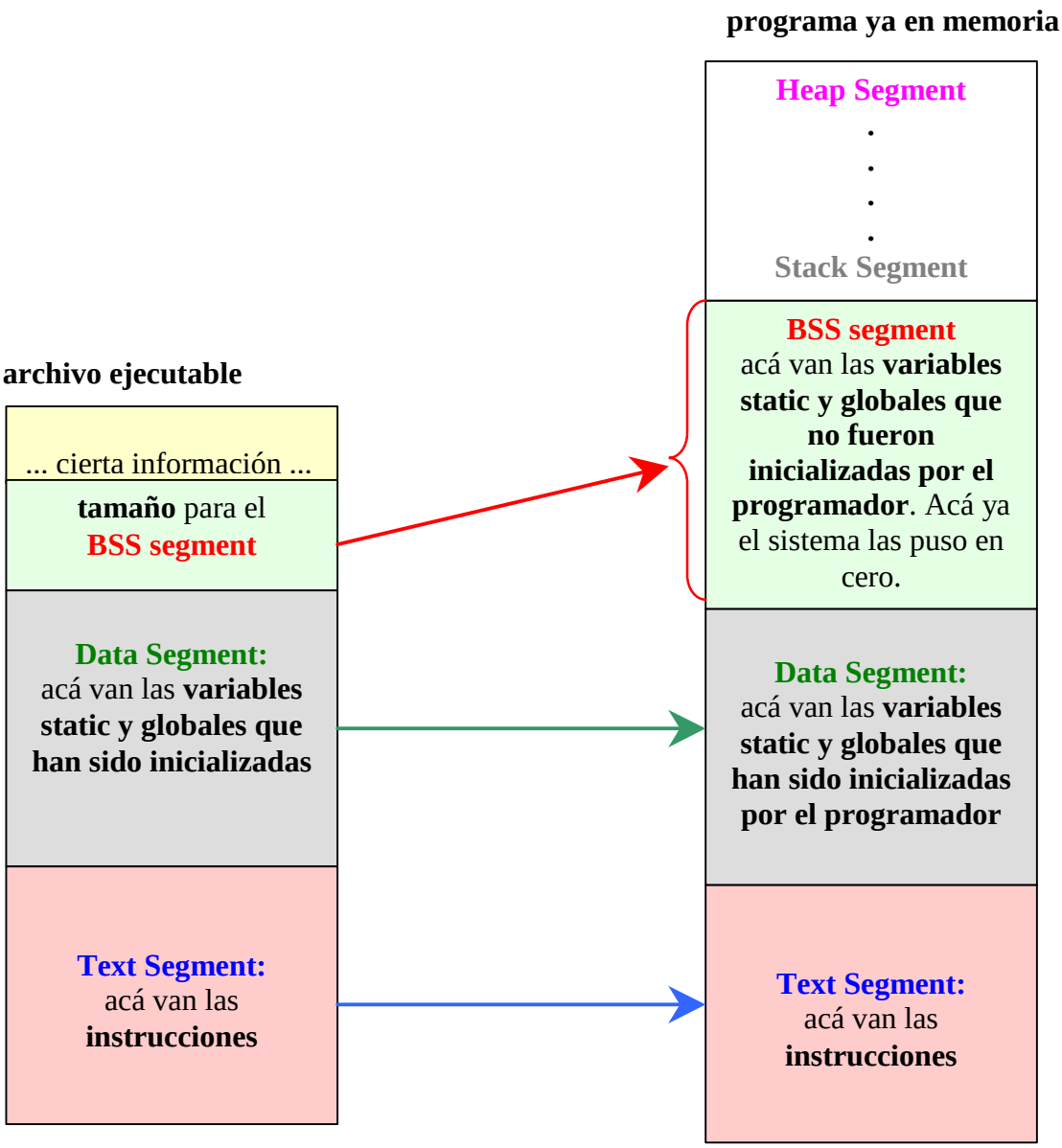
❖ el **Text Segment**: es el área reservada para las instrucciones del programa. El cargador directamente copia esta zona en la zona homónima de la memoria (como una imagen). Su tamaño es **fijo** y **se conoce antes de la ejecución**.

❖ el **Data Segment**: es el área para almacenar las variables globales o estáticas inicializadas por el programador. El cargador directamente copia esta zona en la zona homónima de la memoria (como una imagen). Su tamaño es **fijo** y **se conoce antes de la ejecución**. Así es como estas variables comienzan con el valor solicitado al iniciar la ejecución, y existen durante toda la ejecución del mismo.

❖ el **BSS Segment (Block Started by Symbol)**: es el área para almacenar las variables globales y estáticas no inicializadas por el programador. El cargador típicamente lee el tamaño del BSS Segment y aloca lugar en memoria para el mismo, además se encarga de **inicializarlo en cero**. Su tamaño es **fijo** y **se conoce antes de la ejecución**. Así es como estas variables también comienzan inicializadas por el sistema en cero al iniciar su ejecución y existe durante todas la ejecución del mismo.

❖ el **Stack Segment**: es el área donde se almacenan los stack frames, que recordemos, es el mecanismo que usan los compiladores para manejar la invocación de funciones (guardando dirección de retorno, parámetros y variables automáticas), y las variables temporarias (que surgen temporalmente al evaluar una sub-expresión). Su tamaño **crece y decrece dinámicamente durante la ejecución** (cada vez que se invoca una función o se evalúa una expresión). El cargador alocará cierto lugar en memoria para este segmento.

❖ el **Heap Segment**: es el área donde se almacenan las variables que son explícitamente creadas por funciones especiales que invoca el programador para tal fin. **Su crecimiento y decrecimiento es dinámico (no se conoce antes de comenzar la ejecución)**. La veremos en Estructuras de Datos y Algoritmos.



Aclaración:

Las zonas en memoria no necesariamente van en este orden ya que esto depende de la arquitectura de la computadora.

Según el lugar donde se declare una variable dentro de un programa fuente y el calificador opcional que se le agregue se le está indicando al compilador donde reservar lugar para la misma en el ***run time environment***.

2. Clase de Almacenamiento de las Variables

Toda variable en el lenguaje C tiene un tipo, un nombre y posee una clase de almacenamiento que determina su:

- **duración, persistencia o tiempo de vida**
- **alcance**
- **visibilidad o enlace**

2.1 Duración, Persistencia o Tiempo de vida

Definición

Duración es el período durante el cual dicho identificador existe en memoria.

Existen dos tipos de duración:

- ***Duración Automática***

Las variables de duración automática son creadas y destruidas a medida que se las necesita. Corresponde a las variables que **son declaradas dentro de un bloque y no son precedidas ni por las palabra extern ni static**. Son creadas al introducirse en el ámbito del bloque donde fueron declaradas, existen mientras dicho bloque esta activo y se destruyen al salir del bloque.

Las palabras reservadas utilizadas para declarar una variable de persistencia automática son **auto** y **register**, pero como por omisión las variables declaradas dentro del bloque tienen persistencia automática, la palabra reservada **auto** no suele usarse explícitamente.

Por último, el calificador **register** sirve para sugerirle al compilador que la variable se almacene en un registro en vez de memoria principal, con el fin de lograr una mejor performance. Sin embargo es solo una sugerencia, y si no hay suficiente cantidad de registros el compilador puede ignorar este pedido. Por otra parte los optimizadores suelen de por sí utilizar los registros para mejorar la ejecución en forma automática. **Se aconseja no usarlo** porque puede ser contraproducente: el programador puede querer hacer más rápido el programa pidiendo colocar explícitamente la variable en un registro y algún compilador podría hacer swapings de registros a memoria si no tiene registro suficientes, retardando la ejecución del mismo. No hay reglas para los compiladores al respecto.

El calificador **register** suele usarse al implementar Sistemas Operativos o Bases de Datos (porque justamente no son portables y están diseñados para cierta arquitectura de computador).

➢ ***Duración Estática***

Las variables de duración estática son creadas desde el comienzo de la ejecución del programa y recién son destruidas al terminar de ejecutarse.

Corresponde a las variables declaradas fuera de un bloque o bien a aquellas que dentro de un bloque estén precedidas por las palabras reservadas **extern y static.**

El cargador asegura que las variables **static** se inicializan antes de comenzar la ejecución del programa: ya sea por el valor solicitado explícitamente por el programador o con el valor cero en caso de omisión.

Muy Importante:

Las variables de persistencia automática **NO son inicializadas automáticamente** por el compilador, por lo tanto **NUNCA ASUMIR** que contiene algún valor inicial por omisión. Se debe tener especial cuidado con estas variables cuando funcionan como acumuladores, etc.

Aclaración

Obviamente tiene que ver con dónde se almacenan las variables dentro de las zonas de memoria antes vista.

Aquellas variables que están en el **Data Segment** o **BSS Segment** son creadas antes de comenzar la ejecución del proceso (cuando el cargador coloca el proceso en memoria a partir del archivo ejecutable) y van a existir durante toda la ejecución del proceso. Como estas variables mantienen su lugar en memoria durante toda la ejecución cada vez que se las utiliza mantienen el último valor que se les fue asignado.

Por el contrario, aquellas que se almacenan en el **Stack Segment** y **Heap Segment** son creadas dinámicamente en tiempo de ejecución, y existen sólo desde el momento en que se las crea hasta el momento que se les desasigna lugar. Una vez que son destruidas habría que volver a crearlas para poder volver a usarlas, pero como su nueva creación no necesariamente va al mismo lugar que se les asignó la vez anterior, nunca conservan su valor anterior.

Importante:

Cuando se declara una variable dentro de un bloque, la misma tiene prioridad sobre otras declaraciones de variables locales con el mismo nombre que aparezcan en otros bloques que lo contengan a él, y sobre la declaración de variables globales con el mismo nombre.

Esto permite inclusive que un mismo nombre de variable local tenga distinto tipo que otra global o local declarada en un bloque contenedor.

Ejemplo:

Dado el siguiente programa, si desde la entrada estándar se ingresara **No¿**

```
#include <stdio.h>

int
main(void)
{
      float letra= 43;
      while (getchar() != EOF )
      {
            int letra= 1;
            printf("bloque mas anidado %d\n", letra);
      }
      printf("bloque mas externo %f\n", letra);
}
```

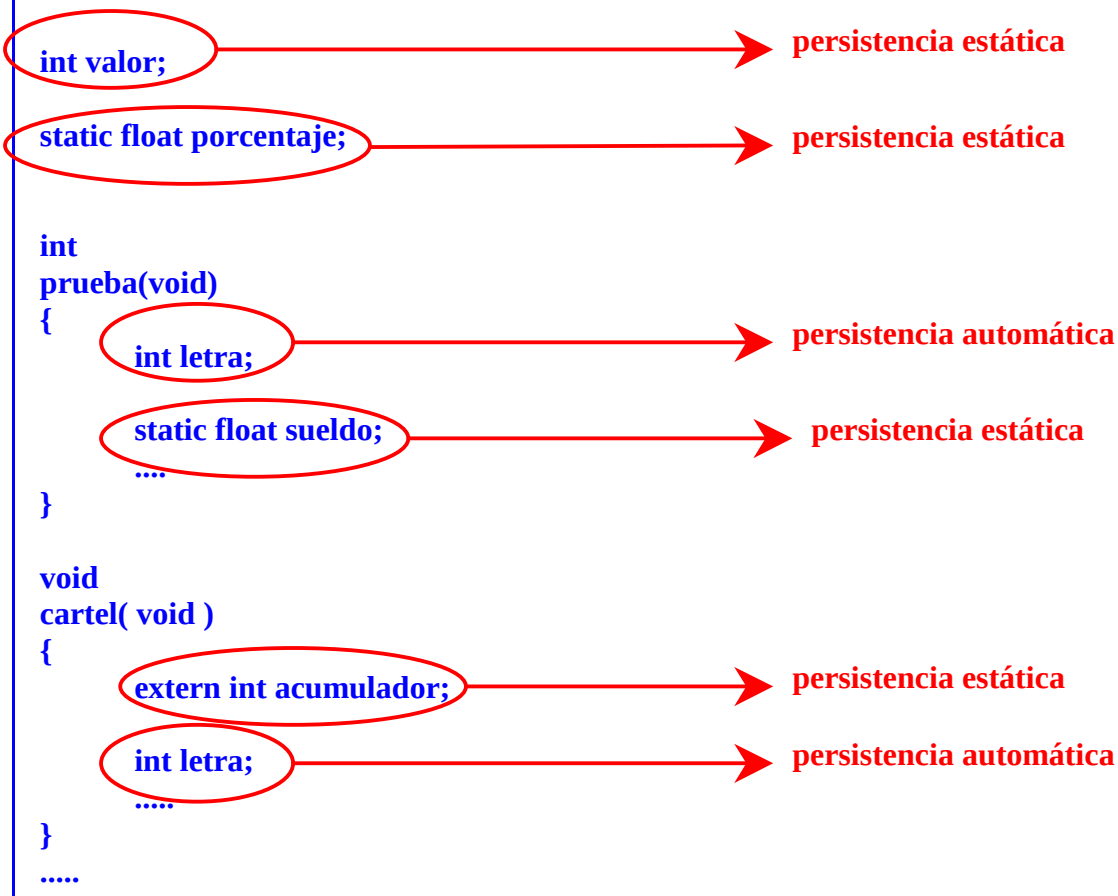
dentro de este bloque que comienza con el *while*, esta nueva declaración tiene prioridad sobre la otra (aunque la anterior tiene alcance sobre ésta)

se obtendría

bloque mas anidado 1
bloque mas anidado 1
bloque mas anidado 1
bloque mas externo 43.00000

Ejercicio:

Indicar para cada una de las siguientes variables cuales tienen persistencia automática y cuales estática.



2.2 Alcance

Definición

El alcance de un identificador en un programa está dado por el lugar donde puede ser usado.

Existen identificadores que sólo pueden ser referenciados en ciertos lugares de un programa.

Existen dos tipos de alcance:

- ***Alcance Externo o Global***

Corresponde a aquellas variables declaradas fuera de las funciones y tienen su alcance desde su declaración hasta el final del lote fuente.

- ***Alcance Interno o Local***

Corresponde a aquellas variables declaradas dentro de un bloque y tienen su alcance restringido a dicho bloque.

Ejemplo:

Para el ejemplo anterior indicar cual es el alcance de las variables.

```
int valor;                          alcance global
static float porcentaje;            alcance global

int
prueba(void)
{
        int letra;                  alcance local
        static float sueldo;        alcance local
        ....
}

void
cartel( void )
{
        extern int acumulador;      alcance local
        int letra;                  alcance local
        .....
}
.....
```

El alcance de una variable es diferente de su duración. Esto se evidencia claramente en el ejemplo anterior:

❖ La variable **sueldo** tiene **persistencia estática** pero **su alcance es local**, ésto significa si la función prueba fuera invocado dos veces seguidas, la segunda vez el contenido de la variable sueldo sería el último que le fue asignado en la invocación anterior (no se destruye su almacenamiento al finalizar de ejecutar el bloque que la contiene), sin embargo su alcance es local ya que cualquier intento de referencia de la misma fuera de dicho bloque (por ejemplo: en la función cartel) resultaría inválido (no compilaría).

❖ La variable **porcentaje** también tiene **persistencia estática** pero **su alcance es global**, esto significa que si bien el valor asignado continúa en los sucesivos usos su alcance es global porque puede ser usada por cualquier función desde el lugar donde aparece su declaración hasta el final del lote fuente. Así es como podrían referenciarla las funciones prueba y cartel.

2.3 Visiblidad o Enlace

Definición

La visibilidad de un identificador en un programa determina si el mismo puede o no ser reconocido por **otros archivos fuentes.**

Existen tres tipos de visibilidad:

➢ ***Visibilidad Externa***

Dado un archivo que integra un archivo ejecutable, una variable declarada en él tiene visibilidad externa si puede ser usada en otro de los archivos que integran dicho ejecutable. O sea esa variable es visible para otros módulos.

La palabra reservada usada para lograr esto es **extern**, pero en el caso de que la variable declarada sea de alcance global puede omitirse. Cuando se tiene una variable de visibilidad externa implica que la misma puede ser declarada por otros módulos, pero en realidad todos se están refiriendo a la misma variable (como se verá en la próxima sección uno de dichos módulo debería en realidad definirla).

Si la variable de visibilidad externa es de alcance local NO puede ser inicializada (no compilaría), caso contrario sí puede hacerlo.

➢ *Visibilidad Interna*

Dado un archivo que integra un archivo ejecutable, una variable declarada en él tiene visibilidad interna si la misma sólo puede ser vista dentro de dicho archivo fuente, y ningún otro archivo que integra el ejecutable la puede ver. O sea cualquier declaración de una variable con el mismo nombre en otro de los archivos que integran el ejecutable se refiere en realidad a otro objeto. Es decir, la variable es visible sólo en ese módulo.

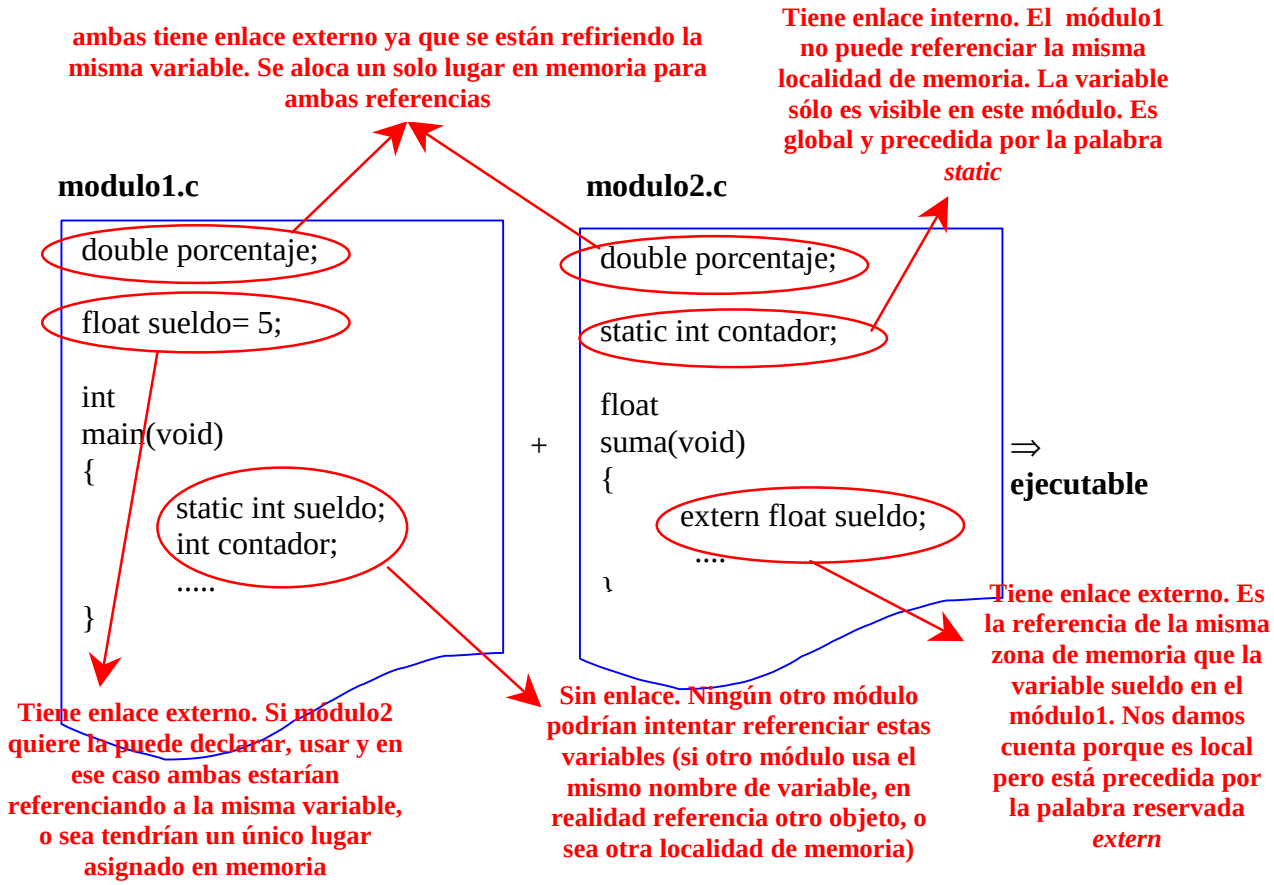
La única forma de lograr esto es calificando con la palabra reservada ***static*** a una variable de alcance global.

➢ *Sin Enlace*

Es la visibilidad que poseen todas las variables locales que **no** son precedidas por la palabra reservada ***extern***. No hace falta utilizar ninguna palabra reservada para lograr esto. Cuando una variable no tiene enlace cada declaración que aparece se refiere a un nuevo objeto. En este caso, como veremos en la próxima sección, cada declaración es en realidad una definición.

Ejemplo:

Se tienen los siguientes fuentes que formarán un único archivo ejecutable



3. Declaración vs Definición de Variables

Existe cierta analogía entre la declaración y definición de funciones y variables.

El ***compilador*** exige que las funciones sean ***declaradas*** o prototipadas antes de ser usadas (se debe brindar su tipo de retorno, su nombre y el tipo de sus parámetros). El ***linkeditor*** exige que cada función utilizada sea ***definida*** exactamente una vez (debe codificarse su cuerpo) caso contrario no finaliza exitosamente a causa de una referencia no resuelta o por repetición de código.

Análogamente el ***compilador*** exige que las variables sean ***declaradas*** antes de ser usadas (debe anunciarse su tipo y nombre). El ***linkeditor*** exige que cada variable utilizada sea ***definida*** exactamente una vez, caso contrario no finaliza exitosamente a causa de una referencia no resuelta o por repetición de definición.

Importante

La definición de una variable sirve para reservar espacio en memoria para la misma.

Según dónde se la defina y que calificador la modifique se reservará espacio en los distintos segmentos antes vistos.

Aclaración

Así como la definición de una función puede servir como prototipación, la definición de una variable sirve como declaración de la misma.

La definición de una variable sirve como declaración, el recíproco no es cierto.

Todas las declaraciones que habíamos realizado en los códigos de las clases anteriores, a través de variables locales eran en realidad definiciones.

Cuando se arma un único programa ejecutable a partir de varios programas fuentes se pone en evidencia si a una variable se la está definiendo o sólo se la está declarando. En ese caso (cuando el sistema es muy grande) puede hacerse uso de alguna variable global que sea compartida por dichos módulos. Solamente uno de ellos la debe definir y el resto la debe declarar.

Viendo un código existen ciertas heurísticas que ayudan a darse cuenta si una variable está siendo solo declarada o definida:

- **si la variable está inicializada**, sin lugar a dudas está **siendo definida**. No está permitido inicializar una variable varias veces porque justamente no está permitido definir una variable más de una vez.

- **si la variable no está inicializada y es *extern* explícitamente** entonces está sólo **siendo declarada**. Así es como si se tiene una variable en distintos módulos como extern y no inicializada, el linkeditor arrojará un error de no definición de la misma.

- **en cualquier otro caso** para saber si la variable está declarada o definida hay que inspeccionar que pasa en el resto del dicho módulo u otros módulos.

Ejemplo 1:

```
extern int valor= 5;          /* Rta: la variable valor está siendo definida */

float sueldo= 500.30;         /* Rta: la variable sueldo está siendo definida */

extern edad;                  /* Rta: la variable edad está siendo declarada */

double precio;                /* Rta: No se sabe. No basta esta zona de código
                                 para darse cuenta. La información es insuficiente */
```

Ejemplo 2:

Para resaltar la ambigüedad del ultimo caso, a continuación se muestran los dos archivos fuentes que formarían luego de la linkedicion un único código ejecutable. En la primera opción se evidencia que en el módulo1.c la variable ***double precio*** sólo declara, en cambio en la segunda opción se muestra que se la define.

Opción A

modulo1.c

```
double sueldo;

int
main(void)
{
   printf("%g\n", sueldo);
  return 0;
}
```

modulo2.c

```
double sueldo = 3;
.....
```

+ ⇒ **ejecutable**

con sólo ver este fragmento no podríamos darnos cuenta que es sólo una declaración. Viendo que en el otro módulo se la define nos damos cuenta que esto es declaración

con sólo ver este fragmento sabemos que es una definición por la inicialización de la variable

Opción B

modulo1.c

```
double sueldo;

int
main(void)
{
   printf("%g\n", sueldo);
  return 0;
}
```

modulo2.c

```
extern double sueldo;
.....
```

+ ⇒ **ejecutable**

con sólo ver este fragmento no podríamos darnos cuenta que es una definición. Viendo que en el otro módulo solo se la declara nos damos cuenta que esto es la definición.

con solo ver este fragmento ya nos damos cuenta que es sólo una declaración

Ejemplo 3:

A continuación se muestran los errores de compilación y linkedición que se obtendría al generar un único programa ejecutable a partir de los siguientes dos módulos

modulo1.c

```
static int dia;

int
main(void)
{
        extern float valor;
        int dato;
        static int dato;
        .....
}
```

modulo2.c

```
static int dia;

extern float valor;

float
suma(void)
{
        extern float valor;
        ....
}
```

No es necesario declararla nuevamente, ya que se trata de la misma variable *valor* externa y global

El **modulo1.c** presenta errores de compilación debido a la redeclaración de la variable ***dato*** dentro del mismo alcance

El **modulo2.c** no presenta errores de compilación

En cuanto a la linkedición, suponiendo que eliminamos la declaración ***int dato*** del **modulo1.c**, se obtiene un error debido a que la variable ***valor*** no fue definida (en todos los módulos están solamente declaradas).

Notar que la declaración ***extern float valor,*** dentro de la función ***suma*** en el **modulo2.c,** está de más (está dentro del alcance de la declaración global del mismo módulo).

A su vez la variable ***dia***, en ambos módulos, refiere a objetos distintos: cada una de dichas definiciones sólo afecta el módulo en el que se encuentran debido a que son globales y están afectadas por el calificador de static, restringiendo su visibilidad.

***Ejemplo 4**:*

A continuación se muestran los errores de compilación y linkedición que se obtendría al generar un único programa ejecutable a partir de los siguientes dos módulos

modulo1.c

```
float valor= 15;

int
main(void)
{
        static int dato;
        .....
}
```

modulo2.c

```
extern float valor= 3;

float
suma(void)
{
        ....
}
```

Ningun módulo presenta errores de compilación

En cuanto a la linkedición, se obtiene un error por redefinción de la variable ***valor***, ya que la misma fue definida en **modulo1.c** por ser inicializada y también en el **modulo2.c**

4. Ejercicios

Ejercicio1

Dado el siguiente código fuente, que es el único módulo para formar un programa ejecutable:

a) Para cada una de las variables indicar persistencia, alcance y enlace. Decir además si se las está sólo declarando o definiendo.

b) Indicar los errores que se obtendrían en tiempo de compilación.

c) Indicar los errores que se obtendrían en tiempo de linkedición debido a no poder resolver alguna referencia externa o redefinición de variables.

d) Mostrar qué se obtendría en la salida estándar al finalizar la ejecución del mismo, si se obtiene de la entrada estándar: **No¿**

```
1.    #include <stdio.h>
2.
3.    int contador;
4.    int letra= 'W';
5.
6.    int
7.    main(void)
8.    {
9.          int letra;
10.         printf("Valor de letra al comenzar el bloque: %c\n",letra);
11.         while ( (letra= getchar() ) != EOF )
12.         {
13.               if ( 'a' <= letra && letra <= 'z')
14.                     putchar( letra – 'a' +  'A');
15.               else
16.                     putchar( letra );
17.               contador++;
18.         }
19.         printf("Cantidad de letras ingresadas: %d\n", contador);
20.         return 0;
21.   }
```

Respuesta:

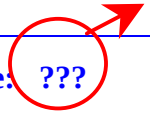
a) Para cada una de las variables indicar persistencia, alcance y visibilidad.

variable y línea	*persistencia*	*alcance*	*visibilidad*	*definición vs declaración*
contador (linea 3)	estática	global	externo	definición
letra (linea 4)	estática	global	externo	definición
letra (linea 9)	automática	local	sin enlace	definición

b) No hay errores de compilación.

c) No hay errores de linkedición.

d) Se obtendría: (este valor es impredecible)



```
Valor de la letra a comenzar el bloque: ???
NO
Cantidad de letras ingresadas: 3
```

Notar que las variables de alcance global no hace falta inicializarlas aunque funcionen como acumulador. Si la variable ***contador*** hubiera tenido alcance local (se hubiera definido dentro del bloque) tendría que haberse inicializado con cero.

Como se observa, en la línea 9 aparece otra vez una declaración de variable, pero como no se hace referencia explícita sobre su correspondencia a la misma variable ya definida en la línea 4, corresponde a una nueva definición de variable (que inclusive podría ser de otro tipo). La variable ***letra*** de la línea 9 tiene alcance local, y tiene prioridad sobre la ***letra*** global de lalínea 4.

Ejercicio 2

Idem al ejercicio anterior, pero donde se tiene dos programas fuentes que juntos formaran un único archivo ejecutable

módulo 1.c

```
1.    #include <stdio.h>
2.
3.    void
4.    imprime(void)
5.    {
6.          printf("el valor de letra es %c\n", letra);
7.    }
8.
9.    int
10.   main(void)
11.   {
12.         extern int letra;
13.         int contador= 0;
14.
15.         imprime();
16.         printf("Valor de letra al comenzar el bloque: %c\n",letra);
17.         while ( (letra= getchar() ) != EOF )
18.         {
19.              if ( 'a' <= letra && letra <= 'z')
20.                   putchar( letra - 'a' +  'A');
21.              else
22.                   putchar( letra );
23.              contador++;
24.         }
25.         printf("Cantidad de letras ingresadas %d\n", contador);
26.         return 0;
27.   }
```

módulo2.c

```
28.   int letra= 'W';
```

Respuesta:

a) Para cada una de las variables indicar persistencia, alcance y visibilidad.

variable y línea	*persistencia*	*alcance*	*visibilidad*	*definición vs declaración*
letra (linea 12)	estática	local	externo	declaración
contador(linea 13)	automática	local	sin enlace	definición
letra (linea 28)	estática	global	externo	definición

b) NO compila porque en la línea 5 se hace referencia a una variable no declarada previamente. Notar que la variable letra de la línea 11 tiene alcance local por lo tanto tampoco hubiera servido cambiar el orden de definición de las funciones.

c) Una vez declarada la variable dentro de la función imprime, no hay errores de linkedición:

```
void imprime(void)
{
        extern int letra;
        printf("el valor de letra es %c\n", letra);
}
```

d) Se obtendría:

```
El valor de la letra es W
Valor de la letra al comenzar el bloque:  W
NO
Cantidad de letras ingresadas 3
```

Notar que la variable ***letra*** definida en el **modulo2.c**, si bien tiene enlace externo y puede ser por lo tanto usada desde otro módulo fuente, exige que el otro módulo que la piensa usar la declare.

Además la variable ***contador*** en el módulo1 se la convirtió en automática, requiriendo su inicialización en cero para funcionar correctamente como contadora.

Ejercicio 3

Indicar en cada uno de los siguientes casos si existen o no errores de compilación y/o linkedición. Para cada una de las variables decir en que zona de memoria (que segmento) la almacenaría.

a)

```
int
cantidadDeCifras( int valor)
{
      static int acumulador= valor;

      while (valor > 0)
      {
            acumulador++;
            valor = valor / 10;
      }

      return acumulador;
}



int
main(void)
{
      cantidadDeCifras(10);

      return 0;
}
```

b)

```
int
cantidadDeCifras( int valor)
{
      static int acumulador;

      acumulador += valor;

      while (valor > 0)
      {
            acumulador++;
            valor = valor / 10;
      }

      return acumulador;
}
```

c)

```
int
main(void)
{
      int letra;

      while ( ( letra = getchar() ) != EOF )
      {
            static int acumulador;
            acumulador++;
      }

      printf("Cantidad de letras = %d\n", acumulador);

      return 0;
}
```

d)

```
int
main(void)
{
      int letra;

      while ( ( letra = getchar() ) != EOF )
      {
            static int acumulador;

            acumulador++;

            printf("Cantidad de letras = %d\n", acumulador);
      }

      return 0;
}
```

Respuestas:

a) **Error de compilación: "initializer element is not constant" .**

Como se observará la variable static ***acumulador*** definida dentro del **main**, es una variable que se almacenará en el Data Segment por haber intentado inicializarlo. Sin embargo como ya se explicó en la sección 2, la zona Data Segment garantiza que las variables allí almacenadas comienzan con el valor solicitado antes de comenzar la ejecución del proceso. Esto sería imposible de

garantizarse en el ejemplo, ya que un parámetro formal adquiere recién su valor durante la ejecución.

La única forma de inicializar variables static es con valores constantes (parecido a lo que ocurre con los labels del switch)

Cabe aclarar que la variable ***valor*** se almacenaría en el Stack Segment.

b) **No hay errores de compilación, ni de linkedición.**
La variable ***acumulador*** se almacena en el BSS Segment (por ser static pero no haber sido inicializada por el programador)
La variable ***valor*** se almacena en el Stack Segment.

IMPORTANTE: Existe una gran diferencia entre inicialización y asignación

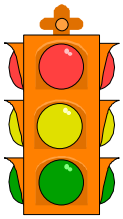
c) **Error de compilación:** la variable ***acumulador*** tiene alcance local y se pretende usarla fuera del bloque donde fue definida. De poder compilarse (no usándola fuera del bloque), dicha variable se almacenaría en el BSS Segment por ser static y no haber sido inicializada por el programador.

d) **No hay errores de compilación, ni de linkedicion.**
La variable ***letra*** se almacena en el Stack Segment. La variable ***acumulador*** se almacena en el BSS Segment por ser static y no haber sido inicializada por el programador. Nótese que por ser static, el sistema la inicializa en cero al cargar el proceso en memoria (antes de la ejecución), pero su persistencia es estática. Esto ultimo garantiza que su valor se conserva durante toda la ejecución del proceso, así es como cada vez que se alcanza le ejecución del bloque no vuelve a pasarse por una inicialización, sino que conserva su valor anterior (si estuviera en el Stack Segment, al crearse y destruirse dinámicamente en cada entrada al bloque, perderían sus valores antes adquiridos)

Reglas sobre Definiciones y Declaraciones

- Para ANSI los identificadores externos deben diferir en los primeros 6 caracteres.
- Los archivos de encabezado que declaran funciones o variables externas deben ser incluidos en el archivo que define dichas funciones y variables. De esta forma el compilador puede hacer chequeo de tipos y la declaración externa siempre coincidirá con la definición. Los linkeditores no suelen arrojar errores al linkeditar módulos donde aparecen variables extern con el mismo nombre pero distintos tipos, ocasionando obviamente serios problemas durante la ejecución
- Cualquier variable que se pretende que comience con un valor debe ser explícitamante inicializada, o al menos debe comentarse la aceptación de su inicializacion por omisión, si ésta existe.
- Las declaraciones globales deben comenzar en la columna 1.
- Todas las declaraciones de datos externos deben ser precedidas por la palabra extern.
- Si una función usa alguna variable externa que **no haya sido declarada globalmente** en el archivo, debería tener su propia declaración en el cuerpo de la función utilizando la palabra clave extern.
- Todas las declaraciones que no están relacionadas deben estar en líneas separadas, aunque sean del mismo tipo.
- Evitar declaraciones locales que sobre-escriban declaraciones de niveles más alto. En particular las variables locales no deberían ser redeclaradas en bloques anidados.
- Las variables que deban ser accedidas desde otros archivos (visibilidad externa) deben sólo ser usadas cuando es demostrable que no existe otra opción para resolver el problema. A lo sumo se usará el calificador ***static*** cuando se precise una variable global en un módulo que integra un gran sistema, para reducir su visibilidad en el resto.
- Definir variables en un archivo de encabezamiento es una mala idea. Evidencia una pobre modularización del código de un sistema y pueden ocasionar redefinición de variables.

Muy Muy Muy Importante para esta Materia



Todo el tema de la gestión de memoria se ha desarrollado en detalle, por ser parte importante dentro del lenguaje C.

El uso de variables de ***enlace externo*** sólo es entendible en grandes proyectos, el cual está fuera de lo límites de las materias ***Programación I*** y ***Estructura de Datos y Algoritmos.***

Por otra parte, el uso de variables de ***alcance global*** denotan un estilo muy pobre de programación, ya que, al igual que las externas, compartir información entre distintas funciones sin usar pasaje de parámetros, enturbia notablemente la semántica de los programas y acarrea efectos colaterales indeseables.

Toda variable debe ser declarada justo antes de ser usada (declaración tardía) y debe haber una sola función responsable por el contenido que posee la misma. Así es como sólo se usarán variables con **alcance local y sin enlace**. Los programas que se desarrollen en las materia antes citadas **no precisan** bajo ningún punto de vista el uso de otra clase de almacenamiento.

Así como no se permite el uso de la proposición ***goto*** (aunque exista en el lenguaje) por no aportar claridad, ni mantenibilidad a los programas, tampoco se permitirán variables **globales** y **externas**, por provocar una semántica poco clara y predisponer a efectos colaterales no deseables.

Siempre deberá utilizarse ***declaración tardía*** y ***pasaje de parámetros*** como única forma aceptable de pasaje de información entre una función y otra (ya sea dentro del mismo programa fuente o no).

La Cátedra

Biblioteca Estándar – Primera Parte

Introducción

El lenguaje C es bastante reducido, sin embargo ofrece un conjunto de funciones en la biblioteca estándar que lo potencian y que si bien no forman parte del lenguaje se encuentran disponibles en cualquier paquete del lenguaje C.

En este documento se presentan algunas de sus funciones y se las utiliza en algunos ejemplos.

1. La Biblioteca Estándar para Entrada/Salida (Standard I/O Library)

Provee un conjunto de funciones útiles para manipular archivos y operaciones de entrada/salida.

Los prototipos de sus funciones se encuentran en el archivo de encabezamiento **stdio.h**

Citaremos algunas de sus funciones:

Prototipo	*Descripción*
int getchar(void);	Lee un carácter de la entrada estándar, y retorna un entero para permitir detectar el fin de archivo (EOF). Ya la vimos
int putchar (char ch);	Escribe el carácter en la salida estándar. Ya la vimos
ungetc(char ch, FILE * infile);	Coloca el caracter ch (no puede ser EOF) otra vez en el stream indicado por ***infile***, haciendo que el mismo esté disponible otra vez en el próxima lectura. Si se desea colocar el dato otra vez en la entrada estándar, el segundo parámetro debe invocarse con stdin. Devuelve el carácter ch si todo fue exitoso o EOF en caso contrario. Se garantiza que por lo menos un carácter es devuelto a la entrada estándar (al realizar sucesivos ungetc sin su correspondiente getchar posterior).
void printf(char*, ...);	Escribe en la salida estándar con el formato pedido. Ya la vimos

Ejemplo:

Si se obtiene de la entrada estándar abc↵ con el siguiente programa

```
#include <stdio.h>

int
main(void)
{
      int letra;

      if ((letra= getchar()) != EOF)
      {
            printf("Primera lectura del buffer: %c\n", letra);

            ungetc(letra, stdin);
            letra= getchar();

            printf("Segunda lectura del buffer: %c\n", letra);
      }
      return 0;
}
```

Se obtendría en la salida estándar:

Primera lectura del buffer: a
Segunda lectura del buffer: a

2. La Biblioteca Estándar para el Sistema (Standard System Library)

Provee un conjunto de funciones útiles de próposito general.

Los prototipos de sus funciones se encuentran en el archivo de encabezamiento **stdlib.h**

Citaremos a continuación algunas de sus funciones:

Prototipo	*Descripción*
int abs(int n);	Devuelve el valor absoluto de un número de tipo int
long labs(long n);	Devuelve el valor absoluto de un número de tipo long
int rand(void);	Devuelve un número pseudo-aleatorio perteneciente al rango entre 0 y RAND_MAX inclusive
void srand(unsigned int seed);	Setea la semilla generadora de la próxima secuencia de números pseudo-aleatorios con el valor especificado. Si se invoca por primera vez la función **rand()** sin haber invocado esta función antes, el sistema utiliza la semilla 1.
void abort(void);	Produce una finalización anormal del proceso en ejecución.
int exit(int status);	Produce una terminación normal del proceso en ejecución (cierra los flujos abiertos, devuelve el control al entorno que invocó la ejecución del proceso, etc). Puede utilizarse los parámetros EXIT_SUCCESS y EXIT_FAILURE para indicarle al entorno que la terminación fue o no exitosa.

Muy Importante

Cuando se le da el control a un proceso (por ejemplo al pedir su ejecución desde la línea de comandos) comienza a ejecutar la primera instrucción ejecutable dentro del módulo ***main***.

Si estamos realizando **programación estructurada** debe **haber un único punto de regreso** al entorno que invocó la ejecución de dicho programa: ésto se logra simplemente haciendo ***return 0*** o bien ***return EXIT_SUCCESS*** para finalizar normal o bien ***return EXIT_FAILURE*** o cualquier otro valor para finalización anormal.

Así es como **NO** existe ninguna necesidad de utilizar ni las funciones ***abort()*** ni ***exit(nro)*** desde el main. Aunque estas funciones podrían ser usadas desde cualquier función (distinta de main), que decidiera devolver el control directamente al entorno (sin volver a la función que en realidad la invocó), su aplicación resulta **absolutamente inadmisible**.

En la materia **Programación I y Estructura de Datos y Algoritmos NO se va a hacer uso de ninguna de estas dos funciones por estar violando el principio de Programación Estructurada.**

Ejemplo:

Si se ejecuta el siguiente programa:

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

int
main(void)
{
      printf("|%d|= %d\n", -5, abs(-5) );
      printf("|%ld|= %ld\n", -20L, labs(-20L) );

      printf("Valor pseudo aleatorio en el rango [0, %d]: %d\n",
                                                RAND_MAX, rand());

      return 0;
}
```

Se obtendría en la salida estándar (para una arquitectura en particular de 32 bits):

```
|-5|= 5
|-20|= 20
Valor pseudo aleatorio en el rango [0,  2147483647]:  1804289383
```

Aclaración

Nótese que en el ejemplo anterior el rango de los valores pseudo aleatorios depende del tamaño del entero en dicha arquitectura (sizeof(int)).

Además si se ejecuta ese mismo programa en la misma computadora sucesivas veces siempre se obtiene el mismo número pseudoaleatorio. Esto es debido a que se está usando un algoritmo generador de números pseudoaleatorios, que inicia sus cálculos a partir de la semilla especificada. Cuando ésta se omite el algoritmo generador siempre comienza por la semilla 1.

No olvidarse de invocar la función que setea la semilla con un valor distinto en los programas que usan esta función, con el fin de obtener distintas secuencias de números pseudoaleatorios.

3. La Biblioteca Estándar para Clasificación de Caracteres (Character Type Library)

Provee un conjunto de funciones útiles para clasificar caracteres. Todas las funciones reciben un entero que representa un caracter representable por un unsigned int o el valor EOF.

Los prototipos de sus funciones se encuentran en el archivo de encabezamiento **ctype.h**

Citaremos todas sus funciones. Las armamos en dos grupos porque el valor que devuelven difieren.

a)

Las funciones de la siguiente tabla regresan un valor diferente de cero si el argumento se satisface o cero en caso contrario

Prototipo	*Descripción*
int isupper(int ch);	Testea si el caracter ch es una letra mayúscula (del alfabeto inglés)
int islower(int ch);	Testea si el caracter ch es una letra minúscula (del alfabeto inglés)
int isalpha(int ch);	Testea si el caracter ch es una letra mayúscula o minúscula (del alfabeto ingles)
int isdigit(int ch);	Testea si el caracter ch es un dígito decimal
int isxdigit(int ch);	Testea si el caracter ch es un dígito hexadecimal
int isalnum(int ch);	Testea si el caracter ch es un dígito, una letra mayúscula o minúscula (del alfabeto ingles)
int ispunct(int ch);	Testea si el caracter ch es un símbolo de puntuación
int isspace(int ch);	Testea si el caracter ch es un caracter blanco, entendiéndose por esto cualquiera de los siguientes: ‘ ‘ (espacio en blanco), ‘\t’, ‘\n’, ‘\f’, ‘\v’
int isprintf(int ch);	Testea si el caracter ch es imprimible, incluyendo los caracteres blancos
int isgraph(int ch);	Testea si el caracter ch es imprimible pero no un caracter blanco
int iscntrl(int ch);	Testea si el caracter ch es de control

b)

Las funciones de la siguiente tabla intentan hacer una conversión. Si se realiza devuelve dicha conversión pedida, caso contrario retorna el valor recibido intacto.

Prototipo	*Descripción*
int toupper(int ch);	Testea si el caracter ch es una letra minúscula (del alfabeto inglés)y la convierte a mayúscula.
int tolower(int ch);	Testea si el caracter ch es una letra mayúscula (del alfabeto inglés)y la convierte a minúscula.

Ejemplo:

El siguiente programa lee desde la entrada estándar y coloca en la salida estándar todos los caracteres pasados a mayúsculas (en el alfabeto ingles) eliminando los caracteres blancos.

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

int
main(void)
{
      int letra;

      while ((letra= getchar()) != EOF)
            if ( ! isspace(letra) )
                  putchar( toupper(letra) );

      return 0;
}
```

Si se obtuviera de la entrada estándar:

```
      Nueva PRUEBA

que   elimina                 espacios

y pasa a MAYusculas. 123.
```

La salida estándar sería:

```
NUEVAPRUEBAQUE ELIMINAESPACIOSYPASAAMAYUSCULAS.123.
```

4. La Biblioteca Estándar Matemática (Math Library)

Provee un conjunto de funciones útiles para realizar cálculos matemáticos.

Los prototipos de sus funciones se encuentran en el archivo de encabezamiento **math.h**

Citaremos algunas de sus funciones:

Prototipo	*Descripción*
double fabs(double x);	Devuelve el valor absoluto de un número de tipo double
double floor(double x);	Retorna en un tipo double la representación del entero más grande menor o igual al parámetro x.
double ceil(double x);	Retorna en un tipo double la representación del entero más chico mayor o igual al parámetro x.
double fmod(double x, double y);	Retorna en un tipo double el resto de división entre x e y, con el mismo signo que x. Si el segundo parámetro fuera cero el resultado es dependiente de la implementación
double sqrt(double x);	Retorna la raíz cuadrada del argumento x. Obviamente x debe ser mayor o igual a cero.
double pow(double x, double y);	Retorna x^y, o sea x elevado al argumento y. Se obtendrá un error de dominio si x=0 y y≤0. o bien si x<0 y el argumento y no representa un entero
double exp(double x);	Retorna e^x, o sea el numero e elevado al argumento x
double log(double x);	Retorna ln(x), o sea el logaritmo natural de x. El argumento debe ser un número mayor que cero
double log10(double x);	Retorna $\log_{10}(x)$, o sea el logaritmo en base 10 de x. El argumento debe ser un número mayor que cero
double sin(double angulo);	Retorna el seno del argumento angulo, donde el mismo debe ser expresado en radianes
double cos(double angulo);	Retorna el coseno del argumento angulo, donde el mismo debe ser expresado en radianes
double tan(double angulo);	Retorna la tangente del argumento angulo, donde el mismo debe ser expresado en radianes
double asin(double x);	Retorna $seno^{-1}(x)$, o sea el arcoseno del argumento x. El argumento debe pertenece al intervalo [-1, 1], y el resultado obtenido representa un angulo expresado en radianes que se encuentra en el intervalo $[-\pi/2, \pi/2]$

Prototipo	*Descripción*
double acos(double x);	Retorna $\cos^{-1}(x)$, o sea el arcocoseno del argumento x. El argumento debe pertenece al intervalo [-1, 1], y el resultado obtenido representa un angulo expresado en radianes que se encuentra en el intervalo $[0, \pi]$
double atan(double x);	Retorna $\tan^{-1}(x)$, o sea el arcotangente del argumento x. El resultado es un ángulo expresado en radianes que se encuentra en el intervalo $[-\pi, \pi]$
double atan2(double y, double x);	Retorna el ángulo formado entre el eje x y la línea que se extiende desde el origen al punto (x, y). x. El resultado es un ángulo expresado en radianes
double sinh(double x);	Retorna el seno hiperbólico del argumento x
double cosh(double x);	Retorna el coseno hiperbólico del argumento x
double tanh(double x);	Retorna la tangente hiperbólica del argumento x
double ldexp(double x, double n);	Retorna $x\ 2^n$

Muy Importante

Como estas funciones son aplicadas a argumentos y el usuario puede intentar usarlas con argumentos fuera del dominio de definición de la función, o bien puede obtener valores fuera del rango representable en la arquitectura de computadora donde se está ejecutando el proceso, se decidió definir una variable global y externa ***errno*** la cual está declarada en el archivo de encabezamiento **errno.h**

Cuando el programador invoca alguna de la funciones que pueden arrojar alguno de estos problemas debe testear el valor con que queda dicha variable después de la invocación de la función, ya que la misma es seteada automáticamente cuando hay problemas con alguna de las siguientes constantes simbólicas (también definidas en el archivo de encabezamiento errno.h):

- ***EDOM***: si hubo un error en el dominio de la función.
- ***ERRANGE***: si el resultado que debe devolverse desborda el tamaño del double. En este caso además la función invocada devuelve el valor ***HUGE_VAL***.
- ***ERRANGE*** u otro valor (dependen del compilador): si el resultado que debe devolver es tan pequeño que tampoco puede representarse en el double (underflow). En este caso lo que se asegura es que la función devuelve **cero** (como representante del valor pequeño).

Muy Importante

Dicha variable global y externa ***errno*** sólo se setea en el caso de que hayan habido errores, por lo tanto si se invoca una función que produce algún ***error*** queda con alguno de los valores antes citados. Pero si después se invoca otra función matemática que no produce errores el contenido de ***errno*** sigue con el valor anterior (el sistema no se encarga de blanquearla**).**

Es absoluta responsabilidad del programador blanquearla antes de invocar a una función y testear su valor cuando la misma retorna, ya que olvidarse de blanquearla explícitamente puede llevar al programador a la conclusión de que hubo algún error que en realidad no existió (error de arrastre).

Esto evidencia una vez más que el uso de variable globales enturbia la semántica de un programa y puede produce efectos colaterales indeseables.

Mucho mejor hubiera sido que el lenguaje C no hubiera decidido en su diseño setear errores en una variable global, y en cambio, hubiera devuelto los errores en algún argumento extra para que el programador testeara algún posible error de dicho parámetro. La única forma prolija de pasar información entre funciones DEBE SER por medio de parámetros.

Hoy en día, mucho años después de la creación del lenguaje C, los nuevos lenguajes (orientados a objetos) se diseñan sin variables globales para el pasaje de información.

Aclaración

En Unix para que el programa *cc* linkedite con la librería matemática hay que utilizar la opción –l de la línea de comandos (con la letra m para indicarle el uso de la librería estándar matemática).

Suponiendo que el programa pepe.c usa la librería estándar matemática, habría que hacer:

$ cc pepe.c -lm

Ejemplo:

Si se ejecutara el siguiente programa

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

int
main(void)
{
      printf("Floor( %g )= %g\n", 23.5, floor( 23.5 ) );
      printf("Ceil( %g )= %g\n", 23.5, ceil( 23.5 ) );

      printf("Floor( %g )= %g\n", -23.5, floor( -23.5 )  );
      printf("Ceil( %g )= %g\n", -23.5, ceil( -23.5 )  );

      return 0;
}
```

Se obtendría en la salida estándar:

```
Floor( 23.5 )= 23
Ceil(  23.5 )= 24
Floor( -23.5 )= -24
Ceil(  -23.5 )= -23
```

Ejercicio:

El siguiente programa intenta realizar dos lecturas de números punto flotante desde la entrada estándar. Para cada una de dichos valores calcula el valor de la raíz cuadrada y lo imprime si no hay errores obtenidos.

a) Decir qué se obtendría si desde la entrada estándar se ingresara –144 y 25

b) Arreglarlo para que funcione correctamente.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include “getnum.h”

int
main(void)
{
      double rta;
      float leido;

      leido= getfloat("Ingrese un valor:");

      rta= sqrt(leido);
      if (errno == EDOM)
            printf("Error en el dominio de la funcion\n");
      else
            printf("sqrt( %f ) = %g\n", leido, rta );

      leido= getfloat("Ingrese un valor:");

      rta= sqrt(leido);
      if (errno == EDOM)
            printf("Error en el dominio de la funcion\n");
      else
            printf("sqrt( %f ) = %g\n", leido, rta );

      return 0;

}
```

Respuestas:

a) Error en el dominio de la función
Error en el dominio de la función

b) El error surge por no haberse acordado de blanquear el valor de la variable errno antes de la segunda invocación (se arrastra el valor anterior). Típicamente debe blanquearse siempre la variable antes de la invocación de cada función que deje algún valor en dicha variable:

```
errno= 0;
rta= sqrt(leido);
if (errno == EDOM)
      printf("Error en el dominio de la funcion\n");
else
      printf("sqrt( %f ) = %g\n", leido, rta );
```

explícitamente blanquear esta variable justo antes de invocar la función que puede setear algún código de error en ella

Aclaración Muy Importante

Es bueno utilizar las funciones de la librería estándar que ofrece el lenguaje C por tres motivos:

- **No hay que reinventar la rueda. No tiene sentido perder tiempo re-programando funciones clásicas que ya fueron implementadas.**
- **Como dichas funciones son usadas por millones de usuarios, están altamente testeadas. Es más probable que introduzcamos errores si las re-programamos.**
- **Fueron implementadas de la forma más eficiente posible**

Sin embargo, cuando se está estudiando materias como programación es interesante, en tiempo de práctica, intentar pensar cómo podrían implementarse (por lo menos algunas de ellas) a los efectos de practicar el estilo del lenguaje. Ese es el motivo por el cual, a veces, incluimos la re-escritura de alguna de ellas por parte del alumno (aunque después utilice la que viene con la librería estándar).

Preprocesador - Parte II

Introducción

En este documento se presentan opciones avanzadas del preprocesador de C. Las mismas permiten evitar la redefinición de identificadores y sirven a los efectos de aplicar técnicas de debuggeo.

1. Inclusión Condicional

Muchas veces queremos que cierto código no llegue a ser visto por el compilador. Para esto nos sirve el uso de la directiva de inclusión condicional.

Un típico caso donde se la usa es en la escritura de un módulo que no va a ser multiplataforma por utilizar funciones de muy bajo nivel que son dependientes de la arquitectura de computadora donde se va a ejecutar.

En este caso es de gran utilidad hacer que "cierto código" sea compilado si se está en cierta arquitectura, pero que no sea tenido en cuenta si se está en otra arquitectura.

Sintaxis

```
#ifdef  IDENTIFICADOR
....
#endif
```

Muy Importante

No confundir una inclusión condicional con el **operador condiciona**l o con la instrucción ***if else*** del lenguaje C.

Las directivas al preprocesador son resueltas por éste antes de que el compilador genere código objeto.

Las líneas de código comprendidas entre el ***#ifdef*** y el ***#endif*** permanecen para que las analice el compilador, siempre que el preprocesador haya encontrado definido al IDENTIFICADOR antes de alcanzar la directiva de inclusión condicional.

Para poder definir al IDENTIFICADOR antes de que se alcance dicha línea se procede obviamente con otra directiva al preprocesador

Sintaxis

#define IDENTIFICADOR

Lo cual tiene una semejanza con la definición de una constante simbólica, pero sin molestarse en asociarle valor alguno.

El alcance de una definición de identificador es hasta el final del lote fuente, por lo tanto si se quisiera más adelante des-definir dicho identificador se procede con otra directiva al preprocesador

Sintaxis

#undef IDENTIFICADOR

Ejemplo:

El archivo de encabezamiento ***time.h*** de Borland C tiene un fragmento donde define a una constante simbólica con cierto valor según hay encontrado o no definida la constante __OS2_. Claramente para el compilador de Borland C los sistemas operativos DOS y OS2 tienen puntos en común. El diseñador de dicha biblioteca decidió definir con distintos valores a las constantes simbólicas CLOCKS_PER_SEC y CLK_TCK y no quiso escribir un archivos de encabezamiento distinto para cada uno de ellos. Prefirió tener todo en un mismo archivo y colocar inclusiones condicionales donde correspondan.

```
.....

#ifdef __OS2__
        #define CLOCKS_PER_SEC 1000
        #define CLK_TCK       1000
#else
        #define CLOCKS_PER_SEC 18.2
        #define CLK_TCK       18.2
#endif

.......
```

Importante

Los trabajos realizados en **Programación I** y en **Estructura de Datos y Algoritmos no son de bajo nivel**.

Recordar que el objetivo de la materia es desarrollar aplicaciones multiplataforma. Sin embargo existen una variedad de casos donde el uso de la inclusión condicional resulta de gran utilidad.

Ejemplo:

Las directivas antes vistas pueden ser usadas como técnica de debuggeo: se puede generar determinado código según se esté corriendo la versión **debugger**, y eliminarlo cuando se esté generando la versión **release**.

Podríamos escribir en la salida estándar "printf" con determinada información mientras se testea el software, para lo cual haríamos:

```
printf("El valor de la variable x=%dºn", x);
```

Pero como no queremos que esos "printf" aparezcan en la versión ***release***, y tampoco queremos tomarnos el trabajo de eliminarlos del archivo fuente, los escribimos dentro de una inclusión condicional:

```
#ifdef DEBUG
        printf("El valor de la variable x=%dºn", x);
#endif
```

Con esto estaría casi listo, salvo que no debemos olvidarnos de que el **preprocesador** encuentre definido el identificados **DEBUG** antes de la directiva ***#ifdef*** en la versión debugger, debiendo no encontrarlo en la versión ***release***. Entonces en el archivo fuente que incluya esas directivas tendremos:

a) Para la versión ***debugger***

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

#ifdef  DEBUG
        printf("El valor de la variable x=%dºn", x);
#endif
...
```

b) Para la versión ***release***

Sólo faltaría editar ese archivo y eliminar la primera línea, quedando

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

#ifdef DEBUG
        printf(“El valor de la variable x=%dºn”, x);
#endif
```

Nota

La inclusión condicional puede servir para evitar incluir un archivo de encabezado más de una vez, colocando en cada header lo siguiente:

```
/* archivoHeader.h  */

#ifndef   NOMBRE_DEL_HEADER

        #define NOMBRE_DEL_HEADER

        .....                          } cuerpo del archivo
        .....
        .....

#endif
```

De esta forma si el preprocesador encontrara un **#include** del archivo **archivoHeader.h** más de una vez, en la primera copiaría su contenido, porque la constante **NOMBRE_DEL_HEADER** no fue definida antes, pero el resto de las veces ya no volvería a copiarlo por encontrar definida a dicha constante.

Abrir cualquiera de los archivos de encabezamiento que vienen con el paquete del lenguaje C y ver que todos usan este esquema.

2. Macro Assert

Existe una macro que puede ser usada para testear si determinada situación ocurre en nuestros códigos, en tiempo de ***debuggeo.***

Si el identificador **NDEBUG** está definido antes de incluir el header **assert.h,** dicha macro es ignorada por el preprocesador, caso contrario es tenida en cuenta.

Sintaxis

void assert(int expresion)

Si la expresión fuera evaluada como cero, entonces se imprimirá en el flujo stderr un mensaje del estilo

Assertion failed: ***expresion*****, file** ***filename*****, line** ***num***

y abortará la ejecución del programa.

Obviamente una terminación del programa desde cualquier función es sólo admisible porque esto ocurre en tiempo de prueba de software y no en la versión ***release.***

Es muy útil el uso de **assert** en tiempo de debuggeo, pues no hace perder tiempo al desarrollador con la ejecución total del programa cuando ya se dio cuenta de que se produjo algún error. En cambio esto sería inadmisible en la versión release que recibe el usuario, para el cual el programa debe informar sobre los posibles errores que ocurran sin abortar bruscamente.

Se colocará ***assert*** en aquellos lugares del código donde se supone que "debería una expresión valer distinta de cero", y por lo tanto el hecho de que valga cero es inadmisible y merece detener la ejecución, arreglar el código y re-testear.

Reglas para Compilación Condicional

- Para prevenir la accidental **doble inclusión** de archivos de encabezamiento, incluir lo siguiente en todo archivo header:

```
#ifndef     EXAMPLE_H
#define     EXAMPLE_H

        /* cuerpo del archivo example.h

#endif
```

- Usar la macro ***assert*** para asegura que cada función recibe valores bien definidos y que los cálculos intermedios también están bien formados.

- Salvo que se trabaje en bajo nivel (módulos de un sistema operativo, drivers), la compilación condicional usada para elegir una arquitectura de computador o sistema operativo, debe evitarse.

Armado de una Biblioteca

Introducción

En este documento se discuten heurísticas para diseñar funcionalidades relacionadas en bibliotecas que permitan potenciar el desarrollo de aplicaciones.

1. Construcción de una Biblioteca

Existen dos perspectivas para analizar una biblioteca:

- **Perspectiva del implementador**: que intenta imaginarse ciertas funcionalidades que pueden resultar útiles y decide armar la biblioteca. Dado que la única forma de comunicación válida entre las funciones que ofrece y las aplicaciones que las vayan a usar es la especificación de una interface, deberá decidir cuidadosamente "qué exportará o publicará" para la misma. Típicamente esa "publicación" debe contener la información necesaria para que los usuarios de las bibliotecas puedan comprender la potencialidad de la misma, y en el caso de las funciones serán justamente los prototipos de las mismas con los comentarios necesarios para saber "qué hace" y "no cómo" lo hace.

- **Perspectiva del usuario de la biblioteca**: que sólo conoce cómo invocar ciertas funcionalidades que posee una librería pero desconoce detalles de implementación de la misma. La ventaja de separar el código que invoca una biblioteca del código que la implementa es muy importante porque si la biblioteca cambia su implementación (no su interface) en versiones posteriores, el cliente no se ve afectado en su desarrollo más que por el hecho de tener que re-linkeditar su aplicación.

1.1 Su Diseño

El diseño de la misma corresponde a decidir qué funcionalidades podrá ofrecer. Este proceso ocurre en momento previo a la codificación y supone poder imaginarse distintos usos para la misma.

Ya que lo único que conocerá el usuario de la biblioteca será su *interface*, es muy importante reparar en ella y evaluarla según los siguientes heurísticas

- **Debe contener criterios unificados.** Las funcionalidades que se encuentren en una biblioteca deben estar relacionadas entre sí. Además la forma de acceder a la interface debe ser muy similar para todas las funciones que la integren (caso contrario conviene separar en varias interfaces, tal como lo hace la biblioteca estándar).

- **Debe ser simple**. La idea de usar bibliotecas surge como objetivo para reducir la complejidad de los programas (es una forma de modularización). La interface siempre debe ocultar los detalles complejos que tengan que ver con la implementación de la biblioteca (ni siquiera deben aparecer reflejados en los comentarios).

- **Debe ser suficiente**. Las funcionalidades serán las necesarias para poder utilizar la biblioteca en distintas aplicaciones. Si los usuarios encuentran que hay funciones que deberían ofrecerse y no están soportadas, buscarán otra biblioteca.

- **Debe ser general**. Una interface bien diseñada sirve a varios propósitos y no sólo para un caso. Recordar que un fuerte objetivo de Ingeniería de Software es la reusabilidad de código. Si una función hace demasiadas cosas es muy difícil reusarla y si hace demasiado poco no le sirve nadie. Si el desarrollo de una biblioteca surgiera en el momento de solución de un problema en particular, debe tomarse distancia de dicho problema específico y pensar a qué otras aplicaciones podría servir su desarrollo.

- **Debe ser estable**. Una vez publicada, una interface no debe cambiar. Justamente lo importante de la separación entre interface e implementación reside en poder cambiar detalles de esta última en el transcurso del tiempo, sin que por eso se vea afectado el código de los programas que usen la biblioteca.

1.2 Su Uso

El usuario de una biblioteca es aquel que, leyendo la documentación de la interface de la misma, está en condiciones de aprovechar las funcionalidades que ésta ofrece para reducir la complejidad de sus aplicaciones. Obviamente, remitirse a la especificación de una simple interface, buscando simplificar la estructura de un programa, implica usar un **nivel de abstracción** ya que consiste en usar módulos de los cuales se desconoce su codificación.

Una vez que se ha optado por alguna biblioteca, el programa del usuario se verá afectado por las funcionalidades publicadas en la interface de dicha biblioteca. Por lo tanto, el usuario **no deseará** que su proveedor le cambie dicha interface en las próximas versiones, porque ello implicaría tener que recambiar también su código. Hay que diferenciar entre el esfuerzo que significa **re-codificar** un programa y la acción de sólo **re-linkeditarlo**.

Nota:
En la mayoría de los casos el implementador de una biblioteca no es el que la usa. Sólo en tiempo de desarrollo el que implementa una biblioteca también escribe algún pequeño código para testearla. No caer en el error de que el código que la testea se vea afectado por el conocimiento interno de su implementación.

2. Ejemplo para la construcción de una biblioteca

Vamos a ponernos por un momento en la situación de querer diseñar e implementar una biblioteca que ofrezca funcionalidades para la generación de números pseudoaleatorios.

Cabe aclarar que usamos el término *pseudoaleatorios* y no aleatorios porque cuando utilizamos una computadora para generar dichos números, el funcionamiento determinístico de la misma (es predecible la salida de un programa porque responde a una secuencia de instrucciones en memoria) no permite la posibilidad de dejar realmente librado al azar cuál va a ser el número esperado, pero el algoritmo que se usará para ello hará parecer que los números realmente son aleatorios, porque su distribución es uniforme en el tiempo.

La idea de armar una biblioteca para números aleatorios responde al hecho de que las funciones para tal fin que ofrece la biblioteca estándar de C son muy escasas, y por lo tanto se puede crear un nivel de abstracción superior extendiendo las funcionalidades existentes y ofreciendo una capa superior que ayude en forma más efectiva a resolver problemas en los cuales se encuentre involucrado el azar.

Recordemos que las únicas funciones que ANSI C ofrece para este fin son:

- **srand(int)**
- **int rand(void)**

Las funcionalidades que **nuestra biblioteca** ofrece pueden ser resumidas en:

- Devolver un número real pseudoaleatorio que pertenezca al intervalo [0; 1).
- Devolver un número entero pseudoaleatorio perteneciente a un intervalo entero [min, max] donde dichos extremos sean especificados por el usuario y no solamente por el intervalo [0, RAND_MAX].
- Devolver un número real pseudoaleatorio perteneciente a un intervalo double [min, max] donde dichos extremos sean especificados por el usuario.
- Setear una nueva secuencia de números pseudoaleatorios.

2.1 Su Interface

```
/*
**    Interface para la biblioteca random
**    Version 1.0
**    Autores G & G
**    Fecha de ultima modificacion 01/01/1998
*/



/*    Esta funcion devuelve un número double pseudoaleatorio
**    perteneciente al intervalo [0, 1)
**    Ejemplo de uso: double n= randNormalize();
*/

double randNormalize(void);



/*    Esta funcion devuelve un número entero pseudoaleatorio
**    pertenciente al intervalo entero [izq, der]
**    Ejemplo de uso: int n= randInt(izq, der);
*/

int randInt( int izq, int der);



/*    Esta funcion devuelve un número entero pseudoaleatorio
**    pertenciente al intervalo double [izq, der]
**    Ejemplo de uso: double n= randReal(izq, der);
*/

double randReal( double izq, double der);



/*    Esta función cambia la secuencia de números pseudoaleatorio a
**    generar en las próximas invocaciones de randNormalize, randInt
**    y randReal. Se la suele invocar una sola vez
**    en la aplicación, pero no hay problema es usarla más veces.
**    Ejemplo de uso: randomize();
*/

void randomize(void);
```

Si esta interface es buena para la biblioteca que vamos a implementar debería responder positivamente a las heurísticas que nos hemos planteado:

❖ **¿Tiene criterios unificados?** Sí, la forma de uso de las funciones es similar (por lo menos en las que se parecen entre sí que son randInt y randReal).

❖ **¿Es simple?** La forma de implementación está oculta. La única función que pone en evidencia un problema de implementación es la necesidad de que el usuario invoque explícitamente la función randomize para cambiar la secuencia (estamos de alguna forma dándole a conocer la necesidad de que cambie la semilla generadora en el algoritmo), pero no es complicada de usar. Hay ciertas operaciones que no se les puede ocultar al usuario de la biblioteca: por ejemplo las funciones que se ofrecen en una biblioteca que escriba o lea datos de un archivo en disco, exigirá que el usuario invoque una función para "abrir" el mismo, y luego de usarlo lo "cierre". De alguna forma se está poniendo al descubierto que antes de leer o escribir a un archivo hay que abrirlo y después cerrarlo, pero se está manteniendo oculto un montón de operaciones que tienen que ver con la gestión que implica dicha apertura o cierre de archivo (alocar o desalocar buffers en memoria, guardar la fecha de ultima modificacion del archivo, etc.). Análogamente, nuestra función no le dice al usuario ni siquiera que se moleste en invocarla con un nuevo valor de semilla como lo hace srand (notar que randomize no tiene parámetros).

❖ **¿Es suficiente?** Es difícil de antemano imaginarse si nos olvidamos alguna función importante. Lo que se puede hacer es pensar varios ejemplo concretos (clientes a los cuales nos querríamos dirigir) y ver si para ese tipo de aplicaciones es suficiente. En nuestro caso apuntaremos a juegos (lotería, ruleta, dados, juego del ahorcado, etc.) y simulaciones (como el método de Montecarlo).

❖ **¿Es general?** Si la interface provista sirviera sólo para "juegos de dados", esto sería un indicador de poca generalidad. Sin embargo vemos que se adapta a diferentes juegos e inclusive a simulaciones, lo que hace pensar que posee generalidad suficiente.

❖ **¿Es estable?** A partir del momento en que terminemos la implementación y publiquemos su interface, vamos a asumir el compromiso de no cambiar esta última con el transcurso del tiempo. A lo sumo le agregaremos nuevas funcionalidades en las próximas versiones, pero las ya publicadas no cambiarán.

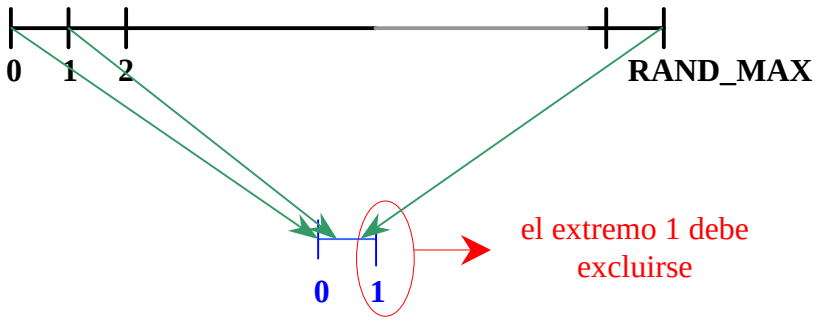
2.2 Su Implementación

Como nos vamos a valer de las funciones de la biblioteca estándar de C para armar la biblioteca random, y ésta garantiza que los números pseudoaleatorios que nos ofrece la función rand() están bien distribuidos en el intervalo [0, RAND_MAX], vamos a tener que analizar precisamente cómo hacer para que los números que nosotros devolvamos con las nuevas funcionalidades **sigan estando bien distribuidos** en los nuevos rangos propuestos.

2.2.1 Implementación de double randNormalize(void)

La función de la biblioteca estándar rand() asegura que los números están bien distribuidos en el intervalo [0, RAND_MAX], pero nosotros precisamos "trasladar" el rango al intervalo [0, 1), y además transformar a cada entero en un double.

La solución es hallar una función de mapeo, que gráficamente podría representarse así:



Por lo tanto el número double buscado podría ser obtenido como un cociente entre dos números, tal que el numerador sea menor que el denominador:

rand() / ((double) RAND_MAX + 1)

Aclaraciones

❖ Notar la necesidad de castear el denominador de dicha expresión para que el resultado no sea siempre cero, y el casteo de sólo RAND_MAX y no la suma de RAND_MAX con 1, porque caso contrario se estaría obteniendo un número negativo ya que RAND_MAX + 1 está fuera del rango representable de los enteros.

❖ La fórmula (double) rand() / RAND_MAX no sirve porque permite obtener el número 1 que debía excluirse

❖ La fórmula (double) (rand() – 1) / RAND_MAX tampoco sirve porque nos permite obtener números negativos.

Así es como nos decidimos por esta implementación. Obviamente el código estará en el archivo ***random.c*** (que el usuario no conocerá)

```
/*
**    Implementación para la biblioteca random
*/

#include <stdlib.h>
#include "random.h"

/*    Esta funcion devuelve un número double pseudoaleatorio
**    pertenciente al intervalo [0, 1)
*/

double
randNormalize(void)
{
      return rand() / ( (double) RAND_MAX + 1);
}
```

siempre incluir el prototipo de las mismas funciones que se están definiendo

2.2.2 *Implementación de int randInt(int izq, int der)*

Primero podríamos pensar simplemente en trasladar el intervalo [0,RAND_MAX] al intervalo [izq, der] pedido.

Una fórmula que podría permitirnos obtener algo así sería:

rand() % (der – izq + 1) + izq

Esta no parece ser una buena distribución para los números pseudoaleatorios por varios motivos:

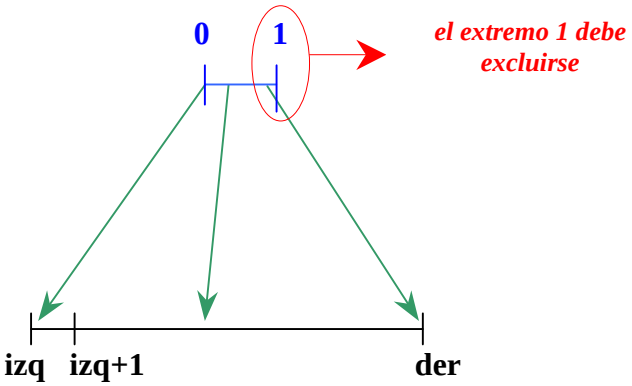
a) Si el intervalo fuera [0, 1], estaríamos haciendo rand() % 2, o sea los pares mapearían al 0 y los impares al 1. Dentro del rango [0, RAND_MAX] la función rand() nos devuelve números bien distribuidos, por ejemplo: 23, 37, 1371, 1999, 9253, 19, 2, etc. lo cual con nuestra función estaría mapeando a los números 1, 1, 1, 1, 1, 1, 0 que no parecen estar bien distribuidos.

b) Si el intervalo fuera [0, 30000] y el tamaño del int en esta arquitectura fuera 2 bytes (o sea el máximo entero fuera 32767), habría números que se generarían con mayor frecuencia, pues con la fórmula rand() % 30001 tendríamos que los números en rango 30001 al 32767 volverían a ser mapeados en los primeros.

Una **segunda aproximación**, nos permitiría elegir mejor la fórmula para obtener números en el rango pedido, aprovechando el hecho de que la función que ya implementamos para randNormalize() aseguraba una buena distribución.

Faltaría escalar los números pseudoaleatorios en el rango double [0, 1) al tamaño del intervalo solicitado, truncarlos para que sean enteros y luego trasladarlos para que el menor de ellos coincida con el extremo izquierdo.

Esto es, aprovechando el nivel de abstracción que nos ofrece la función randNormalize(), expandimos los aleatorios para que caigan en el rango del tamaño del ancho del intervalo y si hace falta los desplazamos (el extremo inferior no tiene por qué empezar en cero)



La fórmula buscada podría ser

(int) (randNormalize() * (der – izq + 1)) + izq

por si hay que desfasarla
(izq no tiene porque ser cero)

```
.....

/*    Esta funcion devuelve un número entero pseudoaleatorio
**    pertenciente al intervalo entero [izq, der]
*/

int
randInt( int izq, int der)
{
      return (int) (randNormalize() * (der – izq + 1) ) + izq;
}
```

Comparando Métodos ...

A modo de verificación hemos probado ambas fórmulas para calcular ***randInt*** en tres casos. Al método que utiliza el resto de la división lo llamamos ***R%*** y al que utiliza *randNormalize* (el propuesto para nuestra biblioteca) lo denominamos ***R****.

En todos los casos generamos **N** números aleatorios dentro del rango entero **[a, b]**, donde lo esperado teóricamente es una distribución uniforme de **N/(b-a+1)** apariciones para cada número entero del intervalo. Obviamente la distribución que se obtiene experimentalmente no es pareja ya que algunos números logran más apariciones a expensas del resto.

Para poder comparar ambos métodos, escribimos un programa que genera **N** números aleatorios en el rango **[a, b]** y calcula la desviación cuadrática media de la disribución. Luego corrimos el programa 10 veces para cada caso y calculamos la desviación cuadrática media del promedio. A continuación mostramos los resultados experimentales:

Caso 1)

```
Intervalo = [0, 100]
N= 1000

sp(R%) =  0.211 E-7
sp(R*) =  0.211 E-7
```

Caso 2)

```
Intervalo = [0, 1]
N= 100

sp(R%) = 0.242 E-4
sp(R*) = 0.162 E-4
```

Caso 3)

```
Intervalo = [0, 1]
N= 20

sp(R%) = 0.0012
sp(R*) = 0.0008
```

En todos los casos se evidencia que la fórmula usada en nuestra biblioteca produce menos desviación respecto de la distribución uniforme que la típica distribución a través del resto de la división entera, aunque la diferencia se hace significativa para pocos tiros.

2.2.3 Implementación de *double randReal(double izq, double der)*

La función es parecida a la anterior pero sin tener que pasar el resultado a un entero. Además no hay que calcular la cantidad de números en el intervalo pedido porque ahora el intervalo es real y hay por lo tanto infinitos números en él. Por lo tanto la fórmula podría ser:

randNormalize() * (der – izq) + izq

Notar que el usuario de esta biblioteca no debería, por estar operando con números reales, preguntar por la aparición de "double der", como una igualdad, sino por la aproximación del mismo.

```
.....

/*    Esta funcion devuelve un número double pseudoaleatorio
**    pertenciente al intervalo double [izq, der]
*/
double
randReal( double izq, double der)
{
      return randNormalize() * (der – izq ) + izq;
}
```

2.2.4 Implementación de *void randomize(void)*

Lo único que pretende realizar esta función es ocultar cómo se seta la nueva semilla, y para ello utilizaremos como semilla la hora del sistema.

```
#include <time.h>
#include <stdlib.h>
#include "random.h"
.........

/*
**    Esta funcion devuelve setea la semilla con la hora del sistema
*/
void
randomize(void)
{
      srand ( (int) time(NULL) );
}
```

agregamos el prototipo de la función time que usaremos para el seteo de la semilla

2.3 Su Prueba antes de la distribución

Antes de pretender distribuir una biblioteca, por supuesto se le aplican los tests de caja negra y blanca correspondiente.

Resultaría muy conveniente complementar su documentación con algún sencillo programa que muestre su uso, para evitar llamados de usuarios que, a pesar de haber recibido una buena interface, duden sobre cómo utilizar las funcionalidades que brindan la biblioteca.

En este caso probaremos la biblioteca con una pequeña aplicación que simula el juego de ruleta de casino, indicando para cada tiro de la bola el número, su columna, paridad, y docena ganadora.

```
/*
**    Juego del Casino. Se usa para probar la biblioteca random
*/

#include <stdio.h>
#include “random.h”
#define BORRA_BUFFER  while (getchar() ¡= ‘\n’)

void
presentacion(void)
{
  printf("\nBienvenido al casino\n");
  printf("1- Sacar numero\n");
  printf("2- Salir\n");
}


int
columna( int numero )
{
  return  (--numero  % 3 + 1);
}


int
docena( int numero )
{
  return  (--numero / 12 + 1);
}


int
main( void )
{
   int numero;
   int opcion;

   randomize();
   presentacion();
```

```
do
{
      printf("Ingrese Opcion:");
      opcion= getchar();
      BORRA_BUFFER
      if ( !isspace(opcion) && opcion != '1' && opcion != '2')
         printf("Opcion Invalida\n");
      else
         if (opcion=='1')
         {
            numero= randInt(0, 36);
            printf("El numero extraido es %d\n", numero);
            if (numero != 0 )
            {
               printf("Columna: %d\n", columna(numero));
               printf("Docena: %d\n", docena(numero));
               printf("Paridad: %s\n", (numero%2)? "Impar":"Par");
            }
         else
            printf("CERO: Oooohhhhh\n");
         }
}while (opcion != '2');


   return 0;
}
```

Aclaración:

Recordar cómo se generan bibliotecas en UNIX y en Borland C, ya explicado en los tutoriales CompilarBajoLinux y CompilarEnBorlandC, presentes en la red y en fotocopiadora.

3. Más Detalles sobre el Armado de una Biblioteca

Por último vamos a agregar dos aclaraciones que potenciarán el armado de bibliotecas y evitarán problemas cuando los usuarios deban linkeditarlas con sus códigos.

3.1 Uso de la palabra static para ocultar la definición de una función

Habíamos visto la visibilidad que podía dársele a una variable cuando se quería o no que fuera usada desde otro módulo en la cual se la definió.

La palabra ***static*** también puede utilizarse para hacer que la definición de una función no sea vista por otros módulos. Esto es muy importante especialmente en el armado de bibliotecas, porque cuando la misma es de cierta complejidad, su implementación requerirá de muchas más funciones que las que se publican en la interface. Esa subdivisión de las funcionalidades exportadas en funciones más pequeñas responde obviamente al objetivo de simplificar el código del implementador de la biblioteca. Sin embargo todas las funciones que estén definidas en un módulo son por omisión visibles en el resto, lo que significará que cualquier usuario de nuestras bibliotecas podría usarlas (pero como no están publicadas, se las usaría sin el correcto prototipo).

Para evitar conflictos, usaremos la palabra ***static*** para todas las declaraciones y definiciones de una función que no figuren en la interface de la biblioteca.

Importante:

Siempre definir como static todas las funciones de una biblioteca que no estén publicadas en la interface de la misma.

3.2 Definición de Alias: typedef

La idea importante de usar bibliotecas es la de generar un nivel de abstracción que permita simplificar el desarrollo de aplicaciones.

Análogamente la elección de nombres significativos para designar constantes, variables y funciones también tiene el objetivo claro de resaltar lo que ellas representan con sólo conocer su nombre.

El lenguaje C ofrece también la posibilidad de designar con un nombre significativo a cualquier tipo de datos.

Sintaxis

typedef tipoDeDatos alias;

Esto tiene un impacto importante en el aporte que realiza a la clarificación de los códigos. No es lo mismo ver una función que puede devolver algún tipo de error cuyo prototipo es:

int imprime();

que si el prototipo fuera

estado imprime();

Hasta ahora la única forma clara de evidenciar que la función no iba a devolver cualquier entero era definiendo una contante de enumeración:

enum estado {STATUS_ERR, STATUS_START, STATUS_OK, STATUS_END};

a partir de esto las funciones, parámetros y variables podían devolver el tipo ***enum estado.*** En nuestro ejemplo anterior tendríamos:

enum estado imprime();

La designación de un nuevo nombre para el tipo ***enum estado,*** lo vuelve sumamente práctico y claro.

Importante

Cuando usamos ***typedef*** **no estamos inventando un nuevo tipo de datos**, sino **un nuevo nombre para un tipo de datos existente**, o sea que simplemente estamos dando un alias más fácil de recordar o escribir.

En el ejemplo anterior, deberíamos haber hecho:

```
enum estado {STATUS_ERR, STATUS_START, STATUS_OK, STATUS_END};

typedef enum estado error;

error imprime();
```

Importante

En el caso de vislumbrar la necesidad de darle un alias a un tipo de datos que será devuelto o que recibirá en sus parámetros algunas de las funciones que sean exportadas en la interface de una biblioteca, dicho typedef deberá estar en el archivo de encabezamiento (o sea también debe ser exportado).

Arreglos - Parte I

Introducción

Hasta el momento nos centramos en las técnicas de desarrollo de algoritmos por medio de estructuras de control. Discutimos la forma de aumentar la claridad de los algoritmos por medio de funciones y macros. Sin embargo los datos que se manipularon eran sencillos. Existe la posibilidad de definir estructuras de datos que permitan representar a un conjunto de datos como una única entidad. Las estructuras de datos también ayudan a darle claridad a los programas.

En este documento se describe la estructura de datos llamada ***array***.

1. Arreglo (Array)

Un arreglo es una colección de datos con dos características fundamentales:

- **Homogeneidad**. Todas sus componentes **son del mismo tipo**. Así es como se puede tener un arreglo de enteros, o de chars pero no de ambos mezclados.
- **Orden**. Cada una de sus componentes tienen un lugar dentro del mismo y pueden accederse directamente con sólo referirse a dicho lugar. Esto no está diciendo que las componentes del arreglo estén ordenadas de acuerdo a sus contenido, sino que el orden del que se habla se refiere al lugar que ocupa una componente dentro de la estructura.

1.1 Declaración

Sintaxis de Declaración de una Variable de tipo Arreglo

/* sólo declaración */

```
tipo    nombreArreglo[cantidad];
```

cantidad debe ser una constante (nunca una variable) que indique hasta cuántas componentes contendrá el arreglo

/* declaración e inicialización */

```
tipo    nombreArreglo[cantidad] = {valor 0,
                                   valor1,
                                   ...
                                   valorCantidad-1
                                  };
```

o bien

```
tipo    nombreArreglo[] = {valor 0,
                           valor1,
                           ...
                           valorCantidad-1
                           };
```

Aclaración

Notar que en la inicialización se pone en evidencia el orden del que hablamos antes. En la estructura de arreglo se debe indicar en qué lugar exacto se desea guardar determinada componente. Lo mismo ocurrirá en la asignación.

Importante

Cuando no se inicializa, la definición de variable debe indicar explícitamente hasta cuántas componentes podrá albergar. Esta información es usada por el compilador para guardar espacio para dicha variable.

El lenguaje C exige que las variables de tipo arreglo sean declaradas antes de ser usadas, igual que cualquier otra variable. Pero, a diferencia de los tipos de datos simples, solicita que se le informe la cantidad máxima de elementos que almacenará.

1.2 Representación Interna

Para una variable de tipo arreglo, se reserva espacio para que pueda guardar **cada una de sus componentes en forma contigua**.

Una variable de tipo arreglo se guardará el Data Segment, BSS Segment o Stack (se aplican los mismo conceptos discutidos en la clase de Gestión de Memoria), según su clase de almacenamiento.

Si se tiene

tipo nombreArreglo[cantidad];

Suponiendo que a la primera componente del mismo se le asigne el lugar de memoria $1000, a partir de allí se asignarán los lugares para cada una de sus componentes. La dirección de la primera componente se denomina **dirección base (*base address*)**.

A la segunda componente se le asignará el lugar de memoria $1000 + sizeof(tipo), a la tercera componente el lugar de memoria $1000 + 2 * sizeof(tipo), y obviamente al última componente $1000 + (cantidad – 1) * sizeof(tipo).

Ejemplo 1:

Si se tiene una arquitectura de 16 bits, con

int vecNum[] = { 5, 7, 9, 10, 2, 2, 5, 8, 9, 3 }

Si la primera componente tiene asignada la dirección de memoria $1000, entonces el arreglo podría visualizarse en memoria:

$1000	$1002	$1004	$1006	$1008	$100A	$ 100C	$100E
5	**7**	**9**	**10**	**2**	**2**	**5**	**8**
9	**3**						

Ejemplo 2:

Si se tiene una arquitectura de 32 bits, con

int vecNum[] = { 5, 7, 9, 10, 2, 2, 5, 8, 9, 3 }

Si la primera componente tiene asignada la dirección de memoria $1000, entonces el arreglo podría visualizarse en memoria:

$1000	$1004	$1008	$ 100C
5	**7**	**79**	**10**
2	**2**	**5**	**8**
9	**3**		

Ejemplo 3:

Si se intenta inicializar un arreglo con más elementos que los declarados

int datos[5] = { 10, 20, 30, 40, 50, 60 };

se obtiene un error en tiempo de compilación.

Ejemplo 4:

Si se intenta inicializar un arreglo con menos elementos que los declarados

```
int   datos[ 5 ] = {  10, 20 };
```

se garantiza que los que faltan inicializar explícitamente se colocarán en cero, pero éstos deben ser los últimos de la lista (no se pueden saltear elementos sin colocar elementos entre las comas)

1.3 Acceso

La variable de tipo arreglo designa con un único nombre a **todo un conjunto de componentes homogéneas individuales**.

Dicho nombre (el de la variable de tipo arreglo) puede ser usado para referenciar **genéricamente** a todas sus componente o para referenciar a **una de sus componentes** en particular.

Para poder acceder a una de sus componentes, se aprovecha el hecho de que éstas tengan un orden (el lugar que ocupan relativo al comienzo del arreglo). La sintaxis de acceso a una componente en particular es:

Sintaxis de Acceso a la componente que ocupa el lugar i

```
nombreArreglo[ i ]
```

Muy Muy Importante

La primera componente de un arreglo en el lenguaje C **ocupa el lugar 0 relativo al comienzo del mismo** (no 1).

Esto significa que si un arreglo se declaró con N componentes, la primera componente que se puede referenciar es la que ocupa el lugar relativo 0, y la última válida es la que ocupa el lugar relativo N-1.

Indice

Cada componente dentro del arreglo es identificada por un número llamado **índice**.

arreglo[i] → **i es el índice**

Como todas la componentes de un arreglo son del mismo tipo (homogeneidad), el acceso a determinada componente puede participar de cualquier expresión donde participaría una variable suelta que hubiera sido declarada de dicho tipo.

Ejemplo

Se mostrará el uso de la variable temperatura para almacenar la temperatura promedio de los 5 continentes.

```
float    temperatura[5];

temperatura[0]= 12.3;
temperatura[1]= 25;
temperatura[2]= 22.4;
temperatura[3]= 16;

temperatura[4]= 28;

printf(" Temperatura promedio en Africa=%g\n", temperatura[4]);
....
```

máxima cantidad a usar 5

la última posición válida a referenciar es 1 menos que la cantidad declarada

Ejemplo

Otra forma hubiera sido:

```
typedef  enum { AMERICA, EUROPA, ASIA, OCEANIA,  AFRICA } continente;
float    temperatura[5];

temperatura[AMERICA]= 12.3;
temperatura[EUROPA]= 25;
temperatura[ASIA]= 22.4;
temperatura[OCEANIA]= 16;
temperatura[AFRICA]= 28;

printf(" Temperatura promedio en Africa=%g\n", temperatura[AFRICA]);
....
```

máxima cantidad a usar 5

Aclaración

Cuando se quiere utilizar en alguna expresión la cantidad de elementos de un arreglo se puede utilizar la siguiente expresión

sizeof(nombreArreglo) / sizeof (tipo)

o bien

sizeof(nombreArreglo) / sizeof (nombreArreglo[0])

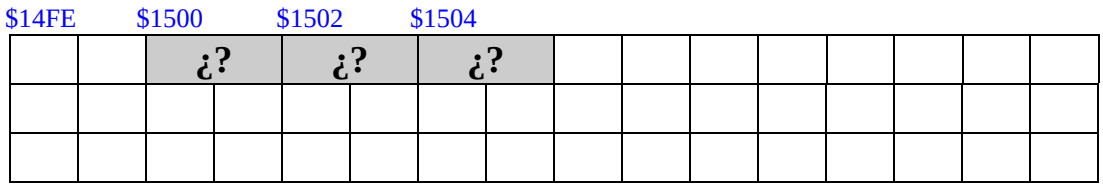
Ejercicio:

Representar cómo iría cambiando la memoria asignada a la variable ***número***, en tiempo de ejecución, suponiendo que comienza almacenando su primera componente en la dirección $1500 y el tamaño de entero en esta arquitectura es de 2 bytes.

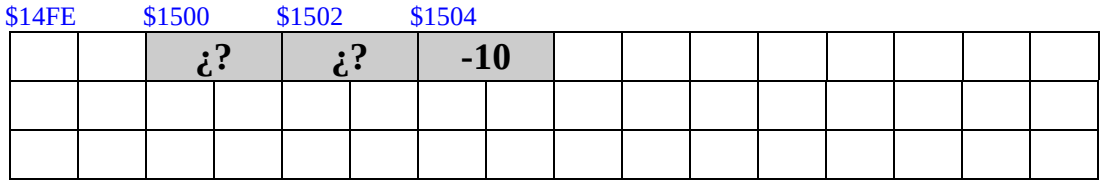
```
#include <stdio.h>

int
main( void )
{
      int numeros[3];

      numeros[2]= -10;
      numeros[0]= numeros[1]= numeros[2] * 3;

      return 0;
}
```

Rta:

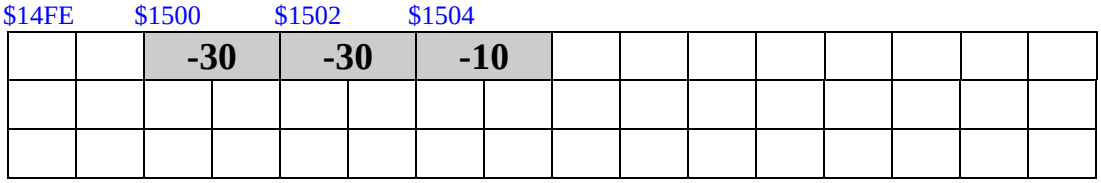
a) antes de la primera asignación se tiene



b) después de numeros[2]= -10;



c) después de numeros[0]= numeros[1]= numeros[2] * 3;



Cuando el compilador encuentra una referencia a una componente individual del un arreglo lo traduce internamente calculando el **desplazamiento *(offset)*** de memoria del mismo desde su dirección base (la del comienzo del arreglo).

Ese cálculo es muy fácil de hacer, ya que no es otra cosa que multiplicar el índice indicado por el sizeof(tipo) del mismo:

dirección base + índice * sizeof(tipo)

desplazamiento

Ejemplo:

De acuerdo al ejemplo anterior

Dirección de **numeros[2] = $1500 + 2 * 2 = $1504**

Muy importante

El nombre de variable de un arreglo (que designa el conjunto de componentes homogéneas**) no es un l-value**.

Una componente en particular de un arreglo de tipo simple **sí es un l-value** (pero no es una variable).

¿Qué ocurre cuando un usuario, quizás por error, intenta referenciar una componente que está fuera de los límites del arreglo?

Desgraciadamente, los compiladores no advierten dicho error. Sólo se remiten a traducir la expresión nombreArreglo[i] en el offset correspondiente, produciendo serios errores en tiempo de ejecución que oscilan desde una acceso a una dirección de memoria en la que en realidad no hay guardada una componente del arreglo, hasta la asignación de un valor en dicho lugar de memoria, produciendo lo que se conoce como "pisar memoria" ya que se está accediendo a una zona aledaña errónea.

Ejercicio 1

Indicar en cada uno de los siguientes ejemplos qué se está haciendo

a)

```
float reales[ ] = { 2.5, -3.7, -1, 0.34 };

for ( i = 0 ; i < sizeof( reales )/sizeof(reales[0]); i++ )
      printf("reales[%d] = %g\n",  i,  reales[i] );
```

Rta: Se imprimen en la salida estándar las componentes del arreglo de tipo float.

b)

```
#define DIM 10;
int  pares[DIM], i;

for (i = 0;  i <= DIM-1;  ++i )
       pares[i]= 2 * i;
```

Rta: Se llena al arreglo pares con los pares 0, 2, 4, 6, 8, 10, 12, 14 ,16 y 18

c)

```
int
main(void)
{
      int cantidad= 10;
      float valores[ cantidad ];

      while (cantidad-- )
            valores[cantidad]=  cantidad;

      return 0;
}
```

Rta: Intenta crear un arreglo de tamaño dado por una variable. No compilará porque el tamaño máximo debe estar dado por una constante. Atención: estamos hablando de forma genérica para los compiladores, **podríamos encontrar algún compilador en particular que lo admita, pero nuestro código no será multiplataforma!!!**. No generalizar casos particulares.

d)

```
int
main( void )
{
      int rec;
      long valores[10];

      for(rec= 0;rec<=sizeof(valores)/sizeof(valores[0]); rec++ )
            valores[ rec ] = 0;
.....
}
```

Rta: Se está intentando blanquear cada componente del arreglo, pero como se está "pisando memoria" debido a la asignación cuando rec es 10 (recordar que el último índice válido es uno menos que el que aparece en la declaración), **el resultado es impredecible.**

e)

```
int
main( void )
{
      int c, rec letra[26];

      for  ( rec = 0; rec<26; rec++)
            letra[rec] = 0;

      while ( (c= toupper(getchar()) ) != EOF)
            if (  c  >= 'A' &&   c  <= 'Z' )
                  letra[  c – 'A' ]++;

      for  ( rec = 0; rec<26; rec++)
            printf("cantidad de %c= %d\n", rec+'A', letra[rec]);

      return 0;
}
```

Rta: Contabiliza la cantidad de letras del alfabeto inglés se encuentran presentes en la entrada estándar, sin diferenciar entre mayúsculas y minúsculas. Notar la necesidad de **desplazar** el intervalo lógico ['A','Z'] al [0,25] ya que el lenguaje C comienza con índice 0.

Muy Muy Importante

El tamaño que se indica en la declaración de un arreglo, es sólo la cantidad máxima de componentes que se calcula se van a poder querer utilizar.

Sin embargo, normalmente dicho tamaño es un tope, lo que significa que se usan muchos menos. En estos casos **no tiene ningún sentido** recorrer todos los elementos para inicializarlos, o lo que es más absurdo mostrar su contenido. Por eso cuando no necesariamente se precisan usar todos los elementos de un arreglo, **siempre** se trabaja con una variable entera asociada en la lógica de un programa que guarda **el valor exacto o verdadera dimensión** de los elementos del mismo (ver próximo ejemplo).

Ejercicio 2

Escribir un programa que lea valores de la entrada estándar que representan las notas de los alumnos e indique la desviación estándar. Los valores ya están validados, y vienen entre 0 y 10, pero un valor negativo indicará el fin del ingreso de datos.

Rta:

```
#include <stdio.h>
#include <math.h>
#include “getnum.h”

int
main( void )
{
      float notas[120];
      int dimension= 0;
      float promedio= 0;
      float total= 0;

      while (dimension < 120 && (notas[dimension] = getfloat(“”))>=0 )
            promedio+= notas[ dimesion++ ];

      if ( dimension > 0 )
      {
            int cant= dimension;
            promedio /= dimension;
            while ( dimension-- )
                  total += pow (  promedio – notas[ dimension ], 2 );
            printf(“La desviacion se calcula en %g\n”,
                                                sqrt(total)/ cant);
      }
      else
            printf(“No se han ingresado notas todavia\n”);

      return 0;
}
```

Notas:

a) Notar la importancia de manejar la variable **dimension,** para validar que el ingreso de datos esté dentro de los límites del arreglo, para recorrerlo en el calculo del promedio y de la desviación sin cometer el grave error de involucrar en dicho cálculo valores que no forman parte de las notas.

b) Advertir que los operandos en la línea de control del ciclo

while (dimension < 120 && (notas[dimension] = getfloat(“”)) > 0)

no pueden aparecer en cualquier orden, ya que hay que asegurarse de que dimension es realmente menor estricto que la dimensión tope, antes de asignarla como componente del arreglo, caso contrario se estaría “pisando memoria”, aunque después no se ejecutara el cuerpo del while. En este caso hay que aprovechar el hecho de que **el operador && es lazy y asegura evaluarse de izquierda a derecha.**

2. Arreglos Multidimensionales

En el lenguaje C los elementos de un arreglo pueden ser de cualquier tipo, inclusive podrían ser otros arreglos. De esta forma un arreglo puede tener: un subíndice si es unidimensional, dos subíndices si es bidimensional, etc.

Aunque los bidimensionales son los más usados, ANSI C asegura un mínimo de 12 subíndices.

2.1 Arreglos Bidimensionales (matriz o tabla)

Si no existiera forma posible en el lenguaje de declarar y manipular arreglos multidimensionales, deberíamos ingeniarnos realizando cálculos matemáticos para acceder a las filas y columnas de los mismos.

Ejemplo:

Para una arreglo de 3 filas por 2 columnas podríamos hacer

```
#define FILAS       3
#define COLUMNAS    2

int matriz[FILAS*COLUMNAS]= {5, 6, -1, 4, 2, 1};
```

referenciando el elemento (i, j) a través de **matriz[i * columnas + j].**

Por ejemplo, si el primer elemento cayera en la dirección $FF00 y la arquitectura de computadora fuera de 16 bits, tendríamos:

$FF00	$FF02	$FF04	$FF06	$FF08	$FF0A	$ FF0C	.....
5	**6**	**-1**	**4**	**2**	**1**		

Afortunadamente el lenguaje C nos permite definir y manipular arreglos multidimensionales de una forma mucho más sencilla y clara.

2.1.2 Declaración

Sintaxis de Declaración de una Variable de tipo Arreglo Bidimensional

```
/* sólo declaración */

        tipo    nombreArreglo [cantFilas] [cantColumnas];


/* declaración e inicialización */

        tipo    nombreArreglo[cantFilas] [cantColumnas]=
        {  { valor1_1, valor1_2, ........          valor1_cantColumnas-1},
           { valor2_1, valor2_2, ........          valor2_cantColumnas-1},
                .....
           { valorCantFilas_1,  ........          valorCantFilas-1_cantColumnas-1}
        };

o bien

        tipo    nombreArreglo[ ] [ cantColumnas]=
        {  { valor1_1, valor1_2, ........          valor1_cantColumnas-1},
           { valor2_1, valor2_2, ........          valor2_cantColumnas-1},
                .....
           { valorCantFilas_1,  ........          valorCantFilas-1_cantColumnas-1}
        };
```

Importante

El compilador precisa conocer la cantidad de elementos en las columnas para poder calcular la fórmula que mapee la referencia a una componente en la dirección correspondiente.

La componente [i] [j] se encontrará en la dirección:

dirección base + (cantColumnas * i + j) * sizeof(tipo)

2.1.2 Representación Interna

C maneja un arreglo bidimensional como un arreglo cuyas componentes son arreglos, por lo tanto la representación interna es una generalización de lo dicho previamente sobre este tema.

Ejemplo:

Si se tiene una arquitectura de 32 bits, con

int numeros[ ] [4] = { { 5, 7, 9, 4 }, {10}, {2, 2}};

El arreglo numeros está formado por 3 componentes. Cada una de ellas es a su vez un arreglo de 4 componentes.

Si la primera componente de la variable numeros tiene asignada la dirección de memoria FD90, implica que numeros[0] estará en dicha dirección, pero como sizeof(numeros[0]) es de 4 * sizeof(int), o sea 16 bytes, entonces numeros[1] se encontrará en la dirección FDA0, y numeros[2] en la dirección FDB0.

$FD90	$FD94	$FD98	$FD9C
5	**7**	**9**	**4**
10	**0**	**0**	**0**
2	**2**	**0**	**0**

Notar que la constante que indica la cantidad máxima de columnas es necesaria en su definición, sólo puede omitirse la cantidad de filas en el caso de inicialización.

2.1.3 Acceso

La variable de tipo arreglo designa con un único nombre a **todo un conjunto de componentes homogéneas individuales**.

En el caso de las matrices o arreglos bidimensionales, una componente sería todo un vector. En el ejemplo anterior las componentes son numeros[0], numeros[1] y numeros[2].

Pero como, a su vez, los arreglos que lo componen tienen componentes, también se puede querer acceder a cada una de ellas, esto es a un elemento indicado por su fila y su columna:

Sintaxis de Acceso a la componente que ocupa el lugar i, j

nombreArreglo[i] [j]

Ejercicio 3:

Completar el siguiente programa para que imprima todos los elementos de la matriz numeros, colocando cada fila en una línea distinta.

```
#define FILAS           10
#define COLUMNAS        5

int
main( void )
{


        int     numeros[ FILAS] [COLUMNAS];
        ..... /* en esta zona se le asignan valores a cada una de sus componentes */

        ....
        return 0;
}
```

Rta:

```
#define FILAS           10
#define COLUMNAS        5

int
main( void )
{
        int     numeros[ FILAS] [COLUMNAS];
        int     fila, col;

        ..... /* en esta zona se le asignan valores a cada una de sus componentes */

        for ( fila= 0; fila < FILAS;  fila++)
        {
                for ( col= 0;  col < COLUMNAS;  col++)
                        printf("num[%d,%d]=%d\t", fila, col, numeros[fila][col]);
                printf("\n");
        }
        return 0;
}
```

Ejercicio 4:

Completar el siguiente programa para que calcule la suma de los elementos de la matriz numeros. La cantidad real de filas usadas está almacenada en la variable dimFilas, y la cantidad real de columnas usadas está almacenada en la variable dimCol.

```
#define FILAS           100
#define COLUMNAS        500

int
main( void )
{
        int     dimFilas= 0, dimCol= 0;
        float   numeros[ FILAS] [COLUMNAS];

        ..... /* en esta zona se le asignan valores a cada una de sus componentes */

        return 0;
}
```

Rta:

```
#define FILAS           100
#define COLUMNAS        500

int
main( void )
{
        int     dimFilas= 0, dimCol= 0;
        float   numeros[ FILAS] [COLUMNAS];
        float   total= 0;

        ..... /* en esta zona se le asignan valores a cada una de sus componentes */

        for ( i =  0; i  < dimFilas;  i++)
                for ( j = 0;  j < dimCol;  j++)
                        total+= numeros[ i] [ j];

        printf("Total acumulado=%d\n", total);
        return 0;
}
```

Ejercicio 5:

Indicar cuál sería la salida del siguiente programa, en una arquitectura de 32 bits.

```
#include <stdio.h>

#define FILAS           5
#define COLUMNAS        10

int
main( void )
{
        int     numeros[ FILAS] [COLUMNAS];
        ...
        printf("%d\n", sizeof( numeros ) );
        printf("%d\n", sizeof( numeros[0] ) );
        printf("%d\n", sizeof( numeros[0][0] ) );

        return 0;
}
```

Rta:

La salida será

200
40
4

Esto muestra que el compilador trata a cada uno de ellos de forma diferente. Por ejemplo, **numeros[0]** es un arreglo de diferente tamaño que **numeros** a secas.

Hay que tener cuidado cuando se los utiliza en expresiones porque cada uno de ellos tiene un significado diferente.

Arreglos - Parte II

Introducción

Las variables de tipo arreglo también pueden participar del proceso de pasaje de parámetros a funciones. Esto nos da la potencia de poder pasar **todo un conjunto de componentes** al subprograma que las quiera manejar.

En este documento se analizará el proceso del pasaje de parámetros, el cual tiene mucha relación con su representación interna.

1. Pasaje de Parámetros en Arreglo

El pasaje de parámetros en C es siempre por valor. Cuando se invoca una función, se crea un *stack frame*: el **contenido del parámetro actual se copia en el parámetro formal correspondiente.**

1.1 Pasaje de una componente de tipo simple (char, int, float, double)

Una componente de un arreglo que es de tipo simple, podría ser pasada como parámetro a un arreglo como ya lo conocemos, ya que es un *l-value y jugaría el rol de una variable*.

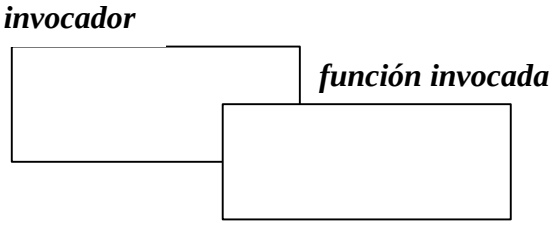
Ejemplo

El siguiente programa lee un mensaje desde la entrada estándar y lo almacena en un vector de caracteres. Luego de realizar ciertos procesos, utiliza la función ***encripta*** para codificar cada letra del mensaje con la siguiente regla: cada letra del alfabeto inglés se cambia por otra (de acuerdo a cierta convención), el resto de los caracteres quedan igual. La función escribe cada letra en la salida estándar, pero la original queda intacta.

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

#define TOPE     5
```

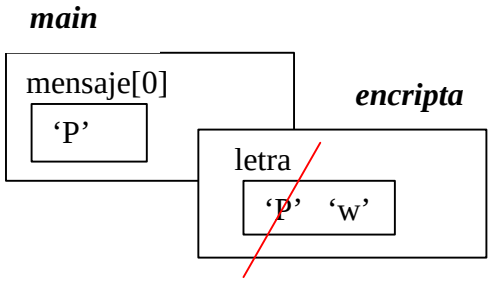
```
char
encripta( char letra )
{

      /* Tabla para la conversión de cada letra mayuscula */
      static char transforma[26]= { '#', '!', '?', '&', '.',
                                    '/', '{', ']', '(', '=',
                                    '[', '}', ')', 'c', 'e',
                                    'w', 'j', 'x', 'a', 'i',
                                    'm', 'o', 'y', 'h',
                                    '\\', '<'};
      letra = toupper(letra);

      if ( isalpha(letra) )
            letra= transforma[letra - 'A' ];

      return letra;
}



int
main( void )
{
      char mensaje[ TOPE ];
      int dimension = 0;
      int leido;
      int rec;

      while(dimension < TOPE && (leido = getchar() )!= EOF)
            mensaje[dimension++]= leido;


      ........................................


      for ( rec=0 ; rec<dimension ; rec++)
            putchar( encripta( mensaje[ rec ]) );


      ........................................

      return 0;

}
```

Cada vez que se realiza la invocación de un función se crea un ***stack frame*** correspondiente, que como ya habíamos indicado en documentos anteriores, representamos:

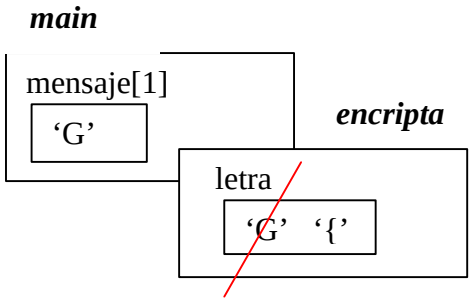


Supongamos que en el ejemplo anterior el mensaje leído desde la entrada estándar es: **PGM1**. Los *stack frames* que se crean en cada invocación resultarían:

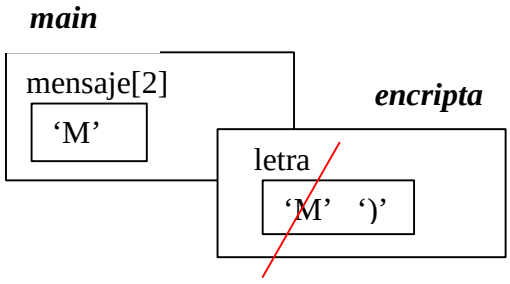
1ª invocación:



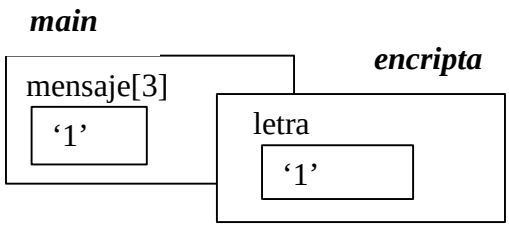
2ª invocación:



3ª invocación:



4ª invocación:



Como se puede observar en cada una de las invocaciones, el valor del parámetro actual NUNCA cambia, ya que el pasaje de parámetros es realizado por valor, o sea cuando se crea el *stack frame*, se crea "una nueva variable" llamada parámetro formal, y su contenido se **inicializa con el contenido** del parámetro actual correspondiente. De esta forma, como dicha variable tiene **otra dirección de memoria**, y su contenido es una **copia del original**, cualquier cambio realizado sobre el parámetro formal NO cambia el contenido del parámetro actual correspondiente.

Más detalladamente, podríamos mostrar las direcciones de memoria y los contenidos de cada una de dichas variables (los parámetros actuales y formales correspondientes).

El ejemplo lo ejecutamos en Linux, obteniendo:

invocación	parámetro actual		parámetro formal	
	dirección	contenido	dirección	Contenido
1ª	0xBFFFFCF8	'P'	0xBFFFFCDF	~~'P'~~ 'w'
2ª	~~0xBFFFFCF9~~	'G'	~~0xBFFFFCDF~~	~~'G'~~ '{'
3ª	~~0xBFFFFCFA~~	'M'	~~0xBFFFFCDF~~	~~'M'~~ ')'
4ª	0xBFFFFCFB	'1'	0xBFFFFCDF	'1'

Las direcciones de los parámetros actuales y formales NUNCA coinciden, por eso un cambio en el contenido uno no afecta el contenido del otro

1.2 Pasaje de un arreglo

La variable de tipo arreglo contiene la dirección de su primer elemento.

Cuando una variable de tipo arreglo se pasa como argumento a una función, se crea un *stack frame* y se copia como siempre el contenido del parámetro actual en el parámetro formal correspondiente.

Como el pasaje de parámetros en C es por valor, cualquier cambio sobre el parámetro formal no afecta el contenido del parámetro actual correspondiente, pero como lo que se pasó es la dirección de la primera componente, gracias a esa dirección se pueden producir cambios en las componentes del arreglo (no en la dirección del mismo).

Ejemplo

El siguiente programa utiliza la función ***leeArreglo*** para leer desde la entrada estándar números positivos de tipo float y guardarlos como componentes de un arreglo, devolviendo además la cantidad de componentes leídas.

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

#define   TOPE    5


int
leeArregloPositivo( float arreglo[] )
{
      int dim= 0;
      float leido;

      while ( dim < TOPE &&  ( leido= getfloat("") ) > 0 )
            arreglo[dim++]= leido;
      return dim;
}


int
main( void )
{
      int cantidad;
      float nrosPositivos[TOPE];

      printf("Ingrese numeros reales positivos\n");
      cantidad= leeArregloPositivo( nrosPositivos);
      while( cantidad )
            printf("%g\n", nrosPositivos[--cantidad]);
      return 0;
}
```

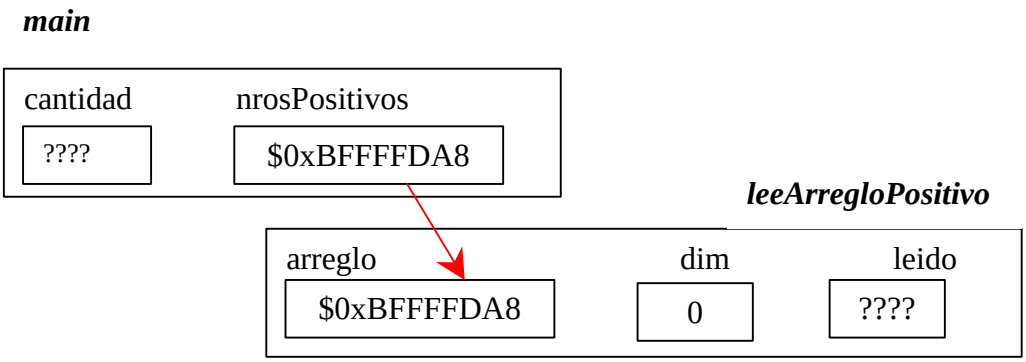
Es indistinto indicar o no la cantidad de componentes de un vector unidimensional. El compilador no chequea acceso fuera de los límites

Supongamos para el ejemplo anterior que los números leídos de la entrada estándar son **3.4**, **10** y **–2**. Para poder hacer el seguimiento con los stack frames ejecutamos el programa en Linux, y obtuvimos que al primer elemento del arreglo nrosPositivos se le asignó la dirección **$0xBFFFFDA8.**

Antes de la invocación de la **función leeArregloPositivos,** tenemos en el Stack:

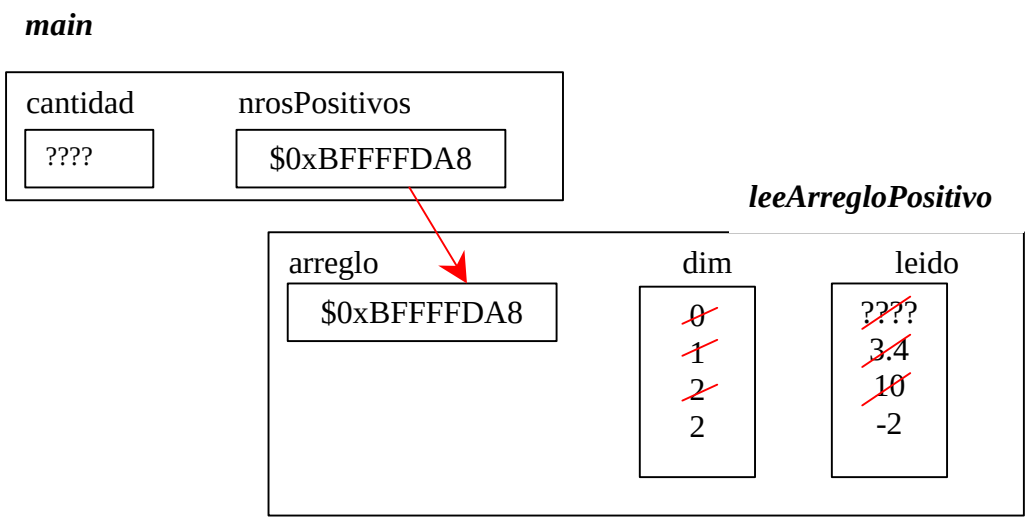
nrosPositivos[0]	$0xBFFFFDA8	**????**
nrosPositivos[1]	$0xBFFFFDAC	**????**
nrosPositivos[2]	$0xBFFFFDB0	**????**
nrosPositivos[3]	$0xBFFFFDB4	**????**
nrosPositivos[4]	$0xBFFFFDB8	**????**
cantidad	$0xBFFFFDBC	**????**

Al realizar la invocación tenemos que la variable **nrosPositivos,** por ser un arreglo, contiene la dirección del primer elemento del mismo. Al hacer que la misma sea una parámetro actual producirá que **su contenido sea copiado** en el parámetro formal **arreglo** que se corresponde con él y que fue creado a través del mecanismo de *stack frame*. Así es como no se copian en el ***stack*** cada una de sus componentes, sólo la dirección de la primera componente. Como los pasaje de parámetros en C son por valor, dicha dirección de la primera componente NO podrá cambiar, sin embargo la función podrá acceder, gracias a dicha dirección, a las componentes del arreglo y modificar sus valores:



Si se cambiara el contenido del parámetro formal **arreglo** dentro de la función **leeArreglosPositivos**, no se produciría ningún efecto en **main**, ya que el mismo fue pasado por valor. Sin embargo gracias a la dirección que se pasó a la función, si la misma cambia el contenido de la dirección de memoria , ese efecto se produce y va a repercutir cuando se acceda al arreglo.

A medida que se van ingresando valores desde la función **leeArreglosPositivos,** se tendrían los siguientes cambios de memoria:



Al tener la dirección base del arreglo, la función está accediendo a direcciones de memoria que no pertenecen a su *stack frame*:

arreglo[dim ++]= leido
ß
dirección base + dim * sizeof(float) = leido

$0xBFFFFDA8 + 0 * 4 = 0xBFFFFDA8 se guarda el 3.4
$0xBFFFFDA8 + 1 * 4 = 0xBFFFFDAC se guarda el 10

Finalmente la memoria se ve afectada de la siguiente forma:

leido	$0xBFFFFD94	~~?~~ ~~3.4~~ ~~10~~ -2
dim	$0xBFFFFD98	~~0~~ ~~1~~ 2
	$0xBFFFFD9C	**usado para**
	$0xBFFFFDA0	**retornar**
arreglo	$0xBFFFFDA4	**$0xBFFFFDA8**
nrosPositivos[0]	$0xBFFFFDA8	**3.4**
nrosPositivos[1]	$0xBFFFFDAC	**10**
nrosPositivos[2]	$0xBFFFFDB0	**????**
nrosPositivos[3]	$0xBFFFFDB4	**????**
nrosPositivos[4]	$0xBFFFFDB8	**????**
cantidad	$0xBFFFFDBC	**????**

stack frame para ***leeArregloPositivos***

stack frame para ***main***

Cuando la función **leeArregloPositivos** alcance su ***return***, su ***stack frame*** desaparecerá, pero los efectos producidos en las componentes del arreglo permanecerán. Si al pasar como parámetro un arreglo, en vez de copiar su dirección de comienzo se copiaran todos los elementos del mismo, sería imposible modificar sus contenidos.

Ejemplo

Supongamos ahora que queremos agregar al programa anterior una función que imprima cada una de las componentes del vector por la salida estándar. Dicha función sería:

```
void
imprimeArreglo( float arreglo[], int dim )
{
      int rec;

      for(rec= 0; rec < dim; rec++)
            printf("%g\n", arreglo[rec] );

      return;
}
```

Nótese que esta función debe recibir la dirección de la primera componente del arreglo, y también la dimensión real del mismo, para recorrerlo sólo hasta donde hay elementos.

Reflexión

El hecho de que al pasar en los parámetros arreglos a las funciones, sólo se copie la dirección de su primera componente ofrece ciertas ventajas y desventajas:

❖ **Ventaja**: la rapidez de la invocación de la función. No es lo mismo copiar cada elemento del arreglo que sólo la dirección de su primera componente. Debido a que las componentes de un arreglo se almacenan en memoria en forma contigua, teniendo la dirección de la primera podemos acceder a las demás.

❖ **Desventaja**: ofrece toda la potencia del cambio de una componente como efecto colateral. En el caso de funciones donde sólo se quiera leer el contenido de una componente (no cambiarlo), puede resultar riesgoso. Sintácticamente estaríamos ofreciendo el mismo prototipo cuando queremos modificar una componente del arreglo que cuando no queremos hacerlo. Para mejorar la semántica respecto de este punto y evitar algunos dolores de cabeza, ANSI C ofrece la posibilidad de agregar el calificador ***const*** a un arreglo para indicar que las componentes no serán modificadas. Si un código intenta modificar las componentes de un arreglo pasado como parámetro y calificado como ***const,*** entonces el compilador indicaría algún error.

Ejemplo

Rehacemos la función ***imprimeArreglo*** con el calificador *const:*

```
void
imprimeArreglo( const float arreglo[], int dim )
{
      int rec;

      for(rec= 0; rec < dim; rec++)
            printf("%g\n", arreglo[rec] );

      return;
}
```

Ejemplo

Si lo hubiéramos intentado usar con la función ***leeArregloPositivos*** hubiéramos obtenido un error de compilación del estilo:

"assignment of read-only location"

```
int
leeArregloPositivo( const  float arreglo[])
{
      int dim= 0;
      float leido;


      while ( dim < TOPE &&  ( leido= getfloat("") ) > 0 )
            arreglo[dim++]= leido;

      return dim;

}
```

No se puede modificar el contenido de una componente porque se usó el calificador const

1.2.1 Pasaje de un Arreglo Multidimensional a una Función

Conceptualmente ocurre lo mismo que se explicó en la sección 1.2.

Si se quiere manipular todas las componentes de tipo simple que forman el arreglo, se procede a invocar sólo con el nombre del arreglo, y en el parámetro formal correspondiente se copia la dirección de la primera componente.

Recordar que un arreglo multidimensional es **un vector de arreglos**, así es como la dimensión primera puede omitirse, ya que C no chequea si se encuentra o no dentro de los límites del arreglo, pero las demás dimensiones deben indicarse, porque las precisa para saber en qué lugar de memoria buscar la segunda componente del vector (que es a su vez otro arreglo).

Ejemplo

El siguiente programa muestra la función ***sumaElementos***, que recibe una matriz, y devuelve la suma de sus componentes:

```
#include <stdio.h>

#define TOPEFILAS       10
#define TOPECOLUMNAS    5
```

no hace falta colocar la cantidad de filas (la primera dimensión), sí es obligatorio indicar la cantidad reservada para las columnas, pues este dato lo necesita el compilador para calcular la dirección del comienzo de cada fila

```
float
sumaElementos(const float arreglo[][TOPECOLUMNAS], int dimFil, int
dimCol)
{
      float total= 0;
      int c;

      for( ; dimFil--; )
            for( c= 0; c < dimCol; c++)
                  total+= arreglo[dimFil][c];
      return total;
}
```

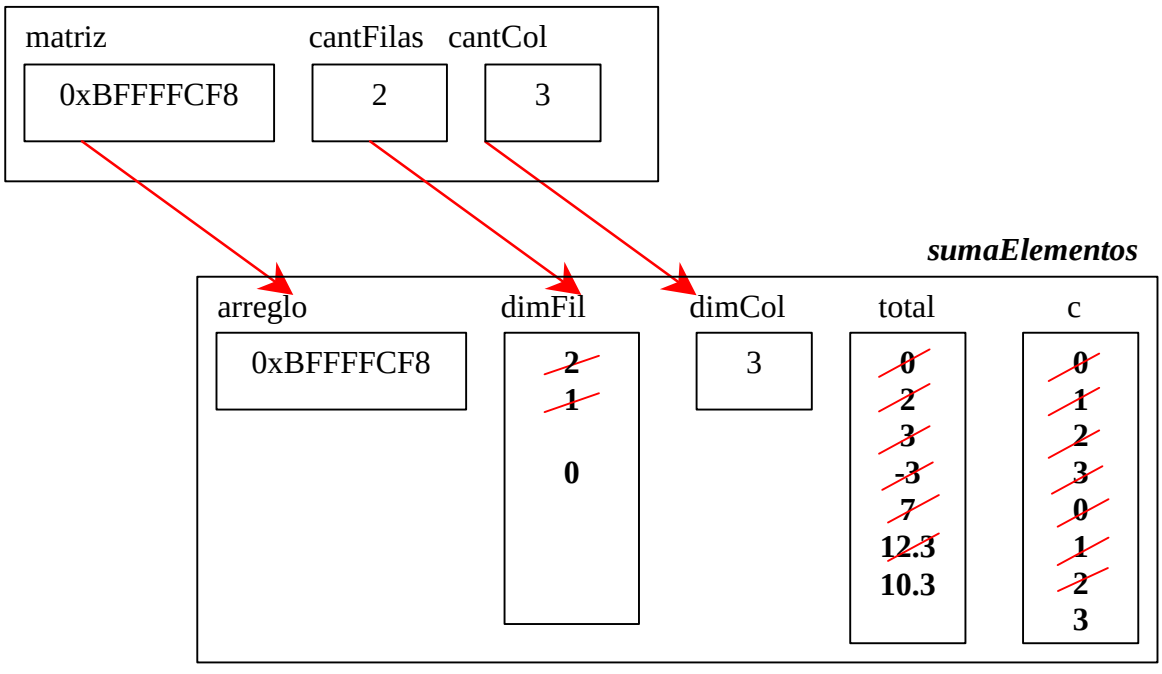
```
int
main(void)
{
      float matriz[TOPEFILAS][TOPECOLUMNAS];
      int cantFilas;
      int cantCol;

      /* aqui se leen las componentes del arreglo, y la verdadera
          dimension queda en la variable cantFilas y cantCol */
      ..........


      printf("Sumatoria=%g\n", sumaElementos( matriz, cantFilas,
                                                    cantCol));

      return 0;
}
```

sólo se invoca con el nombre de la matriz (se va a copiar la dirección de la primera componente)

Para hacer un seguimiento con los *stack frame* , vamos a ejecutar en Linux, suponiendo que el arreglo bidimensional comienza almacenándose en la dirección $0xBFFFFAA0. De los 50 lugares reservados, sólo se ingresaron dos filas y tres columnas, y se asignaron sólo los valores:

matriz[0][0]= 10 **matriz[0][1]=5.3** **matriz[0][2]= -2**
matriz[1][0]=2 **matriz[1][1]= -5** **matriz[1][2]= 0**

main



Cuando la función ***sumaElementos*** comienza a ejecutar, tiene que realizar dos ciclos ***for*** anidados, comenzando con **dimFil** en 2 y **c** en 0.

¿Cómo hace para buscar la componente **arreglo[2[[0]**, o sea la segunda fila del arreglo? Para eso precisa conocer cuántas columnas reservadas (no las indicadas por dimCol) saltear para llegar a la segunda fila. Esto lo hace por medio del calculo:

dirBase + (dimFil * TOPECOLUMNAS + c) * sizeof(float)

donde ***dirBase*** es la dirección de la primera componentes que se le pasó en el primer parámetro formal, **TOPECOLUMNAS** es la constante obligatoria que hay que indicar, ***float*** es el tipo de elemento del arreglo que aparece en la declaración, **dimFil** es el índice de la fila en esa subindicación y **c** es el índice de la columnas en esa subindicación.

En memoria se podrá ir accediendo entonces a las distintas componentes individuales pedidas:

si el arreglo no está lleno hay que saltear algunas componentes hasta llegar a la segunda fila

0xBFFFFCF0			**10**	**5.3**
0xBFFFFD00	**-2**	**?**	**?**	**2**
0xBFFFFD10	**5**	**0**	**?**	**?**
0xBFFFFD20	**?**	**?**	**?**	**?**
0xBFFFFD30	**?**	**?**	**?**	**?**
0xBFFFFD40	**?**	**?**	**?**	**?**
0xBFFFFD50	**?**	**?**	**?**	**?**
0xBFFFFD60	**?**	**?**	**?**	**?**
0xBFFFFD70	**?**	**?**	**?**	**?**
0xBFFFFD80	**?**	**?**	**?**	**?**
0xBFFFFD90	**?**	**?**	**?**	**?**
0xBFFFFDA0	**?**	**?**	**?**	**?**
0xBFFFFDB0	**?**	**?**	**?**	**?**

1.2.2 Pasaje de toda una Fila del Arreglo Bidimensional

En ese caso estaríamos pasando todo un vector (que justamente es su primera componente). Todos se reduciría a lo discutido en la sección 1.2

Ejemplo

El siguiente programa reemplaza todos los elementos de las filas pares de una matriz bidimensional por sus opuestos. Consideramos que para el usuario la primera fila está numerada como 1.

```
#include <stdio.h>

#define TOPEFILAS 10
#define TOPECOLUMNAS 5


void
opuestos( float vector[], int dim)
{
      while ( dim-- )
            vector[ dim ]= - vector[dim];
}



int
main( void )
{
      float matriz[TOPEFILAS][TOPECOLUMNAS];
      int dimFilas, dimCol;
      int rec;


      /* acá se le asignan valores a las componentes de la matriz y se
      deja en las variables dimFilas y dimCol la dimension real */
      .........



      /* se cambian por sus opuestos solo las filas impares */
      for ( rec= 0; rec < dimFilas; rec++)
            if ( (rec + 1) % 2 == 0)
                  opuestos( matriz[ rec ], dimCol);


      return 0;
}
```

Este parámetro recibirá la dirección de la primera componente de un arreglo **unidimensional de float**

2. Naturaleza del Parámetro Formal que recibe un Arreglo

A esta altura nos podríamos preguntar si el parámetro formal que recibe un arreglo es realmente un arreglo. Si fuera así, no sería un **l-value**, o sea no podría soportar figurar en la parte izquierda de una asignación. Vamos a verlo con un ejemplo.

Ejemplo

Vamos a intentar asignarle a una variable de tipo arreglo otra variable del mismo tipo. También vamos a declarar un parámetro formal de tipo arreglo y vamos a intentar asignarle otra variable del mismo tipo

```
#define TOPE      3


void
prueba( float arreglo[])
{
      float auxi[TOPE];

      arreglo= auxi;
}



int
main(void)
{
      float vector[TOPE];
      float otro[TOPE];

      vector[0]= 10;
      vector[1]= 3.5;
      vector[2]= 7;

      vector= otro;

      prueba( vector);

      return 0;
}
```

No hay problema debido a esta asignación.
El parámetro formal sí es un l-value

No compila debido a este intento de asignación.
Vector NO es un l-value

Conclusiones

❖ El **nombre de un arreglo NO es un l-value**, como ya sabíamos. Guarda la dirección del primer elemento como una constante, por eso no permite ser cambiado. Sí permite modificar el contenido de sus componentes.

❖ El **parámetro formal que se corresponde con un arreglo es un l-value**, o sea no es exactamente un arreglo. En realidad es **un puntero** al arreglo**,** como veremos en la próxima clase (variable que guarda la dirección de memoria de otra variable) y así permite que en algún momento cambie su contenido. En el ejemplo anterior, el efecto que se produce al realizar dentro de la función ***prueba*** la asignación ***arreglo = otro*** es que el parámetro ***arreglo*** ahora contenga la dirección de otro arreglo para navegar en sus componentes, pero no produce ningún efecto sobre el parámetros actual original (recordar que los parámetros en C se pasan por valor).

Algunos Ejercicios

a) Indicar qué hace el siguiente programa:

```
void
funcion(int  matriz[][TOPECOLUMNAS], int dim)
{
      int i, j;

      for(i= 0 ; i< dim; i++ )
            for(j= 0; j < dim; j++)
                  matriz[i][j]= i == j;
}
```

Rta: Setea en 1 a todos los elementos de la diagonal principal y en cero al resto, de una matriz cuadrada de dimensión real dada por dim.

b) **Modificar la función anterior para que setee en 1 los elementos de la contradiagonal y en cero al resto.**

Rta:

```
void
funcion(int  matriz[][TOPECOLUMNAS], int dim)
{
      int i, j;

      for(i= 0 ; i< dim; i++ )
            for(j= 0; j < dim; j++)
                  matriz[i][j]= i +  j + 1 == dim;
}
```

c) **Dadas las siguientes declaraciones implementar la función *cantidadHoras* que devuelve la cantidad de horas semanales que tiene una materia dada**

```
typedef  enum {  PGM1, DISCRE, MATE2, FIS1, METOD, LIBRE }  materias;
typedef enum { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES} dias;

int cantidadHoras( materias  horario[ ][VIERNES-LUNES+1],
                             int dimHorasDiarias, materias miMateria);
```

Rta:

```
int
cantidadHoras( materias  horario[ ][VIERNES-LUNES+1],
           int dimHorasDiarias,  materias miMateria)
{
      int hora, dia;
      int total= 0;

      for( hora= 0; hora < dimHorasDiarias; hora++)
            for(dia= LUNES; dia <= VIERNES; dia++)
                  total+= horario[hora][dia] == miMateria;

      return total;

}
```

Un ejemplo de invocación podría ser:

```
int
main(void)
{
      materias horario[][VIERNES-LUNES+1]=
            {
              { METOD,  LIBRE, PGM1,      LIBRE,  MATE2},
              { METOD,  PGM1,  PGM1,      LIBRE,  MATE2},
              { FIS1,   PGM1,  FIS1,      LIBRE,  MATE2},
              { FIS1,   PGM1,  FIS1,            LIBRE,  MATE2},
              { FIS1,   PGM1,  LIBRE,     LIBRE,  LIBRE},
              { FIS1,   PGM1,  LIBRE,     LIBRE,  LIBRE},
              { LIBRE,  PGM1,  LIBRE,     LIBRE,  LIBRE},
              { LIBRE,  LIBRE, LIBRE,     LIBRE,  LIBRE},
              { LIBRE,  LIBRE, LIBRE,     LIBRE,  DISCRE},
              { LIBRE,  LIBRE, LIBRE,     LIBRE,  DISCRE}
            };

      printf("PGM1=%d\n", cantidadHoras(horario,
                        sizeof(horario)/sizeof(horario[0]), PGM1 ));

      return 0;
}
```

Introducción a Punteros

Introducción

Dado que en Lenguaje C el pasaje de parámetros es por valor, cuando una función debe modificar el valor de un parámetro para que lo reciba cambiado quien la invocó, se le debe enviar la dirección de la memoria en la cual se encuentra dicho parámetro. Para poder trabajar con direcciones de memoria hay que utilizar punteros.

En este documento se presenta una introducción al tema de punteros en C, junto a su relación al pasaje de parámetros, y se realizan una serie de aclaraciones acerca de la supuesta intercambiabilidad entre arreglos y punteros.

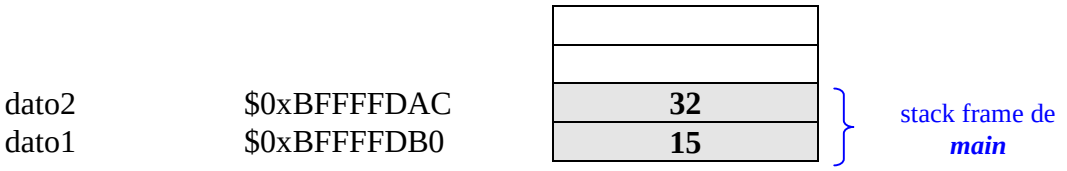
1. Parámetros de Entrada-Salida

Comenzaremos con el típico problema de querer hacer una función que intercambie los contenidos de dos variables.

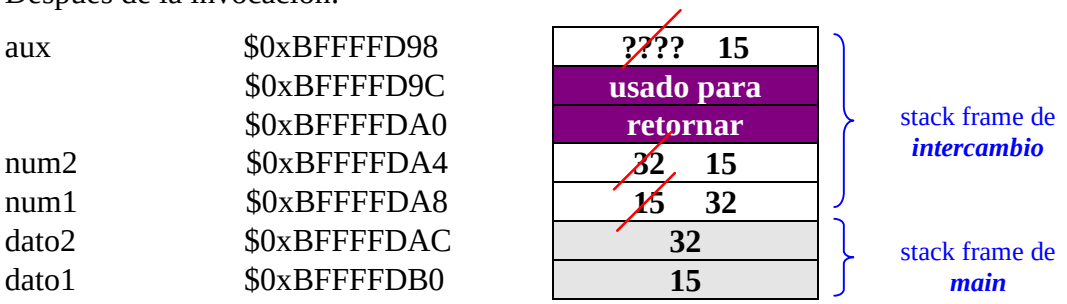
Primera Versión:

```
#include <stdio.h>

void
intercambio( int  num1, int num2)
{
      int aux;

      aux = num1;
      num1 = num2;
      num2 = aux;
}



int
main( void )
{
      int dato1 = 15;
      int dato2 = 32;

      printf( "dato1: %d \t dato2: %d\n", dato1, dato2);
      intercambio( dato1, dato2);
      printf( "dato1: %d \t dato2: %d\n", dato1, dato2);

      return 0;
}
```

Antes de realizar la invocación de la función ***intercambio***:



Después de la invocación:



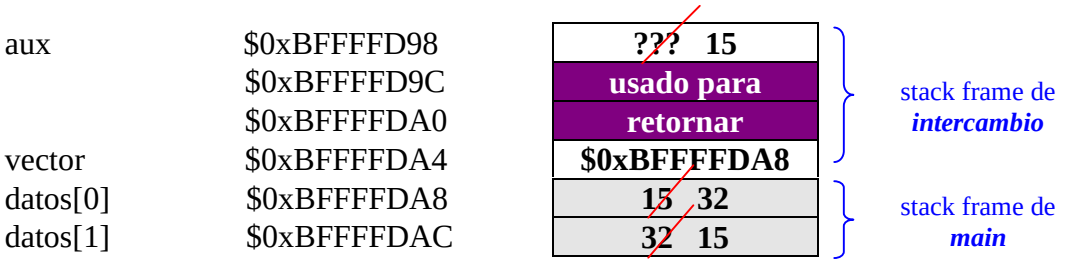
Como se puede observar, al pasar variables simples como parámetros, se intercambiaron los contenidos de ***num1*** y ***num2***, pero no los de ***dato1*** y ***dato2***, pues se trabajó con copias en el stack.

Recordando que las componentes de un arreglo pasado como parámetro sufren los cambios que se realizan sobre ellas, armamos un arreglo con los datos a intercambiar y los pasamos como parámetro:

Segunda Versión:

```
#include <stdio.h>
#define TOPE     2

void
intercambio( int  vector[ ])
{
      int aux;
      aux = vector[0];
      vector[0] =vector[1];
      vector[1] = aux;
}

int
main( void )
{
      int  datos[TOPE];

      datos[0] = 15;
      datos[1] = 32;
      printf( "dato1: %d \t dato2: %d\n", datos[0], datos[1]);
      intercambio( datos);
      printf( "dato1: %d \t dato2: %d\n", datos[0], datos[1]);

      return 0;
}
```

Al realizar la invocación tenemos:



Esto funciona correctamente porque hemos recibido las ***direcciones*** de las variables y ***no sus contenidos***, pero **semánticamente es HORRIBLE!**

Además, otro problema sería el tratar de definir funciones que deban devolver más de un dato, como por ejemplo una función que deba regresar un número complejo: podría devolver la componente real en su nombre, pero ¿cómo regresar la componente imaginaria? Sería de pésima programación devolver un arreglo de un único elemento para solucionar el problema de los parámetros de salida.

La solución sería enviar la dirección de la variable en lugar de enviar su valor. Por suerte, el lenguaje C nos permite hacer esto a través del uso de ***punteros***.

2. Punteros

Un puntero es una variable que contiene la dirección de otra variable. Esto permite considerarlos como un mecanismo de acceso indirecto.

Al declarar una variable tipo puntero, el compilador le asigna una ***cantidad fija*** para que guarde una **dirección** en la cual va a estar el dato en cuestión

Sintaxis

TipoApuntado *** nombreDelPuntero**

Ejemplo:

```
char   *nombrePtero;     /* puntero a char  */
int    *integerPunt;     /* puntero a int */
double *Punt_d:          /* puntero a double  */
```

Mucho cuidado

Si bien todos los punteros tienen el mismo tamaño (el necesario para indicar una dirección de memoria), son distintos en cuanto a lo que apuntan. Se verá más tarde que este punto es muy importante para el momento en el cual se quiera trabajar con el dato apuntado por cada uno de ellos (desreferencia de los punteros)

Hay que prestar mucha atención, ya que el asterisco que se usa en la declaración de una variable tipo puntero, está asociado al nombre de la variable y no al tipo apuntado.

Por lo tanto, si se quieren declarar varias variables tipo puntero en una misma línea, habrá que colocar el asterisco antes de cada nombre de variable.

```
int   *p1, p2;        /* p1 es puntero a int,  p2 es variable int */
int   *p1, *p2;       /* p1 y p2 son punteros a int */
```

2.1. Operador &

Operador & (unario)

Se aplica a un **l-value** y retorna la dirección de memoria en donde se encuentra almacenado el mismo.

Para referenciar una dirección de memoria hay que aplicar el operador & a un **l-value**: una variable o una componente de vector. De esta forma se obtiene la dirección de memoria en la cual se encuentra almacenado dicho objeto.

Ejemplo:

```
int      vector[10];
int      legajo= 5000;
double   sueldo= 890.50;
```

sueldo	$0xBFFFFDA0	**890.50**
legajo	$0xBFFFFDA4	**5000**
vector	$0xBFFFFDA8	**???**
	$0xBFFFFDAC	**???**
	$0xBFFFFDB0	**???**
	$0xBFFFFDB4	**???**
	$0xBFFFFDB8	**???**
	$0xBFFFFDBC	**???**
	$0xBFFFFDC0	**???**
	$0xBFFFFDC4	**???**
	$0xBFFFFDC8	**???**
	$0xBFFFFDCC	**???**

- *Con **&legajo** obtendríamos la dirección **BFFFFDA4***
- *Con **&sueldo** obtendríamos la dirección **BFFFFDA0***
- *Con **&vector[0]** obtendríamos la dirección **BFFFFDA8***
- *Con **&vector[9]** obtendríamos la dirección **BFFFFDCC***

*2.2. Operador **

Operador * (unario)

Se aplica a un tipo puntero y retorna el l-value al cual apunta. Sirve para ***desreferenciar*** un puntero, es decir, para obtener el elemento al cual apunta.

Ejemplo Global

Sea la siguiente declaración de variables

```
int  x= -5,;
int y = 20;
int  *px,  *py;
```

py	$0xBFFFFDA8	???
px	$0xBFFFFDAC	???
y	$0xBFFFFDB0	20
x	$0xBFFFFDB4	-5

Para inicializar punteros se necesitan valores que representen direcciones de objetos del tipo apuntado:

```
px = &x;
py = &y;
```

py	$0xBFFFFDA8	0xBFFFFDB0
px	$0xBFFFFDAC	0xBFFFFDB4
y	$0xBFFFFDB0	20
x	$0xBFFFFDB4	-5

Con *p desreferenciamos como l-value al entero apuntado por px, pudiendo usarlo en una asignación o autoincremento:

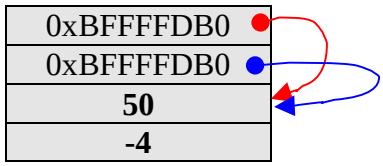
```
(*px)++;
*py = 50;
```

py	$0xBFFFFDA8	0xBFFFFDB0
px	$0xBFFFFDAC	0xBFFFFDB4
y	$0xBFFFFDB0	**50**
x	$0xBFFFFDB4	**-4**

Notar la diferencia entre px = py y *px = *py:

- **px = py;**

py	$0xBFFFFDA8	0xBFFFFDB0
px	$0xBFFFFDAC	0xBFFFFDB0
y	$0xBFFFFDB0	**50**
x	$0xBFFFFDB4	**-4**



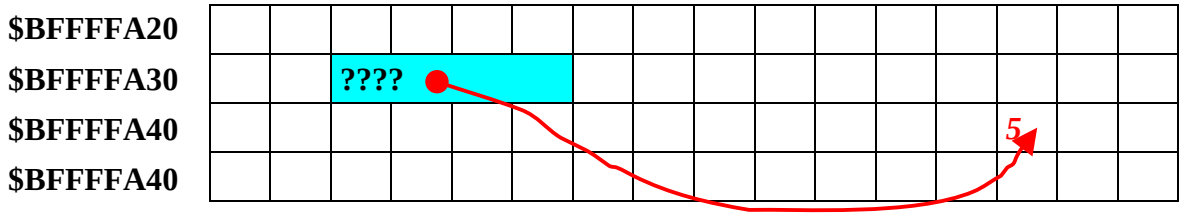
- ***px = *py;**

py	$0xBFFFFDA8	0xBFFFFDB0
px	$0xBFFFFDAC	0xBFFFFDB4
y	$0xBFFFFDB0	**50**
x	$0xBFFFFDB4	**50**

Ejemplo de MALA asignación de un puntero:

```
int  *punt;
*punt = 5;
```

Supongamos que el compilador alocó a la variable **punt** en la dirección 0xBFFFFA32 y esa dirección contiene información desconocida ????



Con la asignación estamos almacenando un valor 5 en lo apuntado por **punt**, es decir estamos tocando la dirección de memoria ????, lo cual puede ser catastrófico !!!

NULL

Constante simbólica (está en el header stdio.h) para indicar que un puntero apunta a "nada" (**dirección nula)**. Por supuesto, es un error grave intentar desreferenciar un puntero con valor NULL.

Ejemplo

Cuando se declara una variable de tipo arreglo se reserva en memoria el lugar necesario para almacenar todas sus componentes, en el lugar correspondiente a su característica (Stack, Data, BSS). En el caso particular de encontrarse en el Stack, sus componentes no quedan inicializadas, a menos que se haya realizado esta acción en la declaración.

Hay que tener mucho cuidado para no obtener resultados impredecibles. El siguiente código brindará una salida correcta para las dirección de cada componente, pero como sus contenidos no fueron asignados ni inicializados, se imprimirá información no deseada ("basura del stack"):

```
int
main(void)
{
      int rec, numeros[4];

      for (rec=0; rec< sizeof(numeros)/sizeof(numeros[0]) ; rec++ )
           printf("La dirección de numeros[%d] es %p y su contenido
                        es %d\n", rec, &numeros[rec], numeros[rec]);
      return 0;
}
```

Una posible versión correcta consitiría en inicializar el arreglo en el momento de la declaración:

int numeros[ ] = {10, 20, 30, 40};

obteniéndose como posible salida:

```
La  dirección de numeros[0] es  BFFFFDCB y su contenido es  10
La  dirección de numeros[1] es  BFFFFDCF y su contenido es  20
La  dirección de numeros[2] es  BFFFFDD3 y su contenido es  30
La  dirección de numeros[3] es  BFFFFDD7 y su contenido es  40
```

3. Pasaje de Puntero como Parámetro

En la definición (prototipo) de la función:

tipo nombreFuncion(tipoApuntado *parametroFormal, . . .);

En la invocación de la función:

nombreFuncion(& parametroActual, . . .);

Volviendo a nuestro problema inicial de la función que intercambia los contenidos de dos variables, presentamos una versión final con punteros, que permite recibir de vuelta los parámetros realmente cambiados.

Versión usando Punteros:

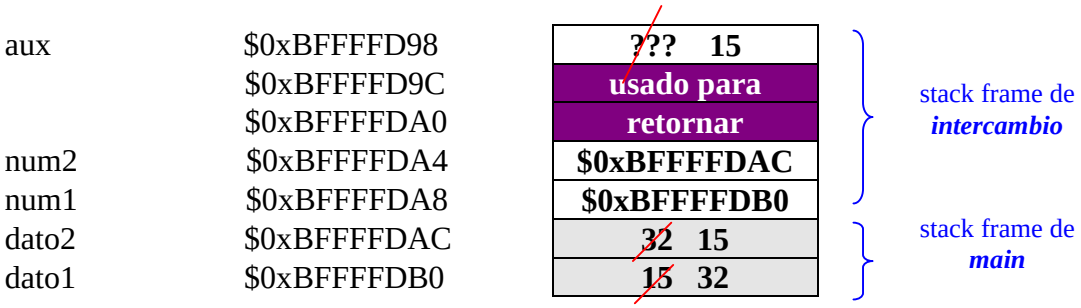
```
#include <stdio.h>

void
intercambio( int *num1, int *num2)
{
        int aux;

        aux   =   *num1;
        *num1   =   *num2;
        *num2   =   aux;
}
```

*Desreferenciamos lo apuntado por **num1** y por **num2**, ya que ambas son variables tipo puntero y contienen las direcciones en donde se encuentran los datos requeridos*

```
int
main( void )
{
      int dato1 = 15;
      int dato2 = 32;

      printf ( "Antes: \n dato1: %d \t dato2: %d \n", dato1, dato2 );

      intercambio( &dato1 , &dato2 );

      printf ( "Despues:\n dato1: %d \t dato2: %d \n", dato1, dato2 );

      return 0;
}
```

enviamos las direcciones de las variables a intercambiar

Al realizar la invocación tenemos:



Importante

Los punteros pueden ser usados para retorno de múltiples resultados

Ejemplo

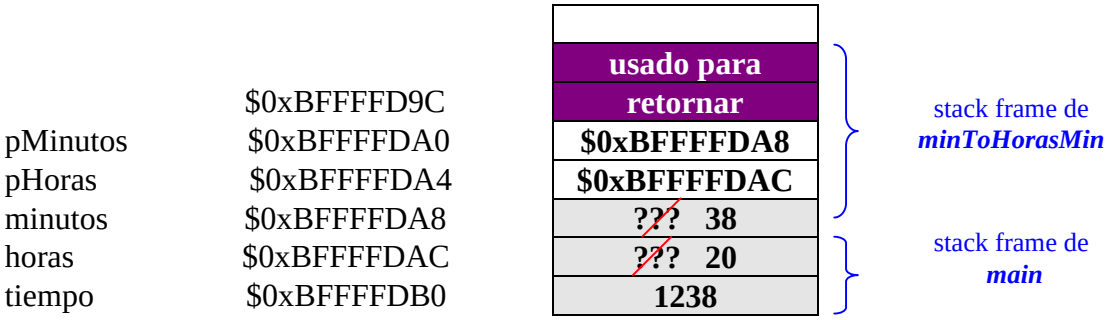
El siguiente programa recibe un intervalo de tiempo expresado en minutos (en un parámetro de **entrada)** y lo devuelve convertido en horas y minutos, a través de dos parámetros de **salida**.

```
/*   Archivo:  convTime.c
**   El siguiente programa lee un tiempo expresado en minutos desde la
**   entrada estándar y lo convierte a horas y minutos
*/

#include <stdio.h>
#include  "getnum.h"

#define  MINUTOS_POR_HORA    60


/*  Funcion que recibe un tiempo expresado en minutos y devuelve en
**  dos parametros ese tiempo expresado en horas y minutos
*/
static void minToHorasMin(int tiempo, int *pHoras, int *pMinutos);



int
main(void)
{
      int tiempo, horas, minutos;

      tiempo = getint("\nIngrese una cierta cantidad de minutos: ");

      minToHorasMin(tiempo, &horas, &minutos);

      printf("Ud. ingreso: %d hs. %02d min.\n", horas, minutos );

      return 0;
}


static void
minToHorasMin(int tiempo, int  *pHoras,  int  *pMinutos)
{
      *pHoras = tiempo / MINUTOS_POR_HORA;
      *pMinutos = tiempo % MINUTOS_POR_HORA;
}
```

Notar que desde ***main***, se invoca la función que realizará la transformación con la variable que contiene el tiempo (copiamos su contenido, ya que es un parámetro de entrada) y con las direcciones de las variables en las cuales se quieren las horas y los minutos (parámetros de salida).

Luego de la invocación, las variables ***horas*** y ***minutos*** ya han sido modificadas en la zona del stack correspondiente a main, por ese motivo se imprimen sus contenidos y no sus direcciones.

Al realizar la invocación tenemos:

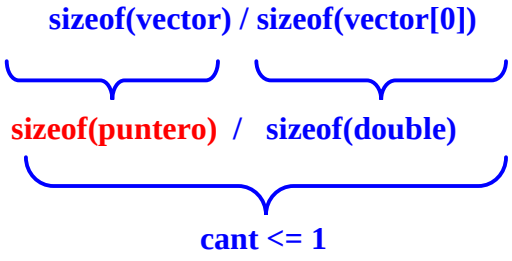


3.1. Tamaño de un Arreglo pasado como Parámetro

Sea el siguiente programa que utiliza una función para imprimir un vector, a la cual se le envía el vector deseado:

```
#include <stdio.h>

void imprimir(double vector[ ]);

int
main(void)
{
      double valores[ ] = {1.5, 2.89, 5.0};

      imprimir(valores);
      return 0;
}


void
imprimir(double vector[ ])
{
      int rec, cant;

      cant = sizeof(vector) / sizeof(vector[0]);

      for (rec= 0; rec < cant ; rec++)
           printf("En la dirección %p  hay un %g\n",
                                      &vector[rec], vector[rec]);
}
```

Al calcular ***cant***, se obtendrá el cociente entre el tamaño de un puntero (cantidad de bytes necesarios indicar una dirección de memoria) y el tamaño de la primer componente del arreglo, en este caso un double:



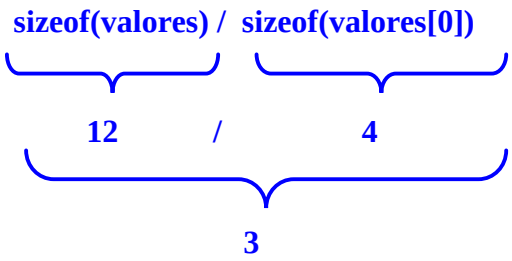
El problema es que el parámetro formal en el cual se va a recibir un arreglo, no es otra cosa que un puntero, en el cual se copiará, en el momento de la invocación, la dirección del comienzo del vector que se use como parámetro actual.

De acuerdo a lo visto anteriormente, la pregunta que surge es ¿cómo hacer para indicarle a una función el tamaño del arreglo que se le va a enviar? La única solución posible es **enviar otro parámetro con el tamaño del arreglo.**

En nuestro ejemplo anterior, dentro de ***main*** (que es donde se declaró el arreglo), se puede calcular el tamaño y enviárselo como argumento a la función, que validará con un ciclo para no recorrer componentes de más:

```
#include <stdio.h>

void imprimir(double vector[ ],  int cant);

int
main(void)
{
      double valores[ ] = {1.5, 2.89, 5.0};

      imprimir(valores, sizeof(valores)/sizeof(valores[0]));
      return 0;
}


void
imprimir(double vector[],  int cant)
{
      int rec;
      for (rec= 0; rec < cant ; rec++)
           printf("En la dirección %p  hay un %g\n", &vector[rec],
                                                    vector[rec]);
}
```

Como en ***main*** está la declaración de la variable ***valores***, si se quiere calcular su tamaño (en bytes) a través del operador ***sizeof***, se obtendrá efectivamente la cantidad de bytes reservados para la misma. Al dividir dicho valor por lo que ocupa una de sus componentes, se obtiene la cantidad tope de componentes (recordar que se pueden usar menos componentes, en cuyo caso habrá que controlar la dimensión real con un contador.



Por otra parte, cuando uno declara el vector, reserva espacio que generalmente no se ocupa realmente. Por esta razón siempre que se **envíe un vector a una función**, hay que enviarle **también su dimensión real**, excepto que el vector sea de caracteres y termine en '\0' (ver sección 2).

4. Arreglos y Punteros

Recordando el uso de los operadores & y *, y sabiendo que el nombre de un arreglo contiene la dirección de la primera componente, podemos decir que:

nombreArreglo *es equivalente a* **&nombreArreglo[0]**

y

***nombreArreglo** *es equivalente a* **nombreArreglo[0]**

Existe, al menos en apariencia, una relación entre arreglos y punteros. Más adelante desarrollaremos este punto en profundidad.

En lenguaje C, a un puntero se le puede sumar o restar un entero. El efecto es el siguiente:

Si **p** *es un* **puntero** *a* **TipoApuntado**, *la expresión*

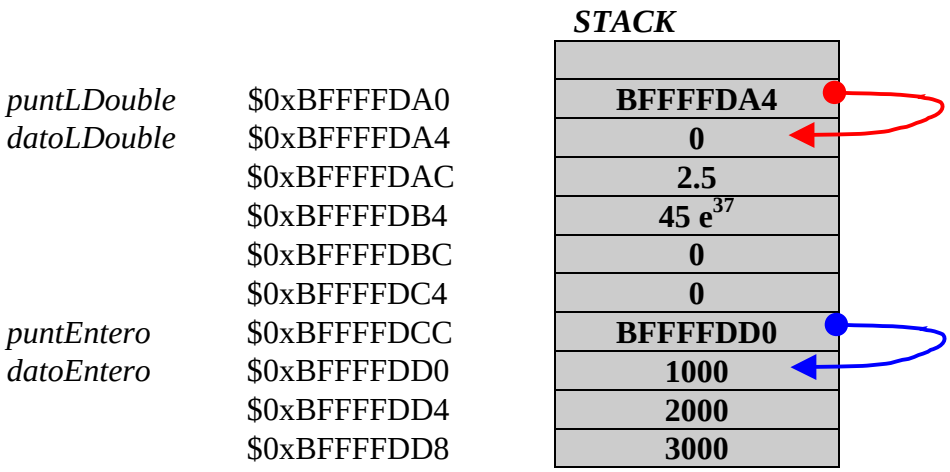
p + n

devuelve la dirección que resulta de hacer

p + n * sizeof(TipoApuntado)

Ejemplo

```
int
main(void)
{
      int   datoEntero[3] = { 1000, 2000, 3000 };
      int   * puntEntero;
      long  double   datoLDouble[5] = { 0, 2.5,  45E+37 };
      long  double *puntLDouble;


      puntEntero =  datoEntero;


      puntLDouble = datoLDouble;
      ..........................


      return 0;
}
```



- Podemos ver que la expresión **puntEntero + 2** devolverá la dirección

  BFFFFFDD0 + 2 * 4 = BFFFFFDD8

  que coincide con la dirección de **datoEntero[2]**.

- En cambio la expresión **puntLDouble + 2** aumentará la dirección en 16 bytes, devolviendo la dirección

  BFFFFFDA4 + 2 * 8 = BFFFFFDB4

  que casualmente es la dirección de **datoLDouble[2]**.

Por la forma en que se realiza el incremento de punteros, existe una equivalencia entre la suma de un entero y un puntero que contiene la dirección de un arreglo, y las componentes de dicho arreglo (ambos con el mismo tipo apuntado):

Si se tienen

nombrePuntero = nombreArreglo

entonces

nombrePuntero + n *es equivalente a* **&nombreArreglo[n]**

y análogamente

***(nombrePuntero + n)** *es equivalente a* **nombreArreglo[n]**

Cabe aclarar que cuando el compilador encuentra una expresión subindicada, donde la variable es un arreglo o un puntero la transforma en su equivalente de incremento de una dirección, sin hacer ningún tipo de chequeo acerca de los límites entre los cuales se encuentra.

Por este motivo, aunque el Stack crezca hacia las direcciones bajas de memoria, las componentes de un arreglo dentro del stack se almacenan de manera que la primera componente quede en la zona más baja reservada para el mismo. de esta forma al aumentar la dirección de la "cabeza", sigue estando dentro del stack.

nombreVariable [n] *es transformada por el compilador en* ***(nombreVariable + n)**

Dado que cuando una función espera recibir un arreglo en su parámetro formal lo que realmente recibe es una copia de la dirección de comienzo del arreglo, los siguientes prototipos son equivalentes:

```
tipoFuncion   nombreFuncion(  tipo  *parametroFormal, . . .  );

tipoFuncion   nombreFuncion(  tipo  parametroFormal[ ], . . . );
```

Cadena de Caracteres

Introducción

En este documento se muestra el tratamiento especial que hace el lenguaje sobre los arreglos de caracteres.

1. Strings (cadenas null terminated)

En lenguaje C no existe el tipo string, como en otros lenguajes. Lo que llamamos ***string*** o ***cadena de caracteres*** es simplemente un arreglo de ***char.***

Para muchas funciones de la Biblioteca Estándar los strings que se envían como argumentos deben se NULL TERMINATED, es decir, deben terminar con el caracter '\0' (ASCII = 0). De esta forma se evita tener que enviar la cantidad de caracteres del string (longitud) , ya que detecta la finalización del mismo al encontrar dicha marca.

Esto resultaba muy difícil de implementar con arreglos de int, long, float, double, ya que los datos numéricos permitirían cualquier número como válido, entonces ¿cuál sería la marca elegida? Simplemente no la hay. Por eso al enviar un arreglo como argumento de una función, se lo debe acompañar con su tamaño.

1.1. Strings Constantes

Cuando el compilador encuentra un string constante (caracteres entre comillas dobles), **aloca lugar para cada uno de dichos caracteres más un lugar para el '\0',** en ***Data Segment*** o bien en ***Text Segment*** (depende del compilador).

¿Dónde se pueden usar los strings constantes?

- **Inicializando punteros a char:**

  char * cartel = "hola";

- **Asignando punteros a char:**

  char * cartel;
  cartel = "hola";

*No pisamos memoria por que estamos asignando al puntero a char la dirección de una zona de memoria que **YA FUE RESERVADA por el compilador para la constante string***

- **Como argumento de función:**

  printf (“hola”);

- **En referencias de punteros a char:**

  printf (“%s\n”, “hola” + 2); /* imprime sólo las dos últimas letras */

Atención

Para abreviar la **inicialización de arreglos de caracteres**, lenguaje C permite utilizar strings, pero en ese caso **NO son tratados como strings constantes**:

char cartel[ ] = “hola”;

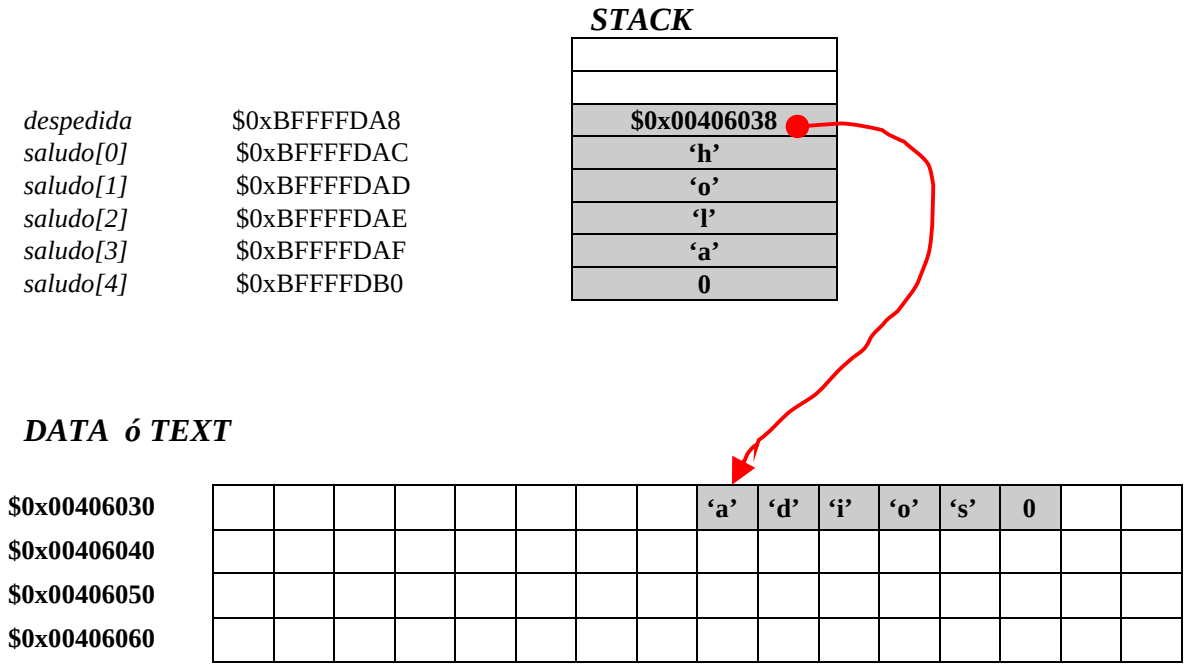
es una forma abreviada de hacer

char cartel[ ] = {‘h’, ‘o’, ‘l’, ‘a’, ‘\0’};

Ejemplo

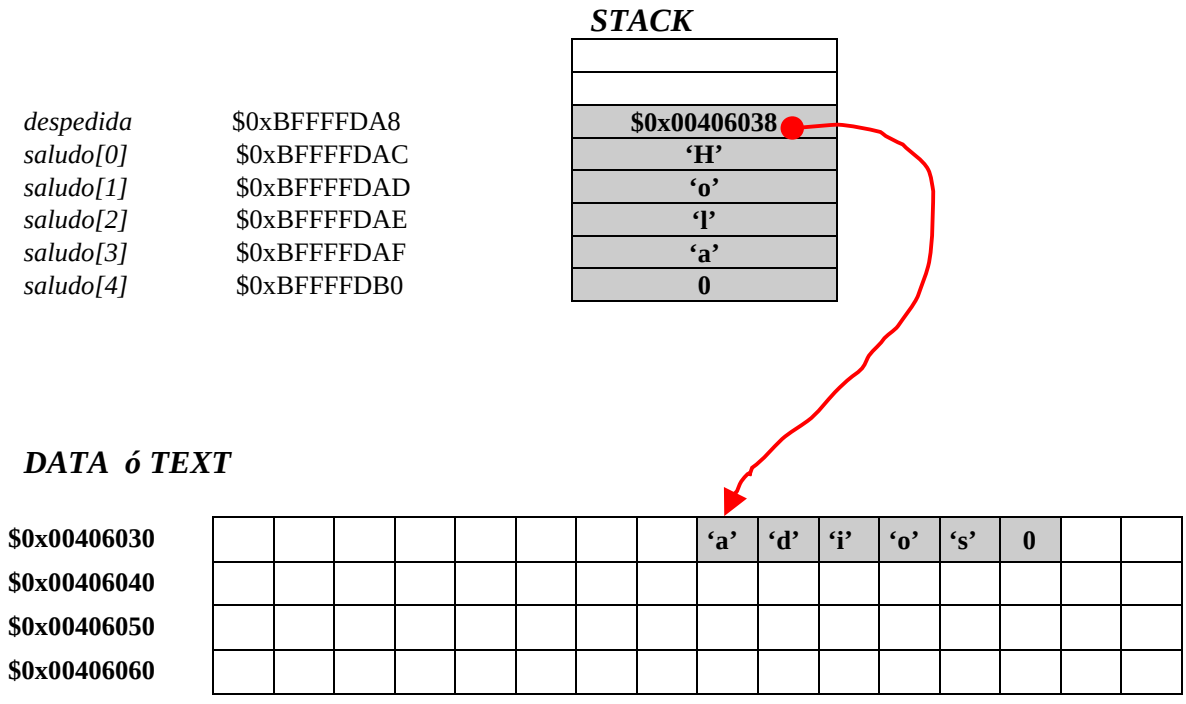
En este ejemplo se muestra la diferencia entre el uso de string para la inicialización de arreglos (donde no se los considera strings constantes) y la inicialización de punteros a char con strings constantes (que ya fueron previamente almacenados por el compilador en Data o Text).

```
int
main(void)
{
      char saludo[] = “hola”;
      char *despedida= “adios”

      ..........................................

      return 0;
}
```



Gracias al ejemplo se puede entender por qué:

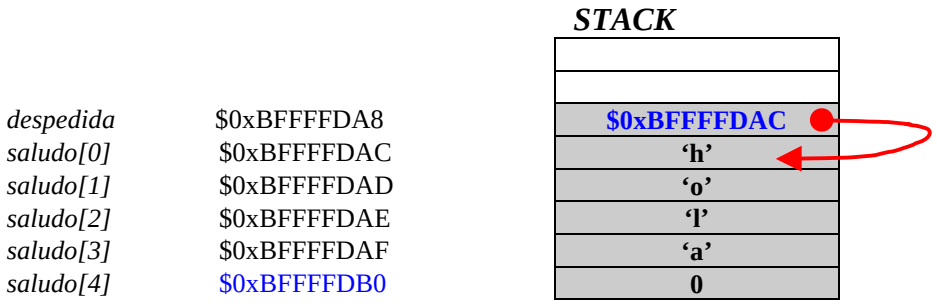
- Es válido hacer **saludo[0] = 'H';**
- Es inválido **despedida[0] = 'A';** (se puede estar intentando acceder a zona Text)



El ejemplo también permite explicar por qué:

- Es válido hacer **despedida = saludo;**
  (despedida guardará la dirección de memoria en donde comienza el arreglo saludo)

- Es inválido hacer **saludo = despedida;**
  (no se puede modificar el contenido de la variable saludo, ya que se trata del nombre de un arreglo, no es un l-value)



Ejemplo de MALA asignación:

```
char  *cartel;

cartel[0] = 'M';
cartel[1] = 'A';
cartel[2] = 'L';
cartel[3] = '\0';
```

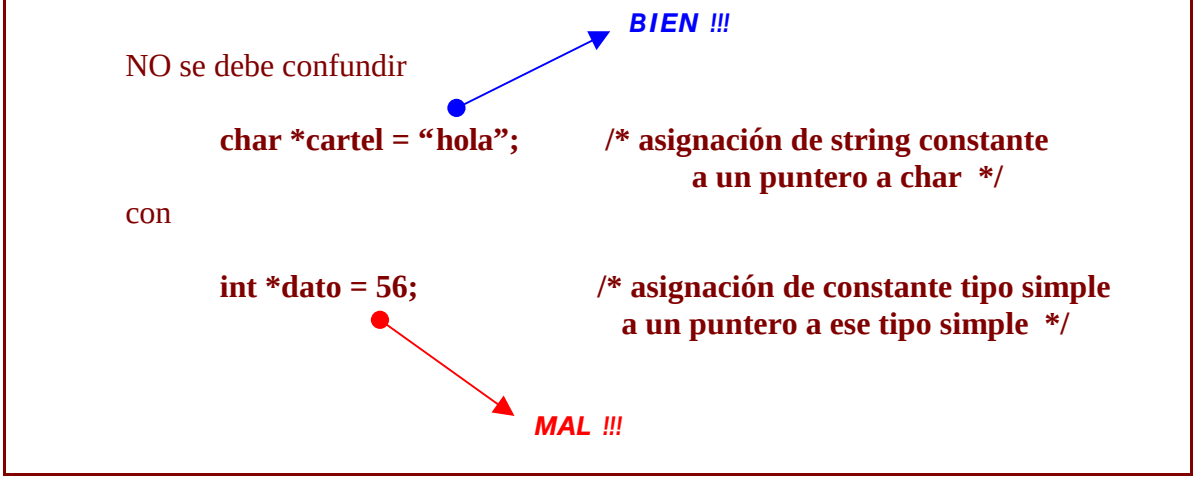
Nunca inicializamos la dirección de ***cartel***, para que a partir de dicha dirección almacene los caracteres que se desea: estaremos ***"pisando memoria"***

Ejemplo de asignación CORRECTA:

```
char  *cartel;
cartel = "BIEN";
```

Es correcto porque con la asignación logramos que cartel apunte a la zona de memoria en donde el compilador **YA almacenó** el string constante "BIEN"

Mucha Atención



Ejemplo 1

Veremos cómo armar un string "null terminated" en tiempo de ejecución, para poder usarlo en una función que requiere como parámetro ese tipo de string:

```
.............................
char   cadena[5];
char  *pCadena;
.............................

for (i = 0;  i < 4; i++)
      cadena[i] = 'a' + i;

printf ("cadena: %s\n", cadena);
.............................
```

		STACK
pCadena	$0xBFFFFDA8	**???**
cadena[0]	$0xBFFFFDAC	**'a'**
cadena[1]	$0xBFFFFDAD	**'b'**
cadena[2]	$0xBFFFFDAE	**'c'**
cadena[3]	$0xBFFFFDAF	**'d'**
cadena[4]	$0xBFFFFDB0	**???**

Este código compila, pero al ejecutarlo imprime: **abcd5#2º?s1|js3. . .**
ya que no encuentra el '\0' después del caracter 'd'

Una posible solución sería:

```
.............................
for (i = 0;  i < 4; i++)
      cadena[i] = 'a' + i;
cadena[4] = '\0';                    /*  o bien cadena[4] = 0  */
printf ("cadena: %s\n", cadena);
.....................................
```

Ejemplo 2

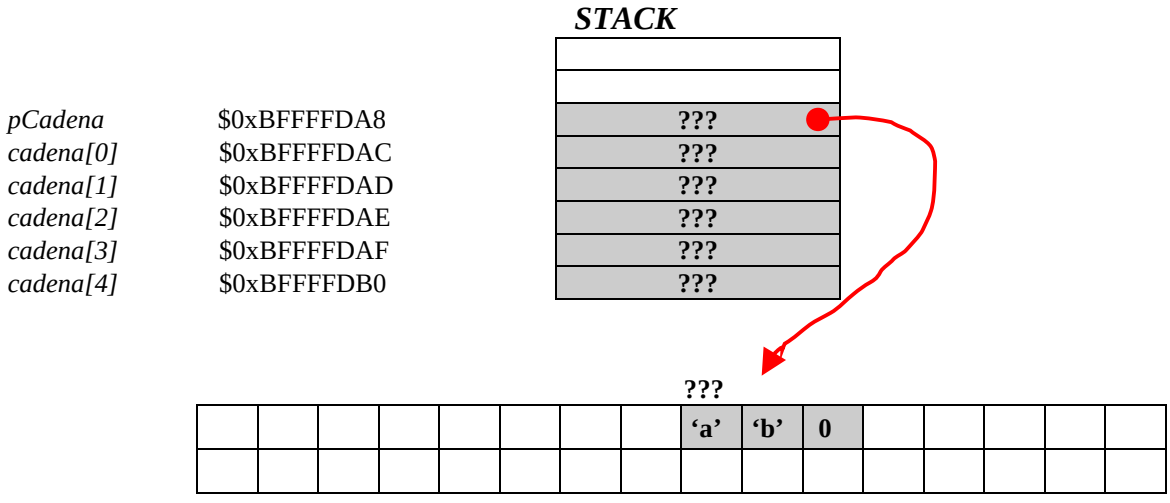
El siguiente código **NO COMPILA** porque se trata de hacer la asignación de una dirección (la del string constante "abcd") en el nombre de un arreglo, que no es un l-value

```
char   cadena[5];
char  *pCadena;

cadena = "abcd";
```

Ejemplo 3

Si bien el siguiente código compila, tiene un **error grave** pues estamos **"pisamos memoria"**, ya que *pCadena* no fue inicializado con una dirección válida. Si en la zona en donde colocamos 'a', 'b' y 0 había información útil, la perdimos.

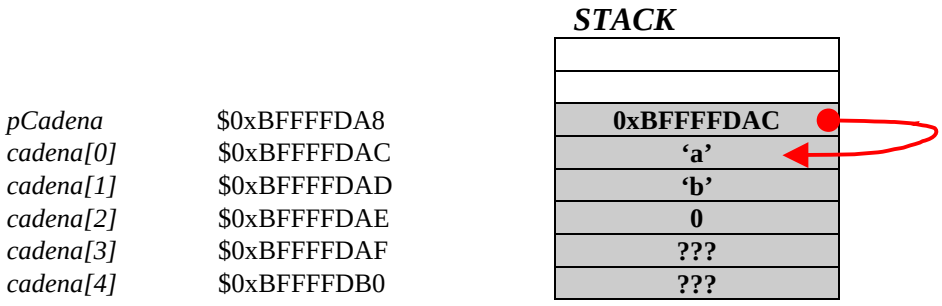
```
char   cadena[5];
char  *pCadena;

*pCadena = 'a';          /*  idem a  pCadena[0] = 'a' */
*(pCadena + 1) = 'b'     /*  idem a  pCadena[1] = 'b' */
*(pCadena + 2) = 0       /*  idem a  pCadena[2] = 0   */
```



Ejemplo 4

El siguiente es un ejemplo de fragmento de código correcto.

```
char   cadena[5];
char  *pCadena;

pCadena= cadena;
*pCadena = 'a';          /*  idem a pCadena[0] = 'a' */
*(pCadena + 1) = 'b'     /*  idem a pCadena[1] = 'b' */
*(pCadena + 2) = 0       /*  idem a pCadena[2] = 0   */
```



1.2. Strings como Parámetros

Supongamos que queremos imprimir un cierto string en mayúsculas, a través de cierta función ***imprimirUpper***, que sólo recibe el string:

Una primera aproximación nos puede llevar a pensar que, como un string es un arreglo de caracteres y cada *char* ocupa 1 byte, el operador *sizeof* nos puede decir la cantidad de caracteres del arreglo recibido.

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

void imprimirUpper(char vector[ ]);


int
main(void)
{
      char cartel[ ] = "Hola";

      imprimirUpper(cartel);
      return 0;
}
```

```
void
imprimirUpper(char vector[])
{
      int rec, cant;

      cant = sizeof(vector);

      for (rec= 0; rec < cant ; rec++)
            putchar(toupper(vector[rec]));
      putchar('\n');
}
```

El error cometido es que ***cant*** siempre va a dar el tamaño necesario(en bytes) para almacenar una dirección (por ejemplo 4), ya que el parámetro que la función recibe en el stack no es todo el string sino la dirección de donde comienza..

Una posible solución es usar una función que calcule la cantidad de caracteres del string. Para esto la Biblioteca Estándar nos brinda la función ***strlen,*** que devuelve la cantidad de caracteres existentes entre el primer caracter del string y la primera ocurrencia del caracter de ASCII cero ('\0'), excluyendo a éste último.

A continuación mostramos una versión de la función imprimirUpper, que utiliza strlen para calcular la longitud del string a imprimir:

```
void
imprimirUpper(char vector[])
{
      int rec, cant;

      cant = strlen(vector);    /* no cuenta el caracter nulo final */

      for (rec= 0; rec < cant ; rec++)
            putchar(toUpper(vector[rec]));

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

Sin embargo hay que tener cuidado al usar funciones de la biblioteca estándar preparadas para trabajar con cadenas "null terminated", ya que como lo indica su nombre, se considera que terminan en '\0'

Ejemplo

En este ejemplo se muestra lo que ocurre si se invoca una función para string "null terminated" con una cadena que no termina en el caracter nulo esperado.

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

void imprimirUpper(char vector[ ]);

int
main(void)
{
      char cartel[5];

      cartel[0] = 'H';
      cartel[1] = 'o';
      cartel[2] = 'l';
      cartel[3] = 'a';
      imprimirUpper(cartel);

      return 0;
}


void
imprimirUpper(char vector[])
{
      int rec, cant;

      cant = strlen(vector);    /* no cuenta el caracter nulo final */

      for (rec= 0; rec < cant ; rec++)
            putchar(toupper(vector[rec]));
      putchar('\n');
}
```

Obviamente no funciona correctamente pues el string que recibe ***strlen*** no es "null terminated". La función ***strlen*** contará la cantidad de caracteres (bytes) entre 'H' y la primera ocurrencia en memoria de un ASCII nulo.

Una posible salida sería: **HOLAY%!PR?R7BA.....**

Ejercicios con Punteros

Introducción

En este documento se plantean una serie de ejercicios que ayudan a conceptualizar los distintos temas vistos durante el dictado de este curso. Las respuestas se encuentran al final.

Ejercicio 1

Dadas los siguientes fragmentos de código indicar cuáles producen resultados equivalentes al siguiente patrón

```
int numeros[4], *pn;
int i;

for ( i= 0; i < 4; i++ )
      numeros[ i ] = 0;
```

a)

```
int numeros[4], *pn;
int i;

for ( i= 0; i < 4; ++i )
      numeros[ i ] = 0;
```

b)

```
int numeros[4], *pn;
int i;

pn = numeros;
for ( i= 0; i < 4; i++ )
      pn[ i ] = 0;
```

c)

```
int numeros[4], *pn;
int i;

pn = numeros;
for ( i= 0; i < 4; i++ )
      *( pn + i ) = 0;
```

Ejercicio 2

Tomar el ejercicio anterior, suponiendo que se ejecuta en una arquitectura de 32 bits, que a la primera componente del arreglo se le asigna la dirección BFFFFDA0, que a la variable ***pn*** se le asigna la dirección BFFFFD9C y que a la variable ***i*** se le asigna la dirección BFFFFD98.

a) Dibuja el ***stack*** un instante después de haber comenzado la ejecución de la función ***main***, pero antes de ejecutar la primera instrucción.

b) Hacer un seguimiento de cómo va cambiando el ***stack*** en el tiempo, para el fragmento de código del ***ejercicio 1 parte c.***

Ejercicio 3

Dadas las siguientes funciones indicar cuáles de los siguientes prototipos resultan equivalentes entre sí:

a) void funcion (int valores[]);
b) void funcion (int * valores);
c) void funcion (int valores [200]);

Ejercicio 4

Para los prototipos del ejercicio 3, indicar cuáles de las siguientes invocaciones compilarían:

a)
```
int nro;
funcion( &nro );
```

b)
```
int * nro;
funcion ( nro );
```

c)
```
int numeros[10];
funcion( numeros );
```

d)
```
int numeros[10];
funcion( &numeros[0] );
```

Ejercicio 5

Explicar por qué lenguaje C decidió que cuando se pasa un arreglo como argumento en la invocación de una función, sólo se pasa la dirección de su primera componente.

Ejercicio 6

Suponiendo que el stack comienza en la dirección 0xBFFFFDB0, que el Data empieza en 0xCCCCAAAA y que la zona BSS comienza en 0xBBBB0000, indicar la salida en cada uno de los siguientes programas, en el caso de que compilen:

a)

```
#include <stdio.h>

int
main(void)
{
      int numero= 5, *puntero;
      puntero= &numero;
      printf("%p\n %p \n %p \n %d\n", &numero, &puntero, puntero,
                                                          *puntero);
      return 0;
}
```

b)

```
#include <stdio.h>

int
main(void)
{
      int numero= 5;
      static int *puntero;

      puntero= &numero;
      printf("%p\n %p \n %p \n %d\n", &numero, &puntero, puntero,
                                                          *puntero);
      return 0;
}
```

c)

```
#include <stdio.h>

int
main(void)
{
      int numero= 5;
      static int *puntero = &numero;

      printf ("%p\n %p \n %p \n %d\n", &numero, &puntero, puntero,
                                                          *puntero);
      return 0;
}
```

Ejercicio 7

Indicar cuál de los siguientes programas contienen errores:

a)

```
#include <stdio.h>

void
leer( char *s)
{
      /* esta función lee en s un string desde la entrada estandar */
}

int
main(void)
{
      char  *buffer;

      leer (cartel);
      printf (“%s”, buffer);
      return 0;
}
```

b)

```
#include <stdio.h>

void
leer( char *s)
{
      /* esta función lee en s un string desde la entrada estandar */
}

int
main(void)
{
      char  cartel[100];

      leer (cartel);
      printf (“%s”, cartel);
      return 0;
}
```

Ejercicio 8

Hacer un programa que lea dos matrices y calcule, si se puede, su producto.

$$\underset{m.n}{A} \;\acute{}\; \underset{n.p}{B} = \underset{m.p}{C} \qquad \text{donde} \qquad c_{ij} = \sum_{1}^{n} a_{ik} * b_{kj}$$

Respuestas

Ejercicio 1

a) Es equivalente ya que las expresiones **i++** e **++i** no son usadas sino para aprovechar sus efectos colaterales: ***"incrementar la variable i".***

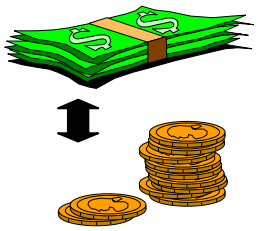
Los valores de las expresiones ++i e ++i sí resultan distintos, pero en este caso son descartados.

b) Es equivalente ya que el puntero **pn** guarda la dirección de la primera componente del arreglo **numeros**, debido a la asignación.

suma 1 componente = sizeof(int)

La expresión **pn[i]** es traducida por el compilador como ***(pn + i)**

valor temporario



pn[0] es traducido como ***(pn + 0)**
pn[1] es traducido como ***(pn + 1)**
pn[2] es traducido como ***(pn + 2)**
pn[3] es traducido como ***(pn + 3)**

c) Es equivalente al primero y la explicación es la misma que en el caso anterior.

El lenguaje C trata a la ***subindicación de arreglos*** como un ***puntero más un desplazamiento***. Esta técnica fue heredada de BCPL (antecesor de C)

ß

NO hay chequeo sobre los índices usados en un arreglo para detectar si se encuentran dentro de los límites del mismo en tiempo de ejecución.

Ejercicio 2

a)

BFFFFD98	????????	*i*
BFFFFD9C	????????	*pn*
BFFFFDA0	????????	*numeros*
BFFFFDA4	????????	
BFFFFDA8	????????	
BFFFFDAC	????????	

b)

- Cuando se ejecuta **pn= numeros**, se obtiene:

BFFFFD98	**????????**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**????????**	*numeros*
BFFFFDA4	**????????**	
BFFFFDA8	**????????**	
BFFFFDAC	**????????**	

- **i= 0;**
- ***(pn + i) = 0;** ⇒ * (BFFFFDA0 + 0) = 0 ⇒ * (BFFFFDA0) = 0

esta suma se realiza sumándole a la dirección dada 0 componente, cada una de tamaño sizeof(int)

BFFFFD98	**0**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**0**	*numeros*
BFFFFDA4	**????????**	
BFFFFDA8	**????????**	
BFFFFDAC	**????????**	

- **i= 1;**
- ***(pn + i) = 0;** ⇒ * (BFFFFDA0 + 1) = 0 ⇒ * (BFFFFDA4) = 0

BFFFFD98	**1**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**0**	*numeros*
BFFFFDA4	**0**	
BFFFFDA8	**????????**	
BFFFFDAC	**????????**	

- **i= 2;**
- ***(pn + i) = 0;** ⇒ * (BFFFFDA0 + 2) = 0 ⇒ * (BFFFFDA8) = 0

BFFFFD98	**2**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**0**	*numeros*
BFFFFDA4	**0**	
BFFFFDA8	**0**	
BFFFFDAC	**????????**	

- **i= 3;**
- ***(pn + i) = 0;** ⇒ * (BFFFFDA0 + 3) = 0 ⇒ * (BFFFFDAC) = 0

BFFFFD98	**3**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**0**	*numeros*
BFFFFDA4	**0**	
BFFFFDA8	**0**	
BFFFFDAC	**0**	

- **i = 4;**

BFFFFD98	**4**	*i*
BFFFFD9C	**BFFFFDA0**	*pn*
BFFFFDA0	**0**	*numeros*
BFFFFDA4	**0**	
BFFFFDA8	**0**	
BFFFFDAC	**0**	

Ejercicio 3

Todos son equivalentes entre sí, ya que cuando el compilador encuentra un arreglo como parámetro formal es tratado como puntero.

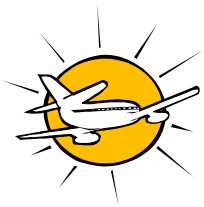
Obviamente el código dentro de la implementación de las funciones deberá tenerlo en cuenta, ya que debería saber hasta cuántas componentes debe acceder. Esto podría ser una convención de algún caracter especial como en el caso de los strings que "incrustan" el ASCII 0 para indicar terminación, o habría que pasar un parámetro extra.

Ejercicio 4

Todas compilan, ya que todas pasan como parámetro actual la dirección de un número entero.

Cabe señalar, que en el segundo caso, dependiendo cuál sea el objetivo de la función, **puede haber problemas en tiempo de ejecución** ya que ***nro*** es un **puntero a un entero pero no se ha inicializado** para que apunte a un entero válido antes de la invocación.

Ejercicio 5



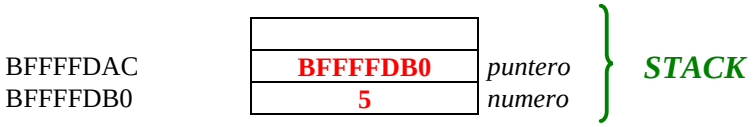
Por eficiencia. De no ser así se estarían copiando todas las componente del mismo en el stack, lo cual consumiría tiempo y espacio en cada invocación.

Ejercicio 6

a)

Salida

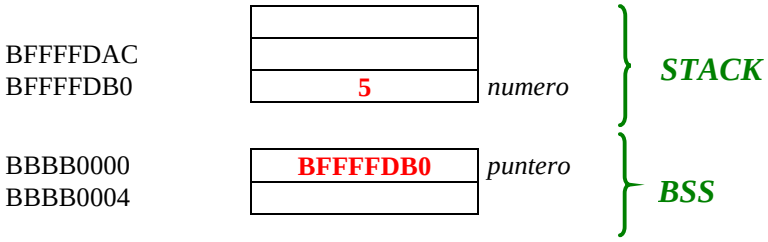
BFFFFDB0
BFFFFDAC
BFFFFDB0
5



b)

Salida

BFFFFDB0
BBBB0000
BFFFFDB0
5



c)

Ni siquiera compila (No puede completar la zona DATA)
"inicializer element is not constant"

El problema es que ***puntero*** es una variable static inicializa por el programa se debe almacenar en tiempo de compilación (se aloca en DATA), pero la dirección de la variable ***numero*** recién se va a conocer en tiempo de ejecución (cuando se aloque en el STACK)

Ejercicio 7

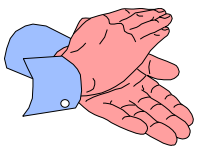
a)

HORRIBLE!

No se reservó lugar para leer la cadena que comienza en buffer.
La variable ***buffer*** (tipo puntero) se aloca en el stack, pero su contenido es desconocido (apunta acualquier zona de memoria). Luego, a través del uso de ***scanf***, el string que se ingrese desde la entrada estándar se guardará a partir de esa zona desconocida de memoria, destruyendo lo que hubiere en su lugar.

b)

CORRECTO!



La función ***main*** reservó lugar en la variable ***cartel*** para la lectura del string y le pasó el nombre del arreglo (r-value) a la función que hace la lectura. Cabe aclarar que el mismo efecto se hubiese logrado con la invocación de **scanf("%s", cartel)** desde ***main.***

Ejercicio 8

```
/*  Archivo iomatriz.h  */

#define DIM 50     /* maxima cantidad de filas o columnas  */

/* función que lee una matriz desde la entrada estándar */
void leeMatriz(float matriz[][DIM], int *cantFil, int *cantCol);

/* función que imprime una matriz por filas en salida estándar */
void imprimeMatriz(float matriz[][DIM],int cantFil, int cantCol);
```

```
/*  Archivo iomatriz.c  */

#include <stdio.h>
#include "getnum.h"
#include "iomatriz.h"


void
leeMatriz( float matriz[][DIM],  int *cantFil,  int *cantCol)
{
   int i,j;

   *cantFil= getint("\nIngrese la cantidad de filas:");

   *cantCol= getint("\nIngrese la cantidad de columnas:");


   for ( i= 0; i < *cantFil; i++)
      for ( j= 0; j < *cantCol; j++)
      {
         matriz[i][j] = getfloat("\nElemento m( %d, %d)=", i+1, j+1 );
      }
}


void
imprimeMatriz ( float matriz[ ][DIM],  int cantFil,  int cantCol)
{
   int i, j;

   for ( i= 0; i < cantFil; i++)
   {
      for ( j= 0; j < cantCol; j++)
         printf("%f\t", matriz[i][j]);
      printf("\n");
   }
}
```

sugerimos, como extensión, validar que cantFil y cantCol no superen la dimensión tope dad por DIM

```
/*  Archivo prodMatriz.c  */

#include <stdio.h>
#include “iomatriz.h”


int
main(void)
{
   float matA[DIM][DIM],  matB[DIM][DIM],  matC[DIM][DIM] ;
   int  cantFilA,  cantColA,  cantFilB,  cantColB;
   int i, j, k;

   leeMatriz( matA,  &cantFilA,  &cantColA);
   printf("Matriz A:\n");
   imprimeMatriz (matA, cantFilA, cantColA);

   leeMatriz( matB,  &cantFilB,   &cantColB);
   printf("Matriz B:\n");
   imprimeMatriz (matB, cantFilB, cantColB);


   if (cantColA  != cantFilB)
      printf ( "No se puede hacer el producto\n");
   else
   {
      for (i=0;  i<cantFilA;  i++)
         for (j=0;  j<cantColB;  j++)
         {
            matC[i][j]= 0;
            for (k=0; k<cantColA; k++)     /* o bien k<cantFilB */
               matC[i][j] += matA[i][k] * matB[k][j];
         }

       printf("Matriz A*B:\n");
       imprimeMatriz (matC, cantFilA, cantColB);
   }

   return 0;
}
```

Muy Importante:

Notar que en la función ***leeMatriz***, que recibe desde quien la invoca el puntero a entero ***cantFil***, hubiera sido incorrecto hacer la lectura como ***cantFil=getint(“”)*** ya que en ese caso el dato leído hubiera quedado en el ***stack frame de leeMatriz***, perdiéndose al regresar a quien la invocó (en este caso ***main***).

¡¡¡¡¡ PENSARLO MUY BIEN, HASTA ENTENDERLO !!!!!

Aritmética de Punteros

Introducción

En el lenguaje C los punteros pueden participar de un número restringido de operaciones aritméticas, de asignación y de comparación. A continuación se describen los operadores que pueden aplicarse a los operandos punteros y para qué sirve su uso

1. Operaciones Aritméticas con Punteros

Sean las variable de tipo puntero: ***tipo * aPointer, * anotherPointer***

Sintaxis	*Significado*
aPointer ++ **++ aPointer**	La variable aPointer quedó apuntando a un elemento a la derecha más que el original
aPointer -- **-- aPoiner**	La variable aPointer quedó apuntando a un elemento a la izquierda más que el original
aPointer + numero	Se devuelve una variable temporaria de tipo puntero que está apuntando a ***numero*** cantidad de elementos a la derecha más que la variable aPointer
aPointer - numero	Se devuelve una variable temporaria de tipo puntero que está apuntando ***numero*** cantidad de elementos a la izquierda más que la variable aPointer
aPointer - anotherPointer	Se devuelve la cantidad de elementos que hay entre ambos punteros

Importante

Nótese que en todos lo casos cuando se suma o resta un número a una variable de tipo puntero, el numero indicado NO SE INTERPRETA como cantidad de bytes, sino como cantidad de elementos (donde el tamaño de elemento está dado por el tipo que se especificó al indicar el puntero).

Nótese además que cuando se restan dos punteros entre sí NO SE OBTIENE la cantidad de bytes entre ellos sino la cantidad de elementos que lo separan.

Muy Importante

No extender la idea de operatoria **entre punteros**. NUNCA se pueden sumar dos punteros (ni siquiera tiene sentido).

Ejercicio 1

Indicar qué se obtiene si se ejecuta el siguiente fragmento de código en una arquitectura de 32 bits donde el tamaño del double es 8 bytes y la dirección de la primera componente del arreglo se encuentra en $BFFFFDA8:

```
double arreglo[] = { 5.5, 6.6, 7.7 };
double  *puntero;

puntero= arreglo;

printf("El contenido de puntero es=%p\n", puntero);
printf("El elemento referenciado por puntero es=%g\n", *puntero);

puntero + 1;

printf("El contenido de puntero es=%p\n", puntero);
printf("El elemento referenciado por puntero es=%g\n", *puntero);

puntero++;
printf("El contenido de puntero es=%p\n", puntero);
printf("El elemento referenciado por puntero es=%g\n", *puntero);

(*puntero)++;
printf("El contenido de puntero es=%p\n", puntero);
printf("El elemento referenciado por puntero es=%g\n", *puntero);
```

Respuesta

```
El contenido de puntero es=BFFFFDA8
El elemento referenciado por puntero es=5.5
El contenido de puntero es=BFFFFDA8
El elemento referenciado por puntero es=5.5
El contenido de puntero es= BFFFFDB0
El elemento referenciado por puntero es=6.6
El contenido de puntero es= BFFFFDB0
El elemento referenciado por puntero es=7.6
```

Nótese que la expresión puntero + 1 devuelve un puntero temporario, o sea que se pierde si no se lo hace participar de otra expresión.

Véase además que la expresión (*puntero)++; permitió la modicación de la segunda componente del arreglo: primero se desreferenció *puntero* y después se incrementó al elemento.

Pregunta

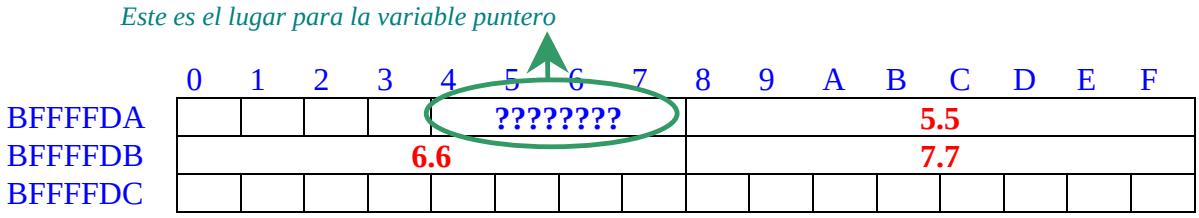
Hubiera sido lo mismo evaluar la expresión **puntero++* y evitar el uso de paréntesis? Explicar.

Respuesta

No, por la precedencia de operadores y asociatividad. En este caso al no tener paréntesis hay que observar cual de los dos operadores se aplica primero: tienen la misma precedencia. Entonces hay que resolverlo con la asociatividad que es de derecha a izquierda. Así es como al evaluar la expresión tenemos que se realiza un post-incremento del puntero y la desreferencia del mismo, pero como es una post-referencia su efecto ocurrirá al finalizar la evaluación de dicha expresión, obtenemos el valor apuntado por la variable *puntero* y como efecto colateral el puntero avanza luego a la componente siguiente.
O sea que de no haber usado paréntesis, los elementos del arreglo hubieran quedado intactos y como al resultado de la desreferencia no se lo usó en ninguna otra expresión, se hubiera obtenido el mismo efecto que hacer *puntero++*

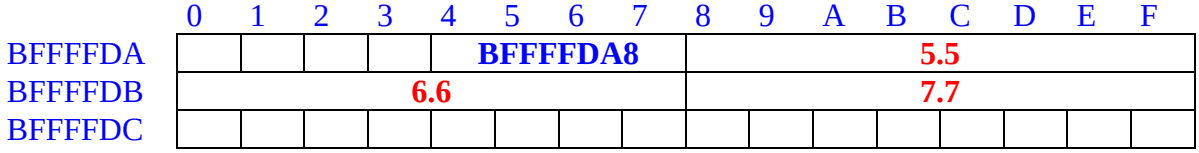
Ejercicio 2

Suponiendo que la memoria se encuentra al comenzar la ejecución del código anterior con los siguiente valores, indicar paso a paso como va modificándose el código del ejercicio 1 al correr la aplicación.



Respuesta

a) Después de **puntero= arreglo:**



b) Después de **puntero + 1** todo sigue igual.

c) Después de **puntero++:**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
BFFFFDA					**BFFFFDB0**				**5.5**							
BFFFFDB	**6.6**								**7.7**							
BFFFFDC																

d) Despues de **(*puntero)++**:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
BFFFFDA					**BFFFFDB0**				**5.5**							
BFFFFDB	**7.6**								**7.7**							
BFFFFDC																

Ejercicio 3

Indicar qué se obtiene si se ejecuta el siguiente fragmento de código en una arquitectura de 32 bits donde el tamaño del double es 8 bytes y la dirección de la primera componente del arreglo se encuentra en $BFFFFDA8:

```
double arreglo[] = { 5.5, 6.6, 7.7 };
double  *puntero= arreglo;

arreglo[2]= *++puntero;

printf("El contenido de arreglo[2] es=%g\n", arreglo[2]);
printf("El contenido de puntero es=%p\n", puntero);
printf("El elemento referenciado por puntero es=%g\n", *puntero);
```

Respuesta

```
El contenido de arreglo[2] es=6.6
El contenido de puntero es=BFFFFDB0
El elemento referenciado por puntero es=6.6
```

Muy Muy Importante

La aritmética de punteros no tiene significado a menos que se use con un puntero que se inicialice con la dirección de alguna componente de un arreglo, ya que solo las componentes de un arreglo son del mismo tipo y se almacenan en forma contigua o sea que tiene sentido "moverse" a la izquierda o derecha de sus componentes

Ejercicio 4

¿Qué se obtiene al ejecutar el siguiente fragmento de código?

```
double arreglo[] = { 5.5, 6.6, 7.7 };
double  *puntero1, *puntero2;

puntero1= arreglo;
puntero2= &arreglo[2];

printf("puntero2 - puntero1=%d\n", puntero2 - puntero1);
printf("puntero1 - puntero2=%d\n", puntero1 - puntero2 );
```

Respuesta

```
puntero2 – puntero1= 2
puntero1 – puntero2= -2
```

Pregunta

¿Se obtendría el mismo resultado si en el ejemplo anterior se realiza puntero1= &arreglo[0] en vez de puntero1= arreglo?

Respuesta

Sí, porque el nombre del arreglo es la dirección de la primera componente

Muy Importante

Es un error muy frecuente recorrer por medio de un puntero a arreglo la memoria, e irse fuera de los límites del mismo (o sea direccionar cualquier cosa)

2. Operaciones de Asignación con Punteros

Sean las variable de tipo puntero: ***tipo * aPointer, * anotherPointer***

Sintaxis	*Significado*
aPointer += numero	La variable aPointer quedó apuntando *numero* elementos más a la derecha que originalmente
aPointer –= numero	La variable aPointer quedó apuntando *numero* elementos más a la izquierda que originalmente
aPointer = anotherPointer	Asignar el contenido de la variable anotherPointer al contenido de la variable aPointer

Muy Importante

En ANSI C los tipos puntero a entero, puntero a double, puntero a char, etc. son distintos, por lo cual un puntero puede ser asignado a otro si **son del mismo tipo, o alguno de ellos es de tipo puntero a void.**

En el resto de los casos un puntero puede ser convertido en otro usando explícitamente el **operador cast** pero pueden producirse efectos no deseados. La impredicibilidad se debe a los alineamientos.

El uso de **punteros a void** lo veremos en Estructuras de Datos y Algoritos.

Recordar

- El nombre de un arreglo es idéntico a la dirección de su primera componente
- Cuando una función declara un parámetro con sintaxis de arreglo:

  void ordena(double vector[], int dim);

  al invocarla con **double arreglo[]= { 5.5, 6.6, 7.7 };**
  ordena(arreglo, 3);

  en realidad lo que se está pasando como parámetro actual es la **dirección de la primera componente del arreglo**. El parámetro formal tiene aspecto de arreglo pero es un puntero, o sea que el prototipo podía haberse declarado así:

  void ordena(double* vector, int dim);

  Para cualquiera de las dos declaraciones se puede usar aritmética de punteros porque el *parámetro formal vector* **es un puntero.**

A *la variable arreglo* NO se le puede aplicar aritmética de punteros porque NO es un puntero, NO es un l_value (algunos lo consideran un l_value constante).

Aparte de esta restricción, las variables de tipo puntero y de tipo arreglo se pueden usar en expresiones de manera que parezcan intercambiables.

- Siendo p un puntero y k un numero entero:
  - la expresión *(p+k) es equivalente a p[k].
  - la expresión p + k es equivalente a &p[k]

  Por lo tanto es frecuente inicializar una variable puntero con la dirección de alguna componente de un arreglo y luego subindicarlo para acceder a otras componentes del arreglo (de ahí que se diga que los punteros y los arreglos son prácticamente intercambiables, ya que participan de formas sintácticas similares).

Ejemplo:

Indicar qué imprime el siguiente programa, suponiendo que la dirección de la primera componente del arreglo es $BFFFFDA8 y la arquitectura es de 32 bits con tamaño de double de 8 bytes

```
#include <stdio.h>

void sorpresa(double * vector)
{
   printf("parámetro vector apunta a la dirección=%p\n", vector);

   printf(   "desrefencio el tercer elemento "
             "con notación puntero desplazamiento=%g\n", *(vector+2));

   printf(   "desrefencio el tercer elemento "
             "con notación puntero subindice=%g\n",  vector[2] );
}


int
main(void)
{
   double arreglo[] = { 5.5, 6.6, 7.7 };

   sorpresa( arreglo );
   return 0;
}
```

Respuesta

```
Parámetro vector apunta a la dirección BFFFFDA8
Desreferencio el tercer elemento con notación puntero desplazamiento=7.7
Desreferencio el tercer elemento con notación puntero subindice=7.7
```

3. Operaciones de Comparación con Punteros

Sean dos variables de tipo puntero: ***tipo * aPointer, * anotherPointer;***

Sintaxis	*Significado*
aPointer == anotherPointer **aPointer != anotherPointer**	Se compara los contenidos de las variable apuntadoras (para saber si apuntan o no al mismo lugar)
aPointer < anotherPointer **aPointer <= anotherPointer** **aPointer > anotherPointer** **aPointer >= anotherPointer**	Se comparan los contenidos de las variables apuntadoras para saber cual de ellos apuntan a direcciones menores o mayores de memoria.

Importante

Las comparaciones entre punteros no tienen ningún sentido, a menos que los punteros señalen a miembros del mismo arreglo.

Una comparación de dos punteros que señalen al mismo arreglo podría servir para no irse fuera de los límites del arreglo.

Uso muy común: **if (p != NULL)** o bien **if (p)**

Ejercicio 5

Re-escribir la función strlen de la librería estándar, utilizando aritmética de punteros para recorrer el arreglo de caracteres.

Respuesta

Para saber si llegué al fin del recorrido no es necesario que reciba como parámetro la cantidad de componentes, siempre que el string sea NULL terminated, pues ya tiene un caracter incrustado al final (‘\0’ o ASCII 0) para saber que allí finaliza.

*Recordar que la mayoría de las funciones sobre string de la librería estándar asumen que se recibe un arreglo de caracteres NULL terminated y por eso **no necesita recibir** la cantidad de caracteres reales del mismo.*

```
int  strlen( char* string)
{
   int cantidad= 0;
   while ( *string++ )
       cantidad++;

   return cantidad;
}
```

Lo nulo es lo apuntado por el puntero, por eso desreferenciamos

Pregunta

¿Se puede invocar con un string constante, como se indica a continuación?

```
int
main(void)
{
        printf("%d\n", strlen ( "hola") );
        return 0;
}
```

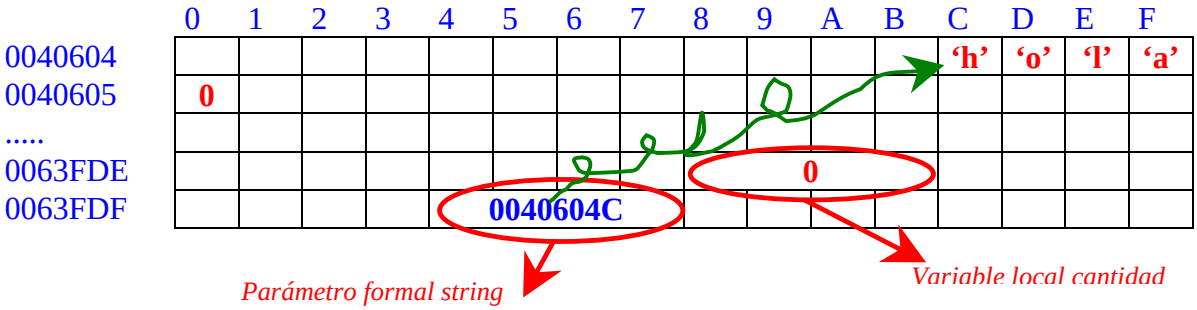
Respuesta

Sí, porque lo que se copia en el parámetro formal es la dirección de la primera componente del arreglo. Cuando se aplica aritmética de punteros al parámetro formal no se está cambiando la dirección de la primera componente del arreglo original. Las componentes del arreglo tiene direcciones de memoria, pero no pueden moverse por la memoria. El puntero que sirve para recorrer el arreglo sí puede ir cambiando su contenido para referenciar en cada instante distintas componentes del arreglo.

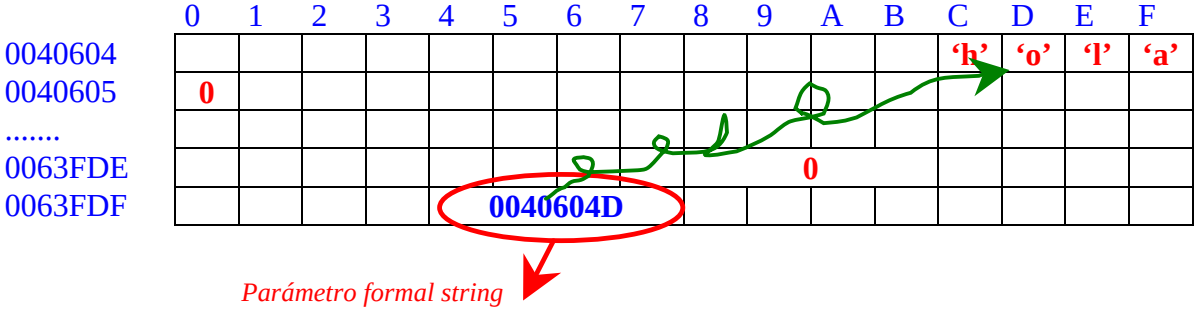
Ejemplo

A continuación vamos a graficar la memoria, paso a paso, durante la ejecución de la función ***strlen*** suponiendo que el string constante **"hola"** se encuentra cargado a partir de la dirección $0040604C, y que las variables ***cantidad*** y ***string*** se encuentran en las direcciones $0063FDE8 y $0063FDF4 respectivamente

- Antes del **while**:



- Al evaluar **(*string++)** se obtiene **'h'** y como efecto colateral avanza el puntero, o sea que por solo evaluar eso se modifica la memoria:



- Dentro del **while** se evalua **cantidad++** , o sea que la memoria cambia a:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**1**							
0063FDF					**0040604D**											

- Al evaluar **(*string++)** se obtiene **'o'** y como efecto colateral avanza el puntero, o sea que por solo evaluar eso se modifica la memoria:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE										**1**						
0063FDF					**0040604E**											

Parámetro formal string

- Dentro del **while** se evalúa **cantidad++**, o sea que la memoria cambia a:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**2**							
0063FDF					**0040604E**											

- Al evaluar **(*string++)** se obtiene **'l'** y como efecto colateral avanza el puntero, o sea que por solo evaluar eso se modifica la memoria:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE										**2**						
0063FDF					**0040604F**											

Parámetro formal string

- Dentro del **while** se evalúa **cantidad++**, o sea que la memoria cambia a:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**3**							
0063FDF					**0040604F**											

- Al evaluar **(*string++)** se obtiene **'a'** y como efecto colateral avanza el puntero, o sea que por solo evaluar eso se modifica la memoria:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**3**							
0063FDF					**00406050**											

Parámetro formal string

- Dentro del **while** se evalúa **cantidad++**, o sea que la memoria cambia a:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**4**							
0063FDF					**00406050**											

- Al evaluar **(*string++)** se obtiene **'\0'** o sea que no entra en el cuerpo del while. Como efecto colateral igualmente avanza el puntero:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040604													**'h'**	**'o'**	**'l'**	**'a'**
0040605	**0**															
.....																
0063FDE									**4**							
0063FDF					**00406051**											

4. Arreglo de Punteros

Si quisiéramos armar una colección de strings tendríamos dos posibilidades:

- armar una matriz bidimensional de caracteres
- armar un arreglo de punteros a caracteres

La primera aproximación es un desperdicio de memoria ya que no todos los string serán de la misma longitud. Así es como habrán muchos elementos del mismo no se utilizarán, pero se habrá reservado espacio para ellos. El hecho de tener un numero fijo de columnas de por lo menos la longitud de la cadena más larga es un desperdicio, especialmente cuando las cadenas tiene longitudes variadas.

La segunda idea resulta mucho más tentadora.

Ejemplo

Veamos la declaración del arreglo de cadenas de caracteres **palo,** que puede servir para representaron un mazo de naipes.

char *palo[ ] = {"Copa", "Basto", "Oro", "Espada"};

Cada una de las cadenas tiene distinta longitud, pero lo único que guarda cada una de las componentes del arreglo palo son punteros a donde se encuentran dichos strings. No guardan sus caracteres (no es una matriz bidimensional)

Ejercicio 6

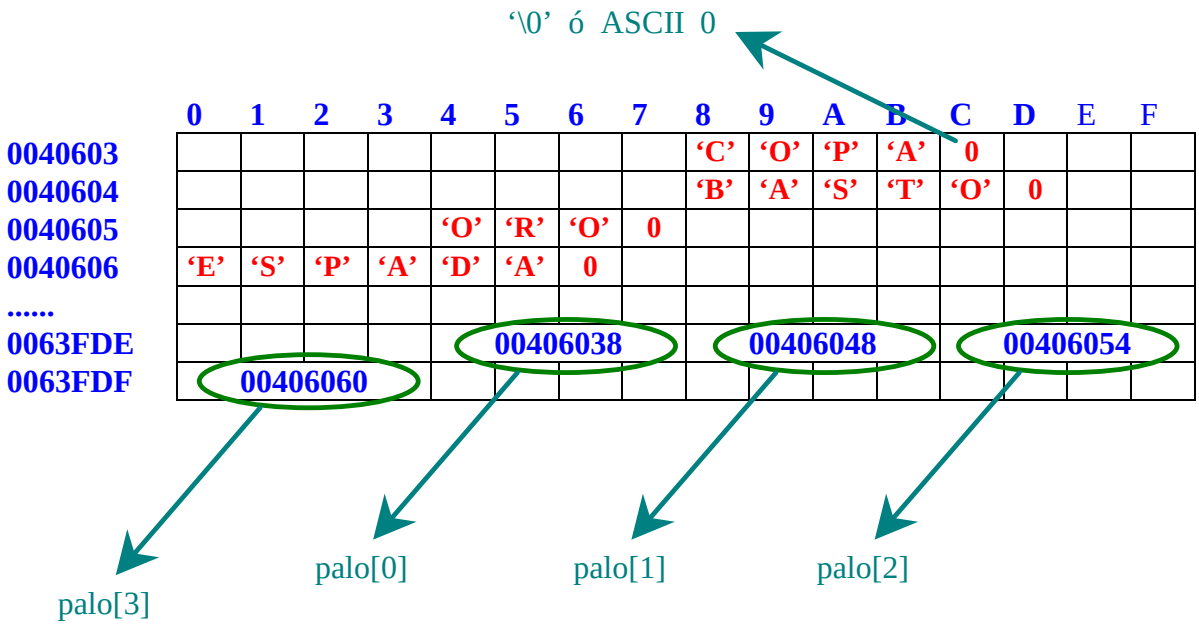
Escribir un programa que primero imprima qué direcciones contiene cada una de esas componentes (para saber dónde están dichos strings) y luego imprima los strings

Respuesta

```
int
main(void)
{
   char *palo[ ] = {"Copa", "Basto", "Oro", "Espada"};
   int i;

   for (i = 0; i < sizeof(palo)/sizeof(palo[0]); i++)
       printf("%p\n", palo[i] );

   for (i = 0; i < sizeof(palo)/sizeof(palo[0]); i++)
       printf("%s\n", palo[i] );

   return 0;
}
```

Notar que no hay que desreferenciar porque el printf con formato %s espera la dirección de comienzo del string y no el primer caracter

Si dibujamos la distribución en memoria, una posibilidad sería:



Biblioteca Estándar – Parte II

Introducción

En este documento presentaremos una serie de funciones se ofrecen en la Biblioteca Estándar de C y que no tratamos anteriormente porque involucraban punteros.

1. La Biblioteca Estándar para el Sistema (Standard System Library)

Los prototipos de sus funciones se encuentran en **stdlib.h**

prototipo	descripción
double atof(const char *s);	Convierte la cadena s a *double*. Trunca la conversión en el primer caracter no convertible a número.
int atoi(const char *s);	Convierte la cadena s a *int*. Trunca la conversión en el primer caracter no convertible a número.
long atol(const char *s);	Convierte la cadena s a *long*. Trunca la conversión en el primer caracter no convertible a número.

Ejemplos de invocación:

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

int
main(void)
{
      char num[10]="23092.654";
      char numBad[10]="230G.654";

      printf("cadena como doble= %f \n", atof(num) );      ®   23092.654
      printf("cadena como entero =%d \n", atoi(num) );      ®   23092
      printf("cadena como entero =%ld \n", atol(numBad) ); ®   230

      return 0;
}
```

2. *La Biblioteca Estándar Matemática (Math Library)*

Los prototipos de sus funciones se encuentran en **math.h**

prototipo	descripción
double frexp(double x, int *exp);	Regresa en su nombre la fracción normalizada [1/2, 1] de **x**, y en *exp la potencia de 2. Si x es cero, ambas partes se devuelven en cero.
double modf(double x, double *ip);	Regresa en su nombre la parte fraccionaria de **x** y en *ip la parte entera (ambas con el signo de x)

Ejemplos de invocación:

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

int
main(void)
{
      double  p_frac , p_ent , num= 1.005;
      int expon;


      p_frac= modf(num, &p_ent);
      printf("parte entera:%f \t parte frac:%f \n", p_ent ,  p_frac);

                                                    1.0       0.005
      p_frac= frexp(num, &expon);
      printf("numero: %f \t 2^ %d\ n",  p_frac , expon);

                                        0.5025        1

                                              1.005 = 0.5025 * 2^1
      return 0;
}
```

3. La Biblioteca Estándar para Cadenas (Math Library)

Estas funciones sirven para el manejo de cadenas de caracteres "null terminated". Los prototipos de sus funciones se encuentran en **string.h**

prototipo	descripción
char *strcpy (char *sd, const char * sf);	Copia **sf** en **sd** y devuelve **sd**.
char *strncpy (char *sd, const char * sf, *int* n);	Copia n caracteres de **sf** en **sd** y devuelve **sd**.
char *strcat (char *sd, const char * sf)	Devuelve **sd** con **sf** concatenado al final
char *strncat (char *sd, const char * sf, *int* n);	Devuelve **sd** con **n** caracters de **sf** concatenados al final
int strcmp (const char *sd, const char * sf);	Compara **sf** con **sd**. Si **sd** < **sf**, devuelve un número < 0 Si **sd** > **sf**, devuelve un número > 0 Si **sd** == **sf**, devuelve 0
int strncmp (const char *sd, const char * sf , *int* n);	Compara **n** caracteres de **sf** con **sd**. Devolución: ídem a ***strcmp***
char *strchr (const char *sd, char c);	Devuelve un puntero a la primera aparición del caracter **c** en **sd**. Si no existe, devuelve NULL.
char *strrchr (const char *sd, char c);	Devuelve un puntero a la última aparición del caracter **c** en **sd**. Si no existe, devuelve NULL.
char *strpbrk (const char *sd, const char * sf);	Devuelve un puntero a la primera aparición en **sd** de cualquier caracter de **sf** . Si no lo hay devuelve NULL.
char *strstr (const char *sd, const char * sf);	Devuelve un puntero a la primera aparición del substring **sf** en **sd**. Si no la hay devuelve NULL.

Ejemplos de invocación:

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


int
main(void)
{
     char cad1[15], cad2[20];
     char *xx;   int compare;


     strcpy(cad2, "proof");
     xx= strcpy(cad1,cad2);
     printf("&cad1=%p \n &cad2=%p \n  xx=%p \n" , cad1, cad2, xx);
     printf("cad1: %s \n cad2: %s \n", cad1, cad2);
```

proof proof contiene la dirección de cad1

```
     strcpy(cad2,"proof");
     xx= strncpy(cad1, cad2, 3);
     cad1[3]=0;
     printf("&cad1=%p \n &cad2=%p \n xx=%p \n", cad1, cad2, xx);
     printf("cad1:%s \n cad2: %s\n", cad1, cad2);
```

pro proof contiene la dirección de cad1

```
     strcpy(cad1,"casa");
     strcpy(cad2,"blanca");
     strcat(cad1,cad2);
     printf("&cad1=%p \n &cad2=%p \n", cad1, cad2);
     printf("cad1:%s \ncad2:%s \n", cad1, cad2);
```

casablanca blanca

```
     strcpy(cad1,"casa");
     strcpy(cad2,"blanca");
     strncat(cad1,cad2,4);
     printf("&cad1=%p \n &cad2=%p \n", cad1, cad2);
     printf("cad1:%s \ncad2:%s \n", cad1, cad2);
```

casablan blanca

```
      strcpy(cad1,"casa");
      strcpy(cad2,"Casa");
      if ( (compare= strcmp(cad1,cad2)) == 0)
            printf("%s y %s Son iguales\n", cad1, cad2);
      else
            if (compare>0)
                  printf("%s > %s \n", cad1, cad2);
            else
                  printf("%s < %s \n", cad1, cad2);
```

casa > **Casa**

```
      strcpy(cad1,"casa");
      strcpy(cad2,"casas");
      if ( (compare= strncmp(cad1,cad2,2)) == 0)
            printf("%s y %s tienen igual el prefijo %d\n", cad1, cad2,
                                                                  2);
      else
            if (compare>0)
                  printf("%s > %s en su %d prefijo\n", cad1, cad2, 2);
            else
                  printf("%s < %s en su %d prefijo\n", cad1, cad2, 2);
```

casa y casas tienen igual el prefijo 2

```
      strcpy(cad1,"abracadabra");
      xx= strchr(cad1,'r');
      printf("&cad1=%p \n cad1=%s letra=%c\n xx=%p \n",
                                          cad1, cad1, 'r', xx);
```

abracadabra

suponiendo que ***cad1*** *tiene la dirección* ***BFFFFFA0,***
XX *quedaría apuntando a* ***BFFFFFA2***

```
      return 0;
}
```

4. La Biblioteca Estándar para Entrada/Salida (Standard I/O Library)

Los prototipos de sus funciones se encuentran en el archivo de encabezamiento **stdio.h.**

prototipo	descripción
int scanf(const char * formato, ...);	Obtiene datos desde el flujo stdin (entrada estándar), con el formato pedido. Devuelve EOF si ocurre un error en la entrada antes de cualquier conversión o el número de ítems que se llegaron a asignar.
int sscanf(const char * s, const char *formato, ...);	Obtiene datos desde el string "null terminated" **s**, con el formato pedido. Se detiene si encuentra el caracter nulo.

Estas funciones son similares a ***printf***, pero proveen funcionalidad para lectura de valores desde una entrada (estándar o un string). Son útiles para la lectura de valores de tipos simples.

Desarrollaremos la función ***scanf,*** ya que ***sscanf*** es idéntica a ella, con la sola diferencia de que los datos los toma del string indicado en el primer parámetro, en vez de tomarlos desde la entrada estándar.

En forma análoga a *printf*, usa un string de control donde se le especifica en que formato se debe convertir los caracteres leídos de la entrada estándar para almacenarlos en memoria. Sin embargo, hay que tener cuidado por que la apariencia similar con los formatos de ***printf,*** en muchos casos es sólo superficial.

El primer argumento de ***scanf*** es un string "null terminated"-"read only", llamado **cadena de formato** y a continuación la lista de punteros a los datos en donde se desea almacenar cada campo de la entrada.

Como la función guarda los valores convertidos en los parámetros provistos a continuación del string de control, obviamente los mismos deben ser direcciones de memoria que indiquen el lugar donde se desea guardar el dato convertido. Si no fuera así , la función no podría retornar por medio de sus parámetros ningún cambio.

La **cadena de formato** le indica a la función qué argumentos adicionales hay y cómo convertir los datos ingresados a los valores que deben ser almacenados (típicamente los argumentos son punteros al tipo de dato esperado). También especifica cualquier texto que deba coincidir entre los campos convertidos.

La **cadena de formato** contiene especificaciones de conversión que se usan para interpretar la entrada:

- **caracteres de espaciado (blanco, \t, \n, etc.)**: su efecto es saltear de la entrada estándar todos los caracteres de espaciado que aparezcan
- **caracteres ordinarios:** a diferencia de los anteriores, deben coincidir exactamente con los caracteres de la entrada estándar
- **especificaciones de conversión:** causan la conversión de los campos de entrada

Las especificaciones de conversión comienzan con el símbolo % y están compuestas por las siguientes opciones:

- el símbolo **asterisco**, opcional. Si el mismo está presente indica que el valor leído con el formato pedido no debe almacenarse a través del argumento puntero.
  Por ejemplo **%*s** saltea todo caracter que no es blanco
- un número opcional para indicar la máxima cantidad de caracteres a ser leídos para luego aplicarles la conversión. Se considera ***unsigned int***
  Por ejemplo, **%5i**
- la letra que especifica conversión. Si es un entero puede ser precedida por la letra h para indicar que es short, o l para indicar que es long. Si es un double debe indicarse con la letra L antes del caracter correspondiente de punto flotante.

Las especificaciones de conversión comienza siempre con % y terminan con un carácter que indica la conversión deseada, de acuerdo al siguiente cuadro:

Símbolo	Tipo de Argumento	Descripción
d	Puntero a ***int***	Convierte a un int en base 10
hd	Puntero a ***short***	Idem pero convierte a short
ld	Puntero a ***long***	Idem pero convierte a long
u	Puntero a ***unsigned int***	Convierte a un unsigned int en base 10
hu	Puntero a ***unsigned short***	Idem pero convierte a unsigned short
lu	Puntero a ***unsigned long***	Idem pero convierte a unsigned long
i	Puntero a ***int***	Convierte a un entero en la base especificada por el input (0x) u octal (0)
hi	Puntero a ***short***	Idem pero convierte a short
li	Puntero a ***long***	Idem pero convierte a long
o	Puntero a ***unsigned int***	Convierte a un entero en base 8
ho	Puntero a ***unsigned short***	Idem pero convierte a short
lo	Puntero a ***unsigned long***	Idem pero convierte a long

Símbolo	Tipo de Argumento	Descripción
x, X	Puntero a ***unsigned int***	Convierte a un entero en base 16
hx, hX	Puntero a ***unsigned short***	Idem pero convierte a short
lx, lX	Puntero a ***unsigned long***	Idem pero convierte a long
e, f, g, E, G	Puntero a ***float***	Convierte a float
le, lE, lf, lg, lG	Puntero a ***double***	Convierte a double
Le, LE, Lf, Lg, LG	Puntero a ***long double***	Convierte a long double
c	Puntero a caracter	Almacena n caracteres (como arreglo), por omisión es un 1
s	Puntero a ***char***	Convierte los caracteres, hasta que aparezca un caracter de espaciado, en el arreglo especificado, como null terminated
[...]	Puntero a ***char***	Coincide con la mayor cadena no vacía de caracteres que pertenezcan al conjunto especificado entre corchetes. Se pueden colocar rangos separando el primer elemento del último con un guión.
[^...]	Puntero a ***char***	Coincide con la mayor cadena no vacía de caracteres que NO pertenezcan al conjunto especificado entre corchetes. Se pueden colocar rangos.

La función retorna la cantidad de conversiones realizadas en forma correcta que fueron almacenadas (no se cuentan la conversiones salteadas por el uso del caracter *)

CONSEJO

No finalizar la cadena de formato con caracteres que no sean especificación de conversión (blanco, \n, \t, A, 6, etc.)

Nota Importante

Notar que, aunque la función ***scanf*** devuelve el número de ítems que pudieron ser asignados, regresando cero si no se llegó a leer ninguno. Sin embargo, en caso de encontrar un error (como por ejemoplo que se temina el archivo antes de leer) devuelve **EOF**, con lo cual ya no se sabe cuántos fueron exitosamente almacenados. Si se quiere tener un control estricto, hay que dividir el ingreso en múltiples llamadas a ***scanf***.

Cuando ***scanf*** obtiene algún caracter inesperado, lo vuelve a colocar en el flujo de entrada. En este sentido e similar al ungetc, pero la diferencia es que ungetc garantiza sólo un caracter de vuelta, es decir si se usan varios ungetc seguidos no es seguro que se coloquen todos los caracteres en la entrada. En cambio, con sucesivos *scanf*, donde cada uno vuelve a la entrada estándar el caracter no reconocido, es seguro que vuelven todos dichos caracteres.

Por ejemplo, supongamos que ***scanf*** está esperando un valor ***float*** y llega **123EASY.** Aunque el subcampo **123E** resulta válido, la conversión requiere al menos un dígito de exponente. Entonces **123E** es consumido, pero la conversión falla. NO se almacena valor alguno (123E se pierde) y la función ***scanf*** retorna. El próximo caracter a leer desde la entrada es **A**. Este problema es típico de un ***float*** o ***double***, en otros tipos esto generalmente no ocurre.

Recomendamos leer detenidamente la descripción de *scanf*
en las secciones 7.4 y B1.3 de Kernighan & Ritchie
(El Lenguaje de Programación C).

Ejemplos de invocación:

```
#include <stdio.h>


int
main(void)
{

   char cad1[15], cad2[20];
   int d,m,a, edad;   int cant;


   printf("ingrese fecha[dd/mm/aa]:");
   cant= scanf("%d / %d / %d", &d,  &m,  &a);
   printf("cant=%d -> dia=%d\t mes=%d\t anio=%d\n", cant, d, m, a);

   printf("ingrese edad:");
   cant= scanf("%d", &edad);
   printf("cant=% d-> edad=%d \n", cant, edad);

   printf("ingrese un número:");
   scanf("%[0123456789]  %[0-9]", cad1, cad2);

   printf("ingrese la patente de su auto:");
   scanf("%[0-9a-zA-Z]", cad1);

   printf("ingrese su número de cuit:");
   scanf("%[0-9 -]", cad1);

   printf("ingrese su número de dirección de e-mail:");
   scanf("%[a-zA-Z.@]", cad1);

   printf("ingrese un número entero:");
   scanf("%[^a-zA-Z.]", cad1);

   return 0;
}
```

5. Ejercicios de Aplicación

Ejercicio 5.1

Escribir una función ***invertString*** que invierta los caracteres de un *string null terminated* (parámetro de entrada) y los deje en un arreglo de caracteres (parámetro de salida) que debe contar con cantidad suficiente resevada para alocar los caracteres del primer parámetro, además del cero final.

Ejemplo:

Si se tiene

char *dato = "HOLA - CHAU!";
char cambio[20];

invertString(dato, cambio);

en **cambio** debe quedar **"!UAHC - ALOH"**

Respuesta

```
void
invertString( const char *fuente,  char *destino)
{
      int cant;

      cant = strlen(fuente);

      /* se recorre la fuente de atrás hacia adelantye */
      while( cant )
            *destino++ = *(fuente + --cant);

      *destino = 0;
}
```

Ejercicio 5.2

Escribir una función ***invertNombre*** que invierta el orden entre nombre y apellido de un *string null terminated* (parámetro de entrada) donde cada nombre está separado por un espacio en blanco y el último de todos ellos es el apellido. La inversión debe quedar en un segundo arreglo de caracteres (parámetro de salida) de manera tal que después del apellido aparezca una coma y luego los nombres en el orden correcto. En un tercer parámetro se recibe la máxima cantidad reservada para el parámetro de salida, sin incluir el lugar para el '\0'.

Si la inversión se realiza con éxito, la función debe retornar en su nombre un 0, caso contrario debe devolver un 1.

Se cuenta con una función ***trim*** que recibe un *string null terminated* (parámetro de entrada-salida) y lo retorna habiéndole sacado todos los blancos de adelante y de atrás (su código se muestra en el ejercicio 5.3)

Ejemplo:

Si se tiene

```
char  *nombre = "Maria Laura Santillan";
char   newNombre[30];
int   error;

error = invertNombre(  nombre, newNombre, 30);
```

en **newNombre** debe quedar **"Santillan, Maria Laura"** y en **error** queda 0;

Respuesta: Versión Estructurada (una sola salida de la función)

```
int
invertNombre( char *fuente,  char *destino, int maximo )
{
      int cant, error = 0;
      char *st;

      trim(fuente);
      cant = strlen(fuente);

      st = strrchr(fuente, ' ');          /* busco el ultimo blanco */

      if (st == NULL)                     /* hay un solo nombre */
      {
            if (cant <= maximo)
                  strcpy(destino, fuente);
            else
                  error = 1;
      }
      else
      {
            cant++;                    /* agrego lugar para la coma */
            if (cant <= maximo)
            {
                  strcpy(destino, st + 1);
                  strcat(result,  ", ");
                  strncat(destino, fuente, st – fuente);
            }
            else
                  error = 1;
      }

      return  error;
}
```

Respuesta: Versión NO Estructurada (varias salidas de la función)

```
int
invertNombre2( char *fuente,  char *destino, int maximo )
{
      int cant;
      char *st;

      trim(fuente);
      cant = strlen(fuente);

      st = strrchr(fuente, ‘ ’);          /* busco el ultimo blanco */

      if (st == NULL)                     /* hay un solo nombre */
      {
            if (cant > maximo)
                  return 1;

            strcpy(destino, fuente);
      }
      else
      {
            cant++;                     /* agrego lugar para la coma */
            if (cant > maximo)
                  return 1;

            strcpy(destino, st + 1);
            strcat(result,  “, ”);
            strncat(destino, fuente, st – fuente);
      }

      return  0;
}
```

Ejercicio 5.3

Escribir una función ***IntValido*** que devuelva en su nombre un número entero (long) leído desde la entrada estándar, correctamente validado como entero, y en un parámetro de salida el valor 0 si el ingreso fue correcto o 1 en caso de error. Cuando el parámetro de salida indique error, la función devuelve en su nombre el valor 0.

Ejemplo:

Si se tiene

```
int   error;
long   num;

num  = intValido( &error);
```

- Al ingresar **“3572”** en **error** queda 0 y en **num** queda el entero **3572**
- Al ingresar **“+3572”** en **error** queda 0 y en **num** queda el entero **3572**

- Al ingresar **"-3572"** en **error** queda 0 y en **num** queda el entero -**3572**
- Al ingresar **"35.72"** en **error** queda 1 y en **num** queda el entero **0000**
- Al ingresar **"3e-5"** en **error** queda 1 y en **num** queda el entero **0000**
- Al ingresar **"364m"** en **error** queda 1 y en **num** queda el entero **0000**

Respuesta:

```
long
intValido( int *error )
{
   char stNum[100];
   char *pstNum;
   int n, c;

   *error = 1;

   pstNum = fgets (stNum, 100, stdin);

   if (pstNum)
   {
      pstNum[strlen(pstNum)-1 ]= 0;    /* tapamos el '\n' con '\o' */

      trim(pstNum);          /* eliminamos blancos adelante y atrás */


      if (strlen(pstNum) != 0)
      {

         /* solo puede haber signo al comienzo */
         if ( *pstNum == '-' || *pstNum == '+' )
            pstNum++;

         /* dentro del numero solo se permiten digitos */
         while( isdigit(*pstNum) )
            pstNum++;

         /* la unica salida correcta del ciclo es llegar al '\0'
            y tener ingresado algo mas que un signo */
         if ( *pstNum==0  &&  *(pstNum-1)!='-' && *(pstNum-1)!='+' )
            *error = 0;
      }
   }

   if (*error)
      return 0;

   /*  al llegar hasta aquí la cadena stNum solo tiene una secuencia
       valida de caracteres para representar un numero entero */
   return  atol(stNum);
}
```

Una posible versión para la función trim:

```
void
trim(char* pInit)
{
      char *pEnd= pInit + strlen(pInit) -1;
      char *pRec;


      /* colocamos pRec apuntando al primer caracter no blanco */
      pRec= pInit;
      while( pRec <= pEnd && isspace(*pRec) )
            pRec++;

      /* colocamos pEnd apuntando al ultimo caracter no blanco */
      while( pRec <= pEnd && isspace(*pEnd) )
            pEnd--;

      /* movemos los caracteres centrales hacia el comienzo */
      while( pRec <= pEnd )
            *pInit++= *pRec++;

      /* como pInit queda apuntando afuera de la cadena copiada,
         la transformamos en null terminated  */
      *pInit= 0;
}
```

A continuación trataremos de implementar otra versión, sin utilizaR el puntero auxiliar pRec

¿Por qué motivo NO FUNCIONA la siguiente versión?

```
void
trim(char* pInit)
{
      char *pEnd= pInit + strlen(pInit) -1;


      /* colocamos pInit apuntando al primer caracter no blanco */
      while ( pInit <= pEnd && isspace(*pInit) )
            pInit++;

      /* colocamos pEnd apuntando al ultimo caracter no blanco */
      while ( pInit <= pEnd && isspace(*pEnd) )
            pEnd--;

      /* transformamos la nueva cadena en null terminated */
      *(pEnd+1)= '\0';

}
```

NO FUNCIONA

Armado de Biblioteca con Uso de Punteros

Introducción

El objetivo de este apunte es presentar otro ejemplo de armado de una biblioteca (anteriormente habíamos armado la biblioteca **random**) con tratamiento explícito de códigos de error y con uso de parámetros de salida.

1. Propósito de la Biblioteca

Vamos a escribir una biblioteca INTERVAL de uso matemático que permita manejar operaciones con intervalos reales cerrados, determinados a través de dos números reales que representan sus extremos.

1.1. Funcionalidad de la Biblioteca INTERVAL

Para saber con qué funciones debe contar nuestra biblioteca, debemos detenernos a pensar las operaciones matemáticas básicas que se pueden realizar sobre intervalos de números reales:

- **Detectar si un par de números reales forman un intervalo válido**
  Por ejemplo, **[5.0 , 8.6]** es válido pero **[8.6 , 5.0]** no lo es.
  También podemos considerar válido al intervalo **[4.32 , 4.32]**.

- **Calcular la norma de un intervalo dado.**
  Por ejemplo, la norma de **[5.0 , 8.6]** es **3.6**.

- **Calcular cantidad de particiones de longitud dada que entran en un intervalo.**
  Por ejemplo, en **[5.0 , 8.6]** entran **6** particiones de longitud **0.6**.

- **Detectar si un número real pertenece o no a un intervalo dado.**
  Por ejemplo, **6.7** pertenece a **[5.0 , 8.6] ,** pero **8.9** no pertenece.

- **Calcular el intervalo intersección de otros dos intervalos dados.**
  Por ejemplo, la intersección entre **[5.0 , 8.6]** y **[7.5 , 10.4]** es **[7.5 , 8.6]**.

- **Trasladar un intervalo en un cierto desplazamiento dado.**
  Por ejemplo, al trasladar **[5.0 , 8.6]** en **3.5** unidades obtenemos **[8.5 , 12.1]**.

1.2. Prototipación de la Biblioteca INTERVAL

Antes de comenzar a escribir los códigos de las funciones propuestas en el ítem anterior, debemos diseñar los prototipos de cada función, determinando su tipo de devolución y los parámetros que debe recibir.

Otro punto muy importante es decidir el manejo de errores. Como todos sabemos la **programación defensiva es muy importante**, por lo cual no se debe dejar librado al azar la acción a seguir en el caso de que aparezca un valor inesperado. Aunque se tenga la esperanza de que el usuario vaya a hacer todas las validaciones posibles antes de cada invocación, siempre se debe contemplar el caso de recibir parámetros inválidos, **avisando en el archivo de encabezado la acción que se tomará ante dicha situación.**

Para nuestra implementación hemos decidido manejar los errores a través de devoluciones de **códigos de** error en el nombre de la función. De esta forma, si una función necesita retornar algún valor, deberá hacerlo en un parámetro de salida.

Por otra parte hemos decidido (puede haber otros diseños alternativos) que, si algún parámetro de entrada es inválido, además de devolver el código de error 0, los **parámetros de salida quedan con información incierta**. Obviamente, en caso de entradas inválidas, los parámteros de entrada-salida no se modifican.

Apuntando a la **homogeneidad de criterio**, todas las funciones deberán hacer el mismo tipo de tratamiento de error: No es bueno que algunas retornen error en un parámetro de salida y otras lo hagan en su nombre.

```
/*  Archivo interval.h
**  Autores:  G&G
**  Encabezamiento de la biblioteca para manejo de intervalos
*/


/*  Todas las funciones que reciben como parámetro un intervalo
**  real, lo hacen a través de sus extremos izquierdo y derecho
*/


/*  MUY IMPORTANTE
**  Si una función recibe un intervalo invalido, retorna en su
**  nombre código 0, no altera los parámetros de entrda-salida,
**  pero los parámetros de salida quedan seteados con valores
**  inciertos:
**
**  SI EL CÓDIGO DEVUELTO ES CERO,  NO USAR LA INFORMACIÓN DEL
**  PARÁMETRO DE SALIDA !!!
*/
```

```
/*  La siguiente función devuelve 1 si los extremos izquierdo y
**  derecho recibidos determinan un intervalo válido y 0 en caso
**  contrario
**  ------------------------------------------------------------
**  Ejemplo de uso:
**        if ( esInterval( 4.5, 6.7) )
**             ....
*/
int esInterval(double izq, double der);



/*  La siguiente función recibe como parámetro de entrada un
**  intervalo y como parámetro de salida un número real.
**  Si el intervalo recibido es válido, retorna 1 y coloca en el
**  parámetro de salida la norma del mismo. En caso contrario,
**  retorna  0 y el parámetro de salida queda con información
**  incierta
**  ------------------------------------------------------------
**  Ejemplo de uso:
**     double n;
**     if ( norma(3.8, 5.4, &n) )
**        printf("la norma es %f \n", n );
**     else
**        printf("intervalo no válido\n");
*/
int norma(double izq, double der, double *rta);



/*  La siguiente función recibe como parámetros de entrada un
**  intervalo y un número real que representa la longitud de una
**  partición, y como parámetro de salida un entero.
**  Si el intervalo recibido es válido, retorna 1 y coloca en el
**  parámetro de salida la cantidad entera de particiones que
**  entran en el intervalo. En caso contrario, retorna 0 y el
**  parámetro de salida queda con información incierta
**  ------------------------------------------------------------
**  Ejemplo de uso:
**
**     int n;
**     if ( cantParticion(3.8, 5.4, 0.4, &n) )
**        printf("cantidad de particiones: %d \n", n );
**     else
**        printf("intervalo no válido\n");
*/
int cantParticion(double izq, double der, double longitud,
                                                    int *rta);
```

```
/*  La siguiente función recibe como parámetros de entrada un
**  intervalo y un número real.
**  Si el intervalo recibido es válido, retorna 1 si el número
**  real pertenece al intervalo y 0 en caso contrario. Si no es
**  válido el intervalo recibido, retorna 0.
**  ------------------------------------------------------------
**  Ejemplo de uso:
**
**     printf("el numero %s al intervalo\n",
**              (pertenece(3.8, 5.4, 0.4) == 1)?"si":"no" );
*/
int pertenece(double izq, double der, double pto);



/*  La siguiente función recibe como parámetros de entrada dos
**  intervalos y como parámetros de salida un intervalo
**  resultante.  Si ambos intervalo recibidos son válidos,
**  retorna 1 si existe intersección entre ambos, en cuyo caso
**  el intervalo de salida contiene dicha intersección, y 0 si
**  no hay intersección o alguno de los intervalos de entrada no
**  son válidos, en cuyo caso el intervalo de salida queda
**  seteado con datos inciertos.
**  ------------------------------------------------------------
**  Ejemplo de uso:
**
**     int n, err;
**     double a, b;
**     if ( intersec(3.8, 5.4, &a, &b) )
**        printf("la intersección es  [%f, %f]\n", a, b);
*/
int  intersec(double izq1, double der1, double izq2,
                    double der2, double *izq3, double *der3);



/*  La siguiente función recibe como parámetros de entrada dos
**  intervalos.
**  Si ambos intervalo recibidos son válidos, retorna 1 si el
**  primer intervalo está incluido en el seguno y 0 en caso
**  contrario.
**  Si alguno de los intervalos no es válido retorna 0.
**  ------------------------------------------------------------
**  Ejemplo de uso:
**
**      printf("el primero %s está incluido en el segundo\n",
**         (estaIncluido( 3.8, 5.4, 0.4, 7.3) == 1)?"si":"no" );
*/
int estaIncluido(double izq1, double der1,
                                      double izq2, double der2);
```

```
/*  La siguiente función recibe como parámetro de entrada un
**  número real que representa un cierto desplazamiento, como
**  parámetro de   entrada-salida un intervalo a desplazar.
**  Si el intervalo recibido es válido, retorna 1 y modifica el
**  intervalo desplazándolo en el valor indicado. En caso
**  contrario, retorna 0 y el intervalo no se modifica.
**  ------------------------------------------------------------
**  Ejemplo de uso:
**
**    double  a= 5.7, b= 9.3;
**    if( traslada(0.4, &a, &b) )
**       printf("[5.7, 9.3] trasladado 0.4 es [%f,%f]°n", a, b );
**    else
**       printf("intervalo no válido\n");
*/
int traslada(double delta, double *izq, double *der);
```

1.3. Codificación de la Biblioteca INTERVAL

Respetando los prototipos dados en el encabezamiento, codificamos cada función.

```
/*  Archivo interval.c
**  Autores:  G&G
**  Biblioteca para manejo de intervalos
*/

#include <stdlib.h>
#include “interval.h”
#define max(a,b)      (a>b)?a:b
#define min(a,b)      (a<b)?a:b


int
esInterval(double izq, double der)
{
      return (izq <= der);
}

int
norma(double izq, double der, double *rta)
{
      *rta = der - izq;
      return  esInterval(izq, der);
}
```

```
int
cantParticion(double izq, double der, double longitud, int *rta)
{
      double  n;

      if ( longitud != 0 )
            if ( norma(izq, der, &n) )
                  *rta = n / longitud;

      return  esInterval(izq, der);
}



int
pertenece(double izq, double der, double pto)
{
      return (izq <= pto && pto <= der);
}



int
intersec(double izq1, double der1, double izq2, double der2,
                                        double *izq3, double *der3)
{
      *izq3= max(izq1, izq2);
      *der3= min(der1, der2);

      return  esInterval(*izq3, *der3);
}



int
traslada(double delta, double *izq, double *der)
{

      /* no se debe modificar un parámetro de entrada si hay
         un error en la entrada de datos*/

      if (*izq <= der)
      {
            *izq += delta;
            *der += delta;
      }

      return  esInterval(*izq, *der);
}
```

1.4. Ejemplo de Uso de la Biblioteca INTERVAL

Antes de distribuir nuestra biblioteca compilada, vamos a utilizarla para detectar posibles errores u omisiones.

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

int
main(void)
{
   int i;
   double izq = -2, der = -1, pto= -1.7;


   /* Forman intervalo los valores de izq y der ?   */
   printf("[%g,%g] %s intervalo válido\n", izq, der,
                           esInterval(izq,der)?"es":"no es");


   /* El valor –1.7, pertenece a dicho intervalo ? */
   for (i=1; i<=10; i++)
   printf("El punto %g %s pertenece a [%g,%g]\n", pto,
                   pertenece(izq,der,pto)?"":"no", izq, der+i);


   /* Se busca la intersección */
   printf ("Interseccion entre [%g,%g] y [%g,%g]) = ",
                                izq + 5, 3.0, izq,  der);


   if ( intersec( izq + 5, 3.0,  izq, der, &izq, &der) )
      printf( "[%g,%g]\n", izq, der);
   else
      printf( "vacio\n");


   /* Calculamos norma y cantidad de particiones */
   norma( izq, der, &i);
   cantParticion( izq, der, longitud, &i);
   printf("Norma de [%g,%g] = %g\n", izq, der, i);
   printf("Cant de particiones de longitud %f en [%g,%g]= %d\n",
                                          longitud, izq, der, i);

   /* Trasladamos un intervalo */
   printf ("Trasladamos [%g,%g] en %f = ", izq, der, long);
   traslada(longitud, &izq, &der);
   printf([%f, %f]\n", izq, der);

   return 0;
}
```

Recursividad - Parte I

Introducción

En este documento se explica la técnica de programación llamada ***recursividad***, su alcance y cuando resulta conveniente usarla.

1. Técnica Recursiva para Solucionar Problemas

Se trata de una técnica por la cual un problema a resolver, debido a su magnitud, es dividido en subproblemas del **mismo tipo** original, pero de menor complejidad. A su vez, cada subproblema se sigue dividiendo de la misma forma, hasta llegar a un subproblema fácilmente resoluble, que llamaremos **caso base.**

Técnica Recursiva

Técnica para resolver problemas reduciéndolos a situaciones similares a la original, pero de menor complejidad

Ejemplo:

Proponemos, a modo de ejemplo, una situación común en la vida real, propuesta pro Eric Roberts en su libro Arte y Ciencia del lenguaje C:

Una Entidad de Beneficencia necesita $1000000. El director sabe que hoy en día un donativo importante no supera los $100. Su problema se reduce a conseguir $1000000 directamente o bien conseguir 10000 personas que donen $100. Sigue siendo muy difícil conocer tantas personas, por lo tanto, delega el problema en 10 voluntarios regionales, cada uno de los cuales deberán recaudar $100000

Cada voluntario regional debe conseguir $100000 , cosa poco probable, o bien conseguir 1000 personas de su región, dispuestas a donar $100. Tal vez sea mejor que cada voluntario regional reparta el problema entre 10 voluntarios zonales, que deberán recaudar $10000

Esos voluntarios zonales podrán repartir la tarea entre 10 personas, fáciles de conseguir entre amigos y vecinos, de manera que cada persona recaude $1000, lo cual se logrará más fácilmente con la venta de 10 bonos de $100.

El trabajo de recolección se puede simplificar en el siguiente esquema algorítmico:

RecolectarDonación de $N

si ($N <= $100)
tomar el dinero de un donante
sino
encontrar 10 voluntarios
pedirle a cada voluntario **RecolectarDonanción de $N/10**
recolectar el dinero de los 10 voluntarios

2. *Recursividad Directa*

Recursividad Directa

Repetición por autorreferencia, de una función que se llama a sí misma.

De acuerdo a lo visto en el ejemplo de la sección anterior, podríamos proponer un esquema básico para toda función recursiva, donde el paso decisivo para saber si se vuelve a autoinvocar o no es que se cumpla la condición que asegura haber llegado al caso base, cuya solución es simple y conocida:

```
tipo
funcionRecursiva (listaParámetros)
{
        if ( se cumple caso base)
                retornar la solución simple
        else
        {
                dividir el problema en subproblemas de igual formato
                resolver cada llamada recursiva de funcionRecursiva (....)
                retornar la combinación de las soluciones parciales
        }
}
```

La recursividad directa consta de tres partes:

- **Prólogo**: se guardan en el stack las variables locales, los parámetros y la dirección de retorno (en C es automático).
- **Cuerpo**: se evalúa una condición, según su resultado se ejecutan pasos y luego se vuelve al prólogo (***paso recursivo***) o se pasa al epílogo (***caso base***).
- **Epílogo**: se restituyen desde el stack las variables locales, los parámetros y la dirección de retorno (en C es automático).

Consejo Muy Importante

NO usar como condición para la recursión ni ***while***, ni ***do-while***, ni ***for***

Un **error grave** es **omitir el caso base o no converger hacia él** en las reiteradas llamadas recursivas, ya que esto produce una **recursión infinita**, que por lo tanto no tiene calidad de algoritmo. (Recordar lo frustrante que son las referencias circulares en un diccionario).

Como uno puede intuir, la resolución recursiva computacional de un problema puede ocupar mayor tiempo de ejecución, debido a las reiteradas llamadas a la función, y también puede ocupar mucho recurso de memoria (cada llamada recursiva copia en el stack los parámetros, las variables locales y la dirección de retorno)

Cabe aclarar que cualquier problema que se pueda resolver en forma recursiva también puede encararse en forma iterativa, aunque en algunos casos pueda resultar muy complicado.

La mejor elección es usar el enfoque recursivo cuando es natural al problema y le confiere mayor claridad y facilidad de depuración que un tratamiento iterativo.

En el resto de la presentación del tema, proponemos muchas funciones recursivas que sería mejor tratarlas iterativamente. Esto lo hacemos a modo de práctica, pero enfatizamos los ejemplos que son de naturaleza típicamente recursiva.

2. Ejemplos de Funciones Recursivas

Para comprender mejor el tema vamos a presentar a continuación varios ejemplos de funciones recursivas, mostrado en algunos de ellos el seguimiento con el modelo de las *cajas con ventanas* (representando al stack)

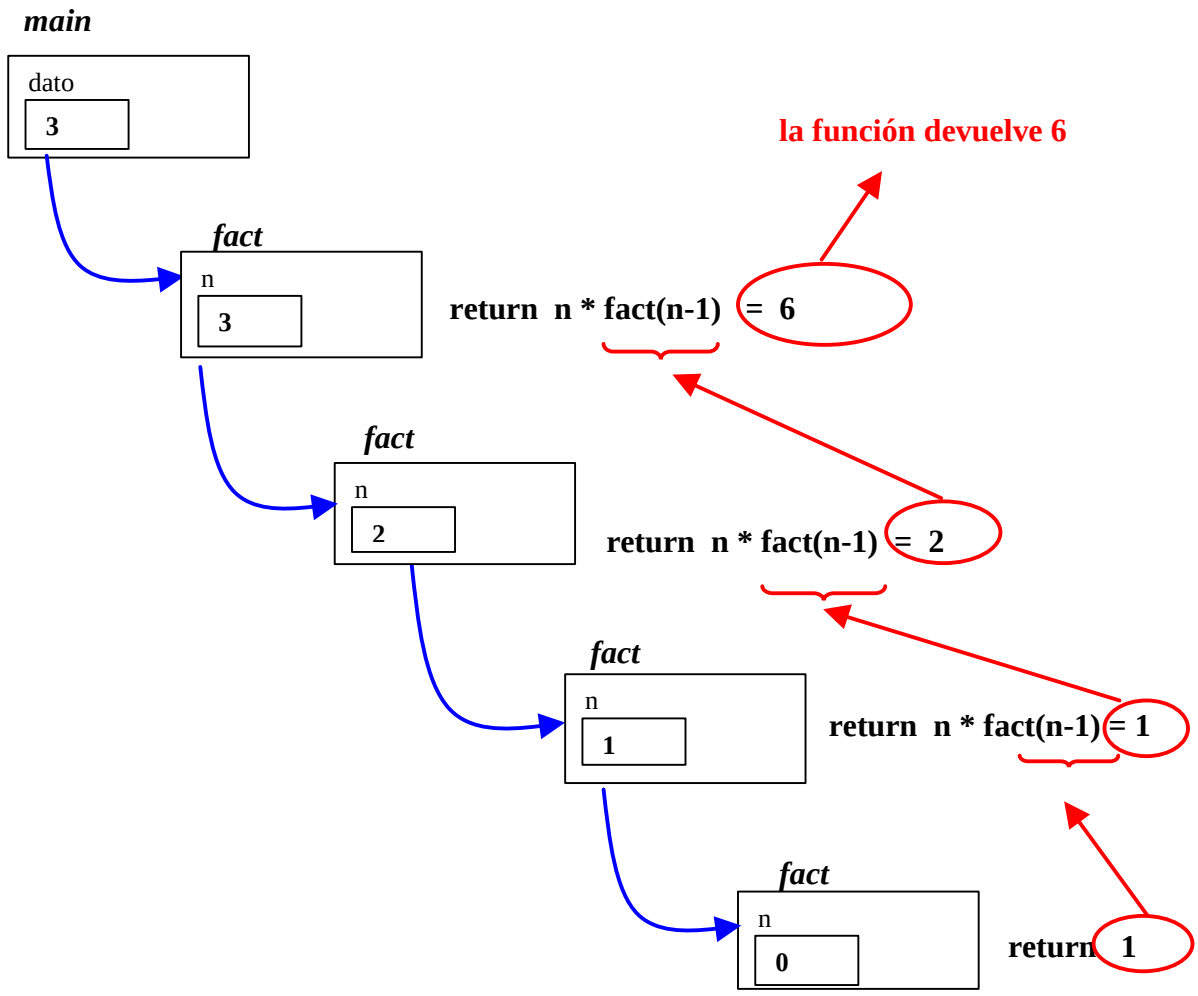
Ejemplo 1

El cálculo de un factorial se puede encarar recursivamente.
¿Cuál es el caso base? Es 0! = 1.

```
long
fact (long n)
{
      if ( n== 0)
            return (1);
      else
            return ( n * fact(n - 1) );
}
```

Posible invocación :

```
int  dato = 3;
printf ( "%d",  factorial (dato) );
```



Ejemplo 2

La potencia entera (exponente entero) de un número real también puede ser tratada aplicando recursividad, ya que matemáticamente se tiene que:

$$N^4 = N * N^3$$
$$N^3 = N * N^2$$
$$N^2 = N * N^1$$
$$N^1 = N * N^0$$
$$N^0 = 1$$

```
/*  los parametros estan validados:
**  exponente >=0   y  nunca 0 con base 0
*/

int
potencia( int base, int exponente)
{
     if (base == 0)
          /* para evitar recursión  cuando la base es nula */
          return  ( 0 );
     else
          if (exponente== 0)
               return  ( 1 );
          else
               return ( base * potencia(base, exponente - 1) );
}
```

Notar que si se olvida el caso base, la última invocación retornaría basura.

Posible Invocación:

```
.................
int base, exponente;

printf("\nIngrese una base y un exponente no negativo:");
scanf("%d %d", &base, &exponente);
printf("\n%d^  %d=%ld\n",     base,   exponente,   potencia(base,
exponente));
.................
```

Ejemplo 3

Un caso ***típico*** de recursividad lo conforma la serie de números de Fibonacci. Recordemos que la sucesión de Fibonacci (Liber Abbaci, 1202) se obtienen con la siguiente regla:

$$T_N = \begin{cases} N & \text{si } N = 0 \text{ ó } N = 1 \\ T_{N-1} + T_{N-2} & \text{si } N >= 2 \end{cases}$$

T_0	T_1	T_2	T_3	T_4	T_5	T_6	...
0	**1**	**1**	**2**	**3**	**5**	**8**	**...**

```
long
fibo( long n)
{
     if ( n < 2 )
          return  ( n );
     else
          return  fibo(n-1) + fibo(n-2);
}
```

Posible Invocación:

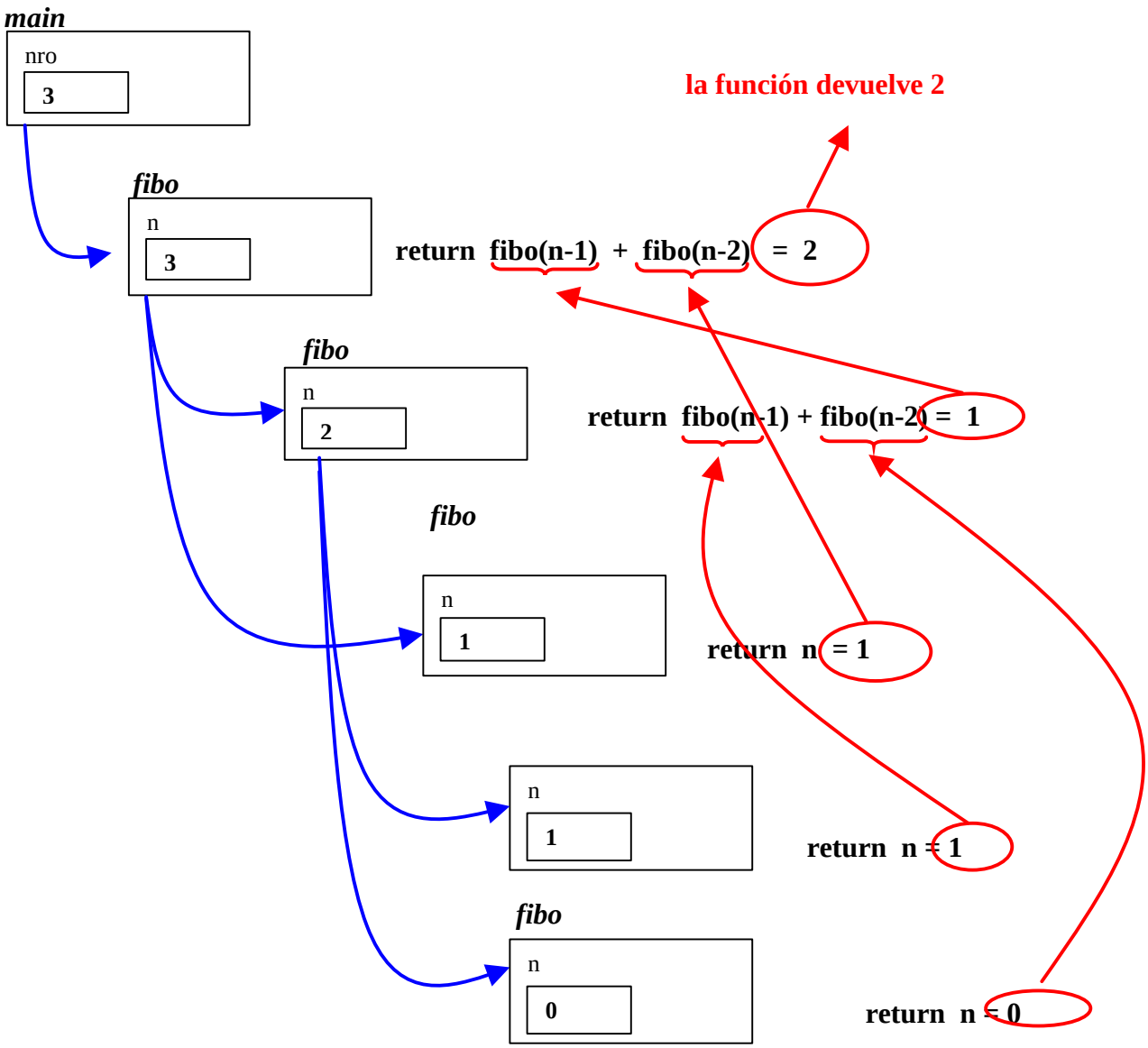
```
.................
printf("\nIngrese un nro:");
scanf("%d", &nro);
printf("\nfibo(%d)= %ld\n", nro, fibo(nro));
...................
```

Muy Importante

Cuando se hacen dos llamadas recursivas simultáneas dentro de una misma expresión hay que tener mucho cuidado porque ANSI no especifica el orden de la evaluación de los operandos.

El programador no debe hacer ninguna suposición relacionada con el orden en el cual se ejecutaran esas llamadas, para no obtener efectos colaterales que pudieran afectar el resultado final de la expresión.

En la función **fibo**, es evidente que el orden no influye en el resultado final.



Supongamos que se ingresó **nro=4**:

fibo(4) = fibo(3) + fibo(2)

= fibo(2) + fibo(1) + fibo(1) + fibo(0)

= fibo(1) + fibo(0) + 1 + 1 + 0

= 1 + 0 + 1 + 1 + 0

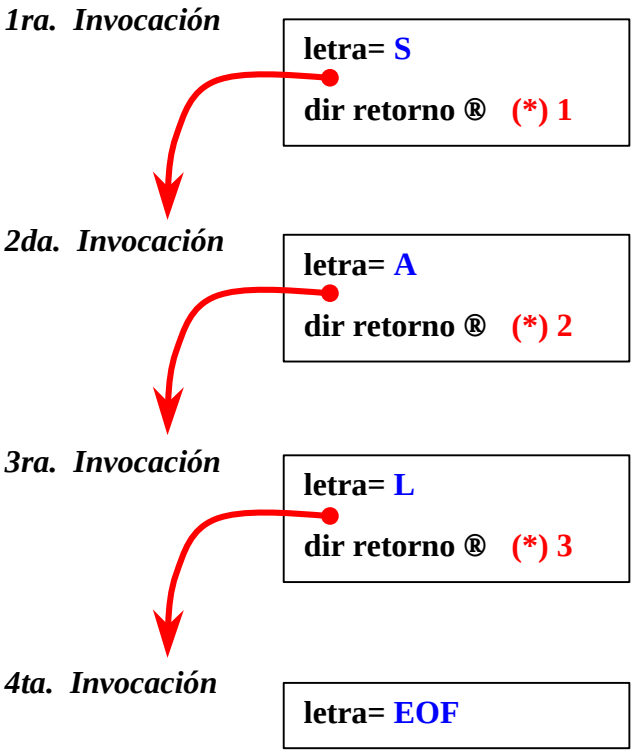
= 3

Ejemplo 4

Indicar qué se logra con cada función recursiva:

```
#include <stdio.h>



/*  Para esta funcion, suponer m>=0  y  m>=n  */
int
queEs(int m, int n)
{
      if (m==0  || n==0)
            return  ( 1 );
      else
            return  ( queEs(m-1, n-1) * m / n );
}



void
cualquiera(void)
{
      int letra;

      if ( (letra= getchar() ) != EOF )
      {
            cualquiera();
(*)         printf("%c", letra);
      }
}



int
main(void)
{
      int n1, n2;

      printf("\nIngrese un texto, y finalice con ^D\n");
      cualquiera();

      printf("\nIngrese dos numeros no negativos:");
      scanf("%d %d", &n1, &n2);
      printf("\n(%d %d)= %d\n", n1, n2, queEs(n1, n2) );

      return 0;
}
```

Para la invocación ***cualquiera()***, suponemos que se ingresa: **SAL *Ctrl-D* :**



Al invocar a la función ***cualquiera*(),** la variable local ***letra*** se lleva la letra **S** y luego, como hay una nueva invocación, se guarda la dirección de retorno, que en este caso es la dirección de la instrucción **printf("%c", letra);** que queda pendiente.

Este proceso se repite hasta encontrarse con **EOF.** En ese punto, la cuarta invocación devuelve en control a quien la invocó, con lo cual el punto de retorno está en la instrucción `printf("%c", letra);` de **(*) 3** , impirmiéndose en la salida estándar el valor de la variable **letra** del **stack frame** correspondiente a la tercera invocación, o sea, 'L'.

Después de esta instrucción termina la ejecución de la tercera llamada, que regresa el control a la instrucción `printf("%c", letra);` de **(*) 2** , impirmiéndose en la salida estándar el valor de la variable **letra** del **stack frame** correspondiente a la tercera invocación, el carácter 'A'.

Finalmente se vuelve a la primera invocación y se imprime el carácter 'S', obteniéndose:

LAS

Luego, la función ***cualquiera()*** no devuelve valor alguno, pero imprime los caracteres de la entrada estándar **al revés.**

IMPORTANTE

Todas las **acciones que se pospongan** hasta después de la llamada recursiva, se realizarán en el **orden inverso** en el cual fueron invocadas (recordar que los stack frames se apilan desde el primero hasta el último).

Para la invocación ***queEs(n1, n2)*****,** supondremos que en **n1** se ingresó **3** y que en **n2** se ingresó **2:**

```
queEs( 3, 2 )   =   queEs( 2, 1 )  *   3 / 2
                =   queEs( 1, 0 )  *   2 / 1   *  3 / 2
                =         1        *   2 / 1   *  3 / 2
```

Para un segundo seguimiento consideraremos que en **n1** se ingresó **5** y que en **n2** se ingresó **3:**

```
queEs( 5, 3 )   =   queEs( 4, 2 )  *   5 / 3
                =   queEs( 3, 1 )  *   4 / 2   *   5 / 3
                =   queEs( 2, 0 )  *   3 / 1   *   4 / 2   *  5 / 3
                =         1        *   3 / 1   *   4 / 2   *  5 / 3
```

Lo que hace la función ***queEs()*** es calcular el número combinatorio (m,n) en forma recursiva:

Combinatorio(m,n) = m! / (n! (m -n)!)

NOTAS IMPORTANTES

- En general las funciones recursivas NO validan sus parámetros, ya que esto suele complicar el código e introduce ineficiencia, porque se termina validando en cada invocación. De querer hacer algún tipo de validación, hay que escribir una función validadora, que luego invoque a la función recursiva.

- Si una función A tiene en su código la invocación a otra función B que se invoca a sí misma, entonces la función B es recursiva pero la función A NO LO ES.

- No usar variables tipo *static* para almacenar respuestas de funciones recursivas, ya que ésto sólo puede ser implementado en Lenguaje C y en la mayoría de los casos la migración de su código a otro lenguaje resulta ser bastante costosa.

En Programación I no aceptaremos funciones recursivas con variables *static*.

Recursividad - Parte II

Introducción

En este documento se presentan varios ejercicios de recursividad con sus respuestas, entre ellos uno de los más famosos casos de recursión: las Torres de Hanoi.

1. Torres de Hanoi

El juego de Torres de Hanoi es milenario, pero aquí presentamos su resolución computacional, obviamente recursiva.

Se tienen N discos de distinto tamaño y tres varillas en las cuales se insertan los mismos. Los discos sólo se pueden cambiar de varilla de a uno por vez y nunca puede colocarse un disco sobre otro si tiene mayor tamaño que el inferior.

El juego consiste en tener una torre de N discos en una de las varillas y trasladarlo a otra varilla predeterminada de antemano siguiendo las reglas y en la menor cantidad de pasos posibles.

Para hacer el planteo recursivo se piensa de la siguiente manera:

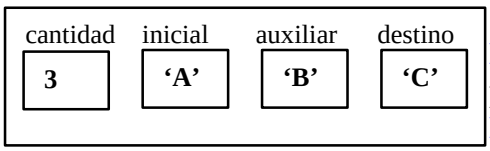
¿Cómo hacer para trasladar N discos desde A hacia C? Sería sencillo si pudiera pasar N-1 discos a B, usando C como auxiliar, luego el último disco de A a C, y finalmente los N-1 volverlos a pasarlos de B a C, usando A como auxiliar.

¿Cómo hacer para trasladar N-1 discos desde un origen hacia un destino? Paso N-2 discos usando la tercera varilla como auxiliar, paso el último disco y luego reordeno los N-2 disco hacia su destino.

Así sucesivamente, hasta llegar a tener que trasladar 0 discos.

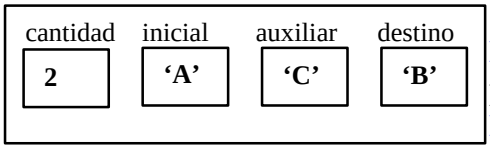
```
void
Hanoi( int cantidad, char inicial, char auxiliar,  char destino)
{
     if (cantidad > 0)
     {
          Hanoi(cantidad-1, inicial, destino, auxiliar);
          printf("Mover %c -> %c\n", inicial, destino);
          Hanoi(cantidad-1, auxiliar, inicial, destino);
     }
}
```

Para realizar el seguimiento, tomaremos la invocación: **Hanoi(3, 'A', 'B', 'C');**



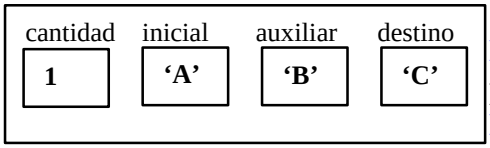
Hanoi(cantidad-1, inicial, destino, auxiliar); (1)
printf("Mover %c-> %c\n", inicial, destino); (11)
Hanoi(cantidad-1, auxiliar, inicial, destino); (12)

(1)



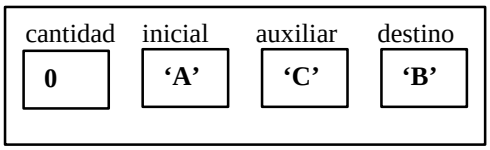
Hanoi(cantidad-1, inicial, destino, auxiliar); (2)
printf("Mover %c -> %c\n", inicial, destino); (6)
Hanoi(cantidad-1, auxiliar, inicial, destino); (7)

(2)

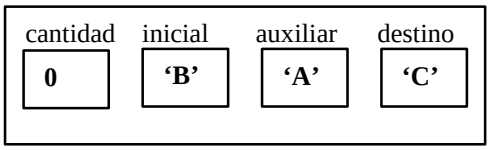


Hanoi(cantidad-1, inicial, destino, auxiliar); (3)
printf("Mover %c -> %c\n", inicial, destino); (4)
Hanoi(cantidad-1, auxiliar, inicial, destino); (5)

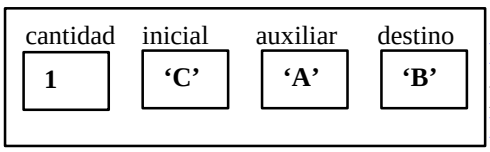
(3)



(5)

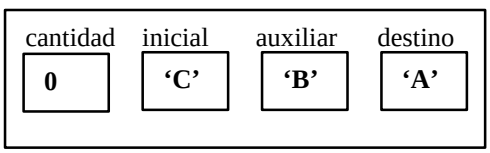


(7)

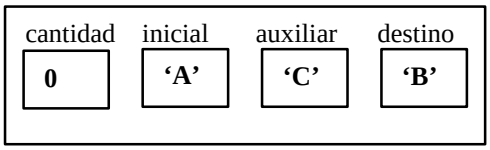


Hanoi(cantidad-1, inicial, destino, auxiliar); (8)
printf("Mover %c -> %c\n", inicial, destino); (9)
Hanoi(cantidad-1, auxiliar, inicial, destino); (10)

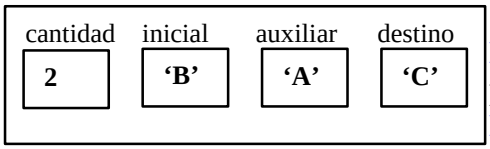
(8)



(10)

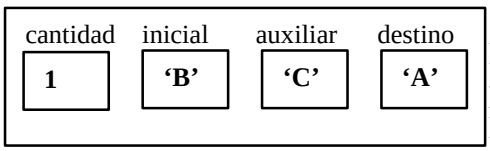


(12)



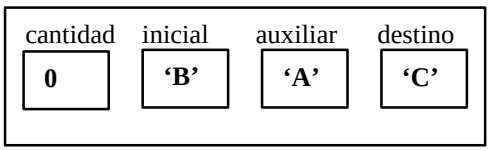
Hanoi(cantidad-1, inicial, destino, auxiliar); (13)
printf("Mover %c-> %c\n", inicial, destino); (17)
Hanoi(cantidad-1, auxiliar, inicial, destino); (18)

(13)

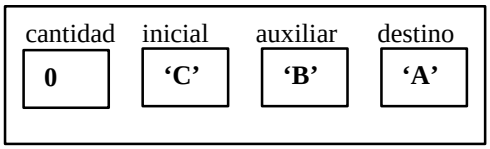


Hanoi(cantidad-1, inicial, destino, auxiliar); (14)
printf("Mover %c-> %c\n", inicial, destino); (15)
Hanoi(cantidad-1, auxiliar, inicial, destino); (16)

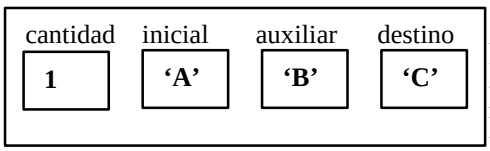
(14)



(16)

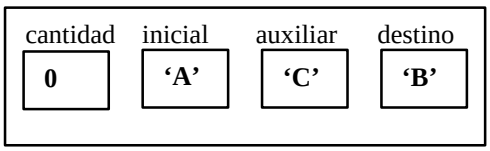


(18)



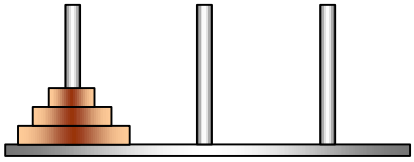
Hanoi(cantidad-1, inicial, destino, auxiliar); (19)
printf("Mover %c-> %c\n", inicial, destino); (20)
Hanoi(cantidad-1, auxiliar, inicial, destino); (21)

(19)

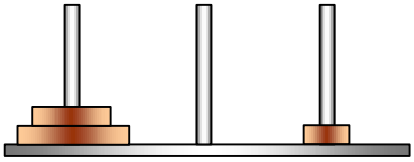


(21)

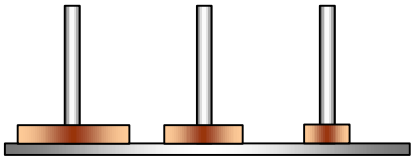
cantidad	inicial	auxiliar	destino
0	**'B'**	**'A'**	**'C'**



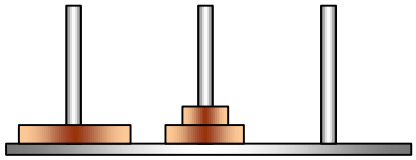
Estado Inicial



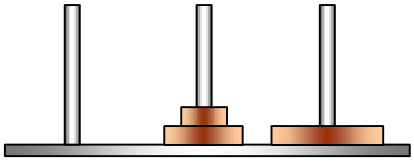
(4) ***A ® C***



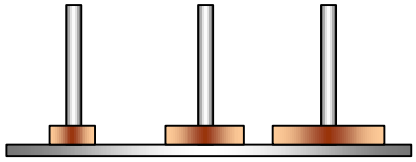
(6) ***A ® B***



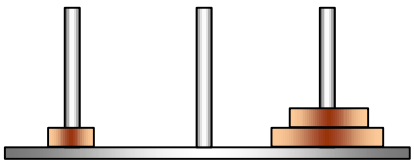
(9) ***C ® B***



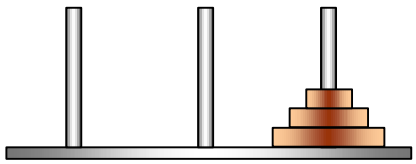
(11) ***A ® C***



(15) ***B ® A***



(17) ***B ® C***



(20) ***A ® C***

Qué Genio!!!

2. Más Ejercicios

Ejercicio 1

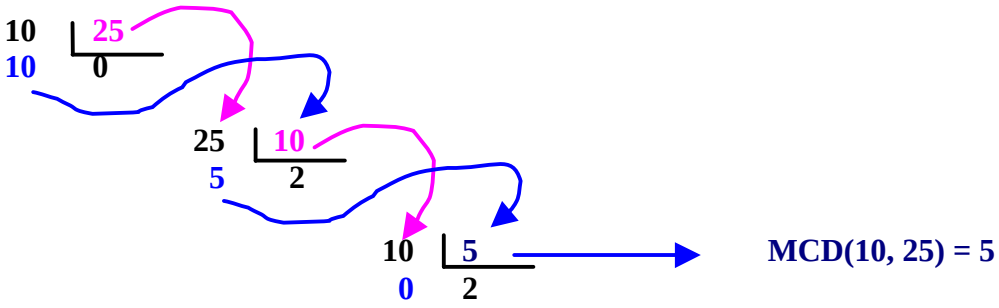
Escribir una función recursiva que reciba como parámetros dos números enteros y que devuelva el máximo común divisor (usar algoritmo de Euclides)

Algoritmo de Euclides

Se divide un número por el otro. Si no se obtiene resto nulo, se continúa dividiendo cada divisor por el resto obtenido, hasta que la división sea exacta. El último divisor es el MCD.

Ejemplo:

Buscamos el MCD entre 10 y 25



La idea es que si entramos una vez más, 5 dividido 0 no se puede realizar, pero en ese caso estamos en presencia del **caso base,** y el dividendo 5 sería el MCD buscado.

Rta:

```
int
mcd( int a, int b)
{
      if  ( b==0 )
            return  a;
      else
            return   mcd(b, a % b);
}
```

no omitir el return

Ejercicio 2

Indicar qué hace la siguiente función recursiva:

```
void
whatIs(int  num)
{
      if (num>=2)
      {
            whatIs( num / 2);
            putchar ( num % 2 + '0' );
      }
      else
           putchar( num + '0');
}
```

Rta:

Es una función recursiva que recibe como parámetro un número entero en base decimal e imprima en la salida estándar su equivalente en sistema binario.

Ejercicio 3

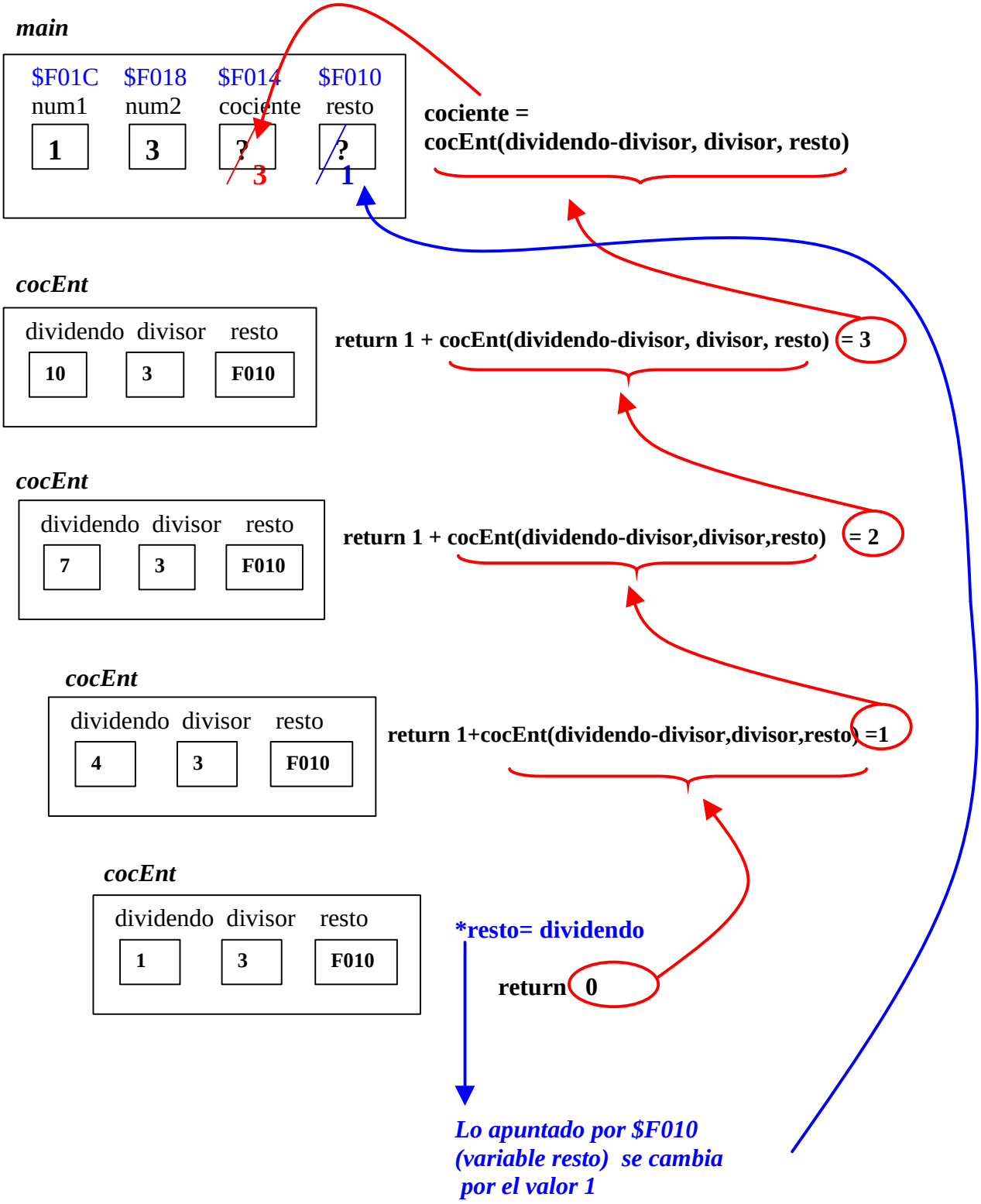
Escribir una función recursiva que calcule el cociente entero entre dos enteros positivos, devolviendo el resto en un tercer parámetro.

Rta:

```
int
cocienteEntero( int dividendo,  int divisor,  int* resto)
{
      if (dividendo < divisor)
      {
            *resto= dividendo;
            return 0;
      }
      else
          return 1+cocienteEntero(dividendo-divisor, divisor, resto);
}
```

Para mostrar un seguimiento, supondremos la siguiente invocación:

```
int  num1= 10, num2= 3, cociente, resto;
cociente  = cocienteEntero( num1, num2, &resto);
```



Ejercicios de Recursividad

Introducción

En este documento se presentan más ejercicios de recursividad con sus respectivas respuestas, para practicar algunos conceptos importantes de la técnica recursiva.

Ejercicio 1

Escribir una función recursiva que reciba un arreglo de enteros con su dimensión y devuelva 1 si el mismo es nulo y 0 en caso contrario.

Posible invocación:

```
..........
int  array[10] = { 0 , 0,  4 , 0, 7 };
printf("array  %s nulo \n",
      esNulo(array, sizeof(array)/sizeof(array[0]) )?"es":"no es");
..........
```

Respuesta

Una posibilidad es correr en cada nueva invocación el puntero recibido como parámetro a la siguiente componente, bajando la dimensión del arreglo apuntado. De esta forma la condición base es tener dimensión nula:

```
int
esNulo(int vector[], int dim)
{
      if (dim == 0)
            return (1);
      else
            return ((vector[0]==0)  &&  esNulo(vector+1, dim-1) );
}
```

¿Qué diferencia hubiera tenido cambiar el orden en el ***return***?

```
      return ( esNulo( vector+1, dim - 1)  &&  (vector[0]==0) );
```

En la primera versión, al encontrar un elemento del arreglo no nulo, como && es “lazy”, la invocación recursiva ya no se realiza y la función devuelve 0 (falso) en su lugar. Luego la respuesta final será 0, ya que se hace un AND entre todos los resultados parciales. En esta última versión se sigue entrando hasta recorrer todo el arreglo y recién se detecta el caso falso, si lo hubiera, al regresar.

Ejercicio 2

Escribir una función recursiva que reciba un string (cadena null terminated) y devuelva la cantidad de vocales que contiene.

Posible invocación:

```
.....................
printf("Ingrese una palabra\n");
scanf ("%s", pal);
printf("vocales= %d \n", cantVocales(pal));
.........................
```

Respuesta

La idea es recorrer el string, corriendo el puntero a char en una componente para cada nueva invocación recursiva, hasta alcanzar el caso base, que es detectar la finalización del mismo (encontrando el caracter '\0'). Dentro de cada ejecución, si la cabeza del string es una consonante se vuelve a invocar la función, si es una vocal se vuelve a invocar pero incrementando en 1 el resultado:

```
int
cantVocales(char palabra[])
{
      if (palabra[0] == '\0')
            return ( 0 );
      else
            switch( toupper(palabra[0]))
            {
                  case 'A':
                  case 'E':
                  case 'I':
                  case 'O':
                  case 'U':
                        return  ( 1 + cantVocales (palabra +1) );
                  default:
                        return  ( cantVocales (palabra +1) );
            }
}
```

Notar que no hace falta colocar ***break***, ya que la proposición ***return*** es un corte.

Ejercicio 3

Escribir una función que detecte si una palabra es palíndroma, recibiendo la palabra y su cantidad de letras:

```
" S A L A S "        " N A R R A N "
  " A L A "            " A R R A "
    " L "                " R R "
     ""                   ""
```

¿Cuál es el caso base? Caso más genérico: string nulo es palíndromo

Respuesta:

```
int
esPalindromo( char palabra[ ], int longitud)
{
      if (longitud <= 1)
            return 1;
      else
            return    ( palabra[0] == palabra[longitud-1]
                              &&  esPalindromo(palabra+1, longitud-2);
}
```

Alterar la cadena de entrada, sería un error grave !!!

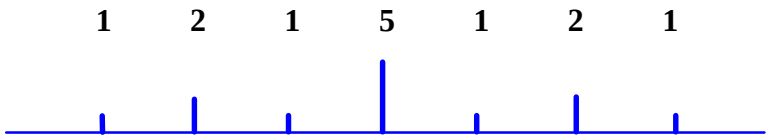
Por ejemplo, en el código anterior, sería pésimo hacer:

```
      strncpy( palabra, palabra +1, longitud-2);
```

Ejercicio 4

Escribir una función recursiva que reciba un entero representando la altura central de una regla y muestre por pantalla los tamaños de las subdivisiones a cada lado, sabiendo que cada subdivisión es la mitad de la anterior.

Ejemplo: La invocación **regleta(5);** debería imprimir



Respuesta

La idea es repetir el mismo patrón antes y después de la impresión del valor deseado, sin imprimirlo hasta no haber llegado al valor 1 de cada lado, recursivamente.

```
void
regleta( int altura)
{
      if (altura > 0  )
      {
            regleta( altura / 2 );
            printf("%d ", altura);
            regleta( altura / 2);
      }
}
```

Notar que si se hubiera usado:

```
            return  regleta( altura / 2 );
            printf("%d ", altura);
            return regleta( altura / 2);
```

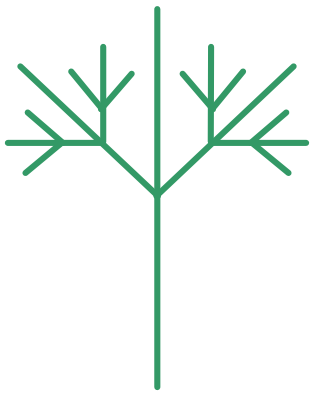
no funciona, ya que el ***return*** corta la ejecución de las instrucciones que le siguen (nunca hubiera alcanzado el ***printf*** o la segunda llamada recursiva).

Ejercicio 5

Escribir una función recursiva que reciba un entero representando la altura de la rama inicial de un árbol fractal y la altura mínima tolerada para graficar, y devuelva la cantidad de ramas que se pueden dibujar del mismo, sabiendo que en cada paso se generan dos nuevas ramas desde la mitad de las ramas del paso anterior, a 45º de inclinación y de mitad tamaño.

Ejemplo:

La invocación **ramasArbol(7, 0.5);** debería devolver **15**



Respuesta

Debemos pensar que en cada rama siempre ocurre la misma secuencia: se dibujan tres líneas, la inicial y dos de mitad tamaño. Este esquema se repite hasta que el tamaño de la rama que se quiere representar sea menor que el mínimo aceptado, en cuyo caso no se dibuja (caso base).

```
int
ramasArbol(float longitud, float minimo)
{
      if (longitud < minimo )
            return 0;
      else
            return 1 + 2 * ramasArbol(longitud / 2, minimo);
}
```

tamaño (7) Þ **1 + 2 * tamaño (3.5) ramas = 15 ramas**

tamaño (3.5) Þ **1 + 2 * tamaño (1.75) ramas = 7 ramas**

tamaño (1.75) Þ **1 + 2 * tamaño (0.875) ramas = 3 ramas**

tamaño (0.875) Þ **1 + 2 * tamaño (0.4375) ramas = 1 rama**

tamaño (0.4375) Þ **0 ramas**

Ejercicio 6

Escribir la función recursiva del ejercicio anterior, pero que en vez de devolver la cantidad de ramas en su nombre, la devuelva en un tercer parámetro.

Posible invocación:

```
.........................
int cant;
ramasArbol2( 7 , 0.5 , &cant );
printf( "cantidad de ramas =%d\n", cant );
.........................
```

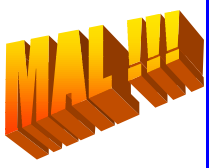
Respuesta

Como la respuesta se debe hacer en un parámetro, éste debe ser un puntero a la zona de memoria donde se desea almacenar el resultado. Dicha dirección apunta a un lugar en el cual no necesariamente está el contador inicializado. Por lo tanto, la inicialización del mismo se debe hacer en el caso base:

```
void
ramasArbol2(float longitud, float minimo, int* cantRamas)
{
      if (longitud < minimo )
            *cantRamas= 0;
      else
      {
            ramasArbol2( longitud / 2, minimo, cantRamas);
            *cantRamas= 1 + 2 * *cantRamas;
      }
}
```

Notar que no hubiera servido un simple cambio de asignación, en lugar de usar el return de la versión del ejercicio 4, ya que la función es ***void:***

```
void
ramasArbol2(float longitud, float minimo, int* cantRamas)
{
      if (longitud < minimo )
        *cantRamas= 0;
      else
        *cantRamas= 1+2*ramasArbol2( longitud / 2, minimo, cantRamas);
}
```



Implementación de Biblioteca Estándar

Introducción

En este documento se presentan las implementaciones de algunas funciones de la Biblioteca Estándar de C.

1. Funciones para Manejo de Strings (string.h)

- ***Función strcpy***

Mostramos a continuación una implementación con uso de subíndices:

```
char*
strcpy(char* dest, const char* fuente)
{
      int i;

      for ( i=0 ; fuente[i] != '\0'; i++ )
            dest[i] = fuente[i];
      dest[i] = '\0';
      return dest;
}
```

La versión anterior se puede resumir, colocando la asignación dentro de la condición, estilo muy típico en C:

```
char*
strcpy(char* dest, const char* fuente)
{
      int i;

      for ( i=0; (dest[i] = fuente[i]) != '\0' ; i++ )
            ;
      return dest;
}
```

¿Por qué entre paréntesis?

La siguiente es una implementación con uso de punteros:

```
char*
strcpy(char* dest, const char* fuente)
{
      char* s;

      for ( s=dest ; (*s++ = *fuente++) != '\0' ; )
            ;
      return dest;
}
```

Se puede obviar, pero resulta más claro

- ***Función strncpy***

```
char*
strncpy(char* dest, const char* fuente, int n)
{
      char* s;

      for(s= dest;  n > 0  &&  *fuente != '\0' ; --n)
            *s++= *fuente++;

      /* si la longitud de fuente es manor a n, se completa con 0 */
      for( ; n > 0 ; --n)
            *s++= '\0';

      return dest;
}
```

- ***Función strcat***

```
char*
strcat(char* dest, const char* fuente)
{
      char* s;

      for(s= dest; *s!='\0'; ++s)
            ;

      for( ; (*s= *fuente) != '\0'; ++s, ++fuente)
            ;

      return dest;
}
```

- ***Función strncat***

```
char*
strncat(char* dest, const char* fuente, int n)
{
      char* s;

      for( s= dest ; *s!='\0'; ++s )
            ;

      for( ; n > 0  &&  *fuente != '\0'; --n )
            *s++= *fuente++;
      *s= '\0';

      return dest;
}
```

- ***Función strcmp***

```
int
strcmp(const char* dest, const char* fuente)
{
      for( ; *dest == *fuente ; ++dest, ++fuente )
            if (*dest == '\0')
                  return 0;

      return *(unsigned char*)dest < *(unsigned char*)fuente? -1: 1;
}
```

- ***Función strchr***

```
char*
strchr(const char* s, int c)
{
      char ch= c;

      for( ; *s!=ch; ++s)
            if (*s == '\0')
                  return NULL;

      return (char*) s;
}
```

2. *Funciones Generales (stdlib.h)*

- ***Función rand***

Hemos elegido esta función para mostrar una posible versión para generar números pseudo-aleatorios.

Suponemos que se tiene la semilla seteada como:

```
unsigned long _Randseed= 1;
```

Una posible implementación de la función puede ser la siguiente, que garantiza una buena distribución:

```
int
rand(void)
{
      _Randseed = _Randseed * 1103515245 + 12345;

      return (unsigned int) _Randseed >> 16 & RAND_MAX;
}
```

3. Funciones Para Manejo de Caracteres (ctype.h)

Cuando se trabaja con texto, ya sea tomando información desde la entrada estándar, tomando datos desde un archivo, etc., se suele dedicar al procesamiento de caracteres más del 50 % de la tarea, como por ejemplo, eliminar blancos de más, pasar a mayúsculas para hacer búsqueda con éxito, validando información numérica, etc.

Este proceso hace que las funciones para manejo de caracteres sean muy utilizadas. Sin embargo la implementación de dichas funciones puede resultar inservible si no se amolda al tipo de alfabeto utilizado en la zona.

Por ejemplo, si la función **isalpha** estuviera implementada simplemente como:

```
int
isalpha( int c)
{
      return  (  c >='A'  &&  c <= 'Z'  ||  c >='a'  &&  c <= 'z'  )
}
```

para pasar el caracter **ch** a mayúscula en alfabeto inglés bastaría con invocar **toupper(ch)**.

Sin embargo, no tendría un comportamiento del todo correcto para alfabeto español, francés, ruso, chino, etc. Con UNICODE se representan caracteres de todo tipo, y hay caracteres que representan letras (á, é, í, ñ, Ä, etc.) mientras que el pasaje a mayúscula de la implementación dada para **toupper** no los contempla.

Una solución es re-escribir cada función de manejo de caracteres para cada idioma en particular, y la otra es tener una única implementación que se base en la tabla de los códigos correspondientes a cada idioma. Obviamente, se eligió la segunda.

De esta forma, se agrupan los caracteres en "categorías" (letras mayúsculas, letras minúsculas, dígitos, espacios en blanco, etc.) y se le asigna un código hexadecimal a cada categoría, de manera tal que los códigos no estén solapados. Para esto, lo mejor es tomar potencias de 2.

Luego se define una tabla (por alfabeto) con tantos lugares como posibles códigos ASCII existan y se coloca en cada lugar de la tabla el código de la categoría a la que pertenece el carácter cuyo ASCII coincide con esa posición.

Evidentemente la implementación de cada función de testeo consistirá en acceder directamente al lugar del caracter a investigar y hacer un AND de bits entre el código allí presente y el código de la categoría deseada.

Cuando un lugar de la tabla corresponde a dos o más categorías, se coloca en su lugar el OR de bits entre los códigos de las mismas.

Por ejemplo, si el código de la categoría "dígito" es **0x20** y el de "dígito hexadecimal" es **0x01**, en los lugares de la tabla correspondientes a **'0', '1', ... '9'** se deberá colocar **0x21**, ya que se trata tanto de dígitos decimales como hexadecimales.

Por comodidad se definen constantes simbólicas para cada categoría:

Para los dígitos decimales:	**#define**	**_DI**	**0x20**
Para los dígitos hexadecimales:	**#define**	**_XD**	**0x01**
Para ambos:	**#define**	**_XDI**	**_DI | _XD**

A continuación, se muestra el archivo de encabezado con la declaración de macros para cada categoría y la tablas utilizadas tanto para el testeo como para la conversión de caracteres.

```
/*  _Ctype.h   */

#ifndef _CTYPE
#define _CTYPE

/* _Ctype code bits */
#define _XA 0x200    /* extra alphabetic */
#define _XS 0xl00    /* extra space */
#define _BB 0x80     /* BEL, BS, etc. */
#define _CN 0x40     /* CR, FF, HT, NL, VT */
#define _DI 0x20     /* '0'-'9' */
#define _LO 0xl0     /* 'a'-'z' */
#define _PU 0x08     /* punctuation */
#define _SP 0x04     /* space */
#define _UP 0x02     /* 'A'-'Z' */
#define _XD 0x01     /* '0'-'9', 'A'-'F', 'a'-'f' */


int isalpha(int);
int isdigit(int);
int isxdigit(int);
int isalnum(int);
int ispunct(int);
int isspace(int);
int iscntrl(int)
int isprint(int);
int islower(int);
int isupper(int);
int tolower(int);
int toupper(int);

extern const  short * _Ctype, *_Tolower, *_Toupper;

#endif
```

```
/* _Ctype  conversion table -- ASCII version */

#include <ctype.h>
#include <limits.h>
#include <stdio.h>

#if EOF != -1  ||  UCHAR_MAX != 255
#error WRONG_Ctype TABLE
#endif


/* macros */
#define XDI ( _DI | _XD )
#define XLO ( _LO | _XD )
#define XUP ( _UP | _XD )


/* static data */
static const short ctyp_tab[257] = {0,

_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,

_BB, _CN, _CN, _CN, _CN, _CN, _BB, _BB,

_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,

_BB, _BB, _BB, _BB, _BB, _BB, _BB, _BB,

_SP, _PU, _PU, _PU, _PU, _PU, _PU, _PU,

_PU, _PU, _PU, _PU, _PU, _PU, _PU, _PU,

XDI, XDI, XDI, XDI, XDI, XDI, XDI, XDI,

XDI, XDI, _PU, _PU, _PU, _PU, _PU, _PU,

_PU, XUP, XUP, XUP, XUP, XUP, XUP, _UP,

_UP, _UP, _UP, _UP, _UP, _UP, _UP, _UP,

_UP, _UP, _UP, _UP, _UP, _UP, _UP, _UP,

_UP, _UP, _UP, _PU, _PU, _PU, _PU, _PU,

_PU, XLO, XLO, XLO, XLO, XLO, XLO, _LO,

_LO, _LO, _LO, _LO, _LO, _LO, _LO, _LO,

_LO, _LO, _LO, _LO, _LO, _LO, _LO, _LO,

_LO, _LO, _LO, _PU, _PU, _PU, _PU, _BB,

};

const short *_Ctype = &ctyp_tab[1];
```

Dígitos del '0' al '9'

Letras de la 'A' a la 'F'

Letras de la 'G' a la 'Z'

```
/* Tolower conversion table -- ASCII version */

#include <ctype.h>
#include <limits.h>
#include <stdio.h>

#if EOF != -1 || UCHAR_MAX != 255
#error WRONG TOLOWER TABLE
#endif


/* static data */
static const short tolow_tab[257] = {EOF,
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0xl0, 0xll, 0x12, 0xl3, 0xl4, 0x15, 0xl6, 0xl7,
0x18, 0xl9, 0xla, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
0x40, 'a',  'b',  'c',  'd',  'e',  'f',  'g',
'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
'x',  'y',  'z',  0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
0x98, 'ö',  0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
0xaO, 0xal, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xse, 0xaf,
0xbO, 0xbl, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
0xcO, 0xcl, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
0xc8, 0xc9, 0xca, 0xcb, 0xac, 0xed, 0xce, 0xcf,
0xdO, 0xdl, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
0xeO, 0xel, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
0xe8, 0xe9, 0xea, 0xab, 0x c, 0xed, 0xee, 0xef,
0xfO, 0xfl, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
0xf8, 0xf9, 0xfa, 0xfb, 0xEc, 0xEd, 0xfe, 0xff
};

const short *_Tolower = &tolow_tab[1];
```

Al caer en el lugar del ASCII de la 'G' se lleva la 'g'

Si la tabla fuera para alfabeto alemán, al caer en el lugar del ASCII de la 'Ö' se lleva la 'ö'

- La distancia entre 'A' y 'a' es 20
- La distancia entre 'Ö' y 'ö' es 5
- La distancia entre 'Ñ' y 'ñ' es 1

Complicada la codificación por método aritmético

Muy simple la codificación por acceso directo a una tabla de

```
/* _Toupper conversion table -- ASCII version */

#include <ctype h>
#include <limita h>
#include <stdio h>

#if EOF != -1 || UCHAR_MAX != 255
#error WRONG TOUPPER TABLE
#endif


/* static const short toup_tab[257]= {EOF,
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0xl0, 0xll, 0x12, 0xl3, 0xl4, 0x15, 0xl6, 0xl7,
0x18, 0xl9, 0xla, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
0x40, 'A',  'B',  'C',  'D',  'E',  'F',  'G',
'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
'X',  'Y',  'Z',   0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
0xaO, 0xal, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xse, 0xaf,
0xbO, 0xbl, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
0xcO, 0xcl, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
0xc8, 0xc9, 0xca, 0xcb, 0xac, 0xed, 0xce, 0xcf,
0xdO, 0xdl, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
0xeO, 0xel, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
0xe8, 0xe9, 0xea, 0xab, 0x c, 0xed, 0xee, 0xef,
0xfO, 0xfl, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
0xf8, 0xf9, 0xfa, 0xfb, 0xEc, 0xEd, 0xfe, 0xff};


const short *_Toupper = &toup_tab[1];
```

Veremos a continuación la implementación a alguna de las funciones de manejo de caracteres, usando los accesos directos a tabla y la manipulación de bits, que resultan más rápidas que la condiciones booleanas.

- ***Función isdigit***

```
int
isdigit (int c)
{
      return (_Ctype[c] & _DI);
}
```

- ***Función isxdigit***

```
int
isxdigit(int c)
{
      return (_Ctype[c] & _XD );
}
```

- ***Función isalpha***

```
int
isalpha(int c)
{
      return (_Ctype[c] & ( _LO | _UP | _XA ) );
}
```

- ***Función isalnum***

```
int
isalnum(int c)
{
      return (_Ctype[c] & ( _DI | _LO | _UP | _XA ) );
}
```

- ***Función isspace***

```
int
isspace(int c)
{
            return (_Ctype[c] & ( _CN | _SP | _XS ) );
}
```

- ***Función islower***

```
int
islower (int c)
{
      return (_Ctype[c] & _LO);
}
```

- ***Función tolower***

```
int
tolower(int c)
{
      return (_Tolower[c]);
}
```

Ejercicio

Para que un programa escrito en C tenga en cuenta el lenguaje correcto (español, inglés, etc.) hay que setear una variable de entorno y luego debe tenerse en cuenta dicho seteo. Por ejemplo, si se busca imprimir la mayúscula de la letra 'n' y el alfabeto seteado es el inglés, se obtiene la misma letra. En cambio si estuviera seteado el alfabeto español se obtendría la 'Ñ'. Escribir el siguiente programa:

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

int
main(void)
{
      setlocale(LC_CTYPE, "");
      printf("%c\t%c\n", 'ñ', toupper('ñ'));
      return 0;
}
```

Compilarlo y ejecutarlo dos veces, previo seteo de la variable de entorno, de la siguiente manera:

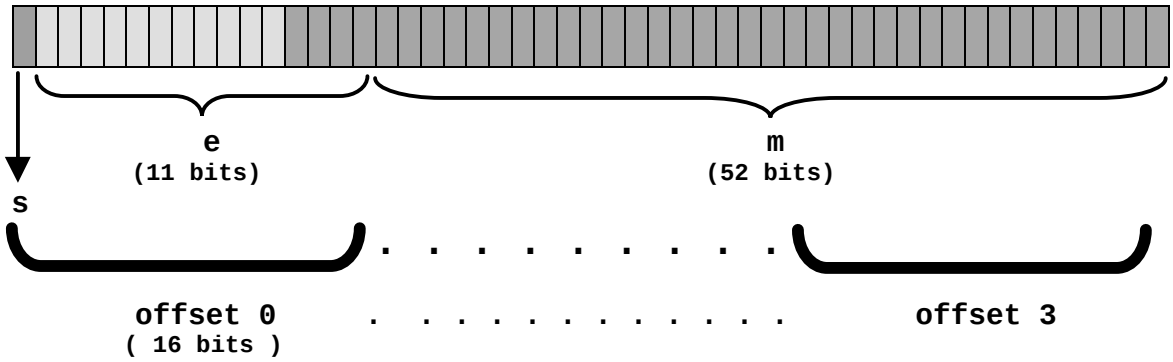
```
$ LC_ALL=en_UK; export LC_ALL
$ a.out

$ LC_ALL=es_ES; export LC_ALL
$ a.out
```

La primera opción setea el inglés de Reino Unido, por lo cual no interpreta la ´ñ´como una letra. La segunda, en cambio, setea el español de España y pasa la letra ´ñ´ a mayúscula correctamente (´Ñ´).

4. Funciones Matemáticas (math.h)

Tal vez, una de las mayores dificultades en la implementación de las funciones matemáticas que trabajan con tipos double sea validar el parámetro recibido y detectar desbordamientos de rango en los resultados, para poder setear la variable global **errno**.

La versión IEEE 754 para double de 8 Bytes estipula:



Recordemos que la formación del double en IEEE 754 está dado por:

- $0 < e < 255$ Þ $N = (-1)^s \ 1.f \ 2^{e-127}$
- $e = 255$ y $m \neq 0$ Þ NaN
- $e = 255$ y $m = 0$ Þ $N = (-1)^s \ ¥$
- $e = 0$ y $m \neq 0$ Þ $N = (-1)^s \ 1.f \ 2^{e-127}$
- $e = 0$ y $m = 0$ Þ $N = (-1)^s \ 0$

El objetivo de la función **_Dtest** es recibir un puntero a una zona de memoria donde se almacena un supuesto double e ir recorriéndolo de a 16 bits para detectar, de acuerdo a la norma IEEE 754, si se trata de un número válido (incluyendo el cero), de INFINITO o NAN. Debe quedar claro que **_Dtest no forma el número double**, sino que simplemente lo valida, devolviendo una constante de acuerdo a la siguiente convención:

```
#define NAN      2
#define INF      1
#define FINITE   -1
```

Como en los primeros 16 bits tenemos 1 bit de signo, 11 de exponente y 4 de mantisa, debemos crear máscaras para filtrar exponente y mantisa. Por otra parte debemos conocer el valor máximo del exponente **e** almacenado, a los efectos de detectar casos especiales:

```
#define _DFRAC    0xF         /* máscara para filtrar mantisa */
#define _DMASK    0x7FF0      /* máscara para filtrar exponente */
#define _DMAX     0x07FF      /* exponente máximo posible */
```

A continuación, mostramos una versión de la función _Dtest:

```
/* Funcion para validar un double almacenado con norma IEEE 754
** en una arquitectura donde el double ocupa 8 bytes
*/

short _DTEST(double * px)
{
      unsigned short* ps= (unsigned short*) px;
      short xchar= ( ps[0] & 0x7FF0) >> 4;            /* levanto e */


      /* con e==maximo y m==0 se tiene infinito,
         con e==maximo y m!=0 se trata de un real no válido */
      if (xchar == _DMAX)
            return ( ps[0] & 0xF || ps[1] || ps[2] || ps[3])?
                                                      NAN: INF;

      /* con 0 < e < maximo  o con e==0 y m!= 0,
         se tiene un real válido*/
      if (xchar > 0 || ps[O] & 0xF || ps[1] || ps[2] || ps[3])
            return FINITE;

      /* si se llega hasta aquí, se tiene e=0 y m=0 */
      return 0;
}
```

Una versión genérica de la función **_Dtest**, debería poder adaptarse a cualquier formato de IEEE:

- cualquier tamaño de bytes para el double
- cualquier cantidad de bits para el e almacenado (cambiando el valor del e máximo)
- cualquier tipo de arquitectura (que intercambie o no bytes en memoria)

Para esto, definimos todas las constantes en función del tamaño del double, medido en *short*, unidad que se usa para recorrerlo:

```
#define _DOFF     4          /* cantidad de shorts a recorrer */

#define _DFRAC    ( (1 << _DOFF) –1 )
#define _DMASK    ( 0x7FFFF & ~ _DFRAC )

#define _DMAX     ( (1 << (15 - _DOFF)) – 1 )
```

Por otra parte, previendo que se pueda estar usando una arquitectura en la cual cada acceso a memoria invierta o no los bytes de la palabra (recordar el word de Z-80), habrá que considerar que el short que contiene signo y exponente puede ser el primero o el último. Para que el código sea portable, conviene usar constantes:

```
#if _D0 == 3
#define     _D1= 2      /* little-endian */
#define     _D2= 1
#define     _D3= 0
#else
#define     _D1= 1      /* big-endian */
#define     _D2= 2
#define     _D3= 3
#endif
```

```
/* Versión genérica de la funcion para validar un double almacenado con
** norma IEEE 754
*/
short _DTEST(double * px)
{
      unsigned short* ps= (unsigned short*) px;
      short xchar= ( ps[_D0] & _DMASK) >> _DOFF;

      if (xchar == _DMAX)
            return  ( ps[_DO] & DFRAC || ps[_D1] || ps[_D2] || ps[_D3])?
                    NAN: INF;

      if (xchar > 0 || ps[_DO] & DFRAC || ps[_D1] || ps[_D2] || ps[_D3])
            return FINITE;

      return 0;
}
```

Finalmente, mostraremos el código de una de las funciones matemáticas de la biblioteca estándar.

- ***Función fabs***

Retorna el valor absoluto del double recibido como parámetro. En caso de un error de dominio, la variable **errno** toma el valor **EDOM**. Si el resultado produce un overflow, la función devuelve un valor máximo y **errno** toma el valor **ERANGE**.

```
double
fabs(double x)
{
      switch( _DTEST( &x)
      {
            case NAN:
                        errno= EDOM;
                        return x;
            case INF:
                        errno= ERANGE;
                        return MAX_INF;
            case 0:
                        return 0;
            default:
                        return  ( x<0.0? -x: x );
      }
}
```