

Programación Procedural



Apuntes de Cátedra

Licenciatura en Ciencias de la Computación
Licenciatura en Sistemas de Información
Tecnatura en Programación Web

Departamento de Informática

2020



Programación Procedural

La asignatura Programación Procedural pertenece al área “Algoritmos y Lenguajes” y tiene como objetivo introducir los conceptos de la programación procedural, incluir construcciones estándar de programación, estrategias de resolución de problemas y estructuras fundamentales de datos.

En el desarrollo de la misma se promueve que el estudiante interprete conceptos básicos de diseño e implementación de los lenguajes imperativos, y por otro lado que realice prácticas intensivas en un lenguaje imperativo, a través de programas que ofrezcan soluciones óptimas a una amplia gama de problemas. Se incluyen también contenidos inherentes a verificación y derivación de programas iterativos.

Teniendo en cuenta que en la materia correlativa de segundo año trabaja el paradigma orientado a objetos, en particular a través del uso de C#, se elige al lenguaje C para la discusión del modelo de computación procedural.

Así, través de lenguaje C se analiza la especificación e implementación de distintos tipos de datos, distintos tipos de operaciones y se desarrollan programas en el marco del diseño modular.

Estos Apuntes de Cátedra son el resultado de muchos años de trabajo de un equipo de cátedra comprometido con su trabajo, un especial agradecimiento a Mg. Adriana Valenzuela y Mg. Myriam Llarena por el invaluable aporte que realizaron a la cátedra Programación Procedural.

Redacción de Unidades teóricas a cargo de:

Dr. Mario Díaz

Lic. Laura Gutierrez

Redacción de Prácticos:

Lic. Cristina Vera

Lic. Daniela Villafañe

Lic. Gerardo Barud

Lic. Andrea Ferreyra

Compaginación:

Dr. Mario Díaz

Índice

PROGRAMACIÓN PROCEDURAL.....	3
PROGRAMA DE EXAMEN 2020.....	9
UNIDAD 1: VERIFICACIÓN Y DERIVACIÓN DE PROGRAMAS.....	11
INTRODUCCIÓN.....	11
PRECONDICIONES Y POSTCONDICIONES.....	11
VERIFICACIÓN VS DERIVACIÓN.....	12
VERIFICACIÓN.....	13
DERIVACIÓN.....	19
BIBLIOGRAFÍA.....	32
PRÁCTICO 1.....	33
VERIFICACIÓN Y DERIVACIÓN.....	33
UNIDAD 1: LENGUAJES PROCEDURALES.....	35
INTRODUCCIÓN.....	35
LENGUAJE DE PROGRAMACIÓN.....	35
LENGUAJES IMPERATIVOS.....	36
DEFINICIÓN DE UN LENGUAJE DE PROGRAMACIÓN.....	36
TRADUCTORES Y COMPUTADORAS SIMULADAS POR SOFTWARE.....	37
<i>Simulación de software (interpretación)</i>	37
<i>Traductores</i>	38
<i>Distintos tipos de traductores</i>	39
<i>Traducción e Interpretación</i>	40
IMPLEMENTACIÓN DE LENGUAJES.....	40
CRITERIOS DE DISEÑO DE LENGUAJES DE PROGRAMACIÓN.....	41
MODELOS DE COMPUTACIÓN.....	42
PARADIGMA DE PROGRAMACIÓN.....	42
CLASIFICACIÓN DE LOS PARADIGMAS.....	42
VARIANTES DE LOS PARADIGMAS DE PROGRAMACIÓN.....	43
<i>Programación imperativa o procedural</i>	43
<i>Programación declarativa</i>	43
LENGUAJE C.....	44
ETAPAS EN LA OBTENCIÓN DE PROGRAMA EJECUTABLE EN LENGUAJE C.....	44
BIBLIOGRAFÍA.....	46
UNIDAD 2: OBJETO DE DATOS.....	47
INTRODUCCIÓN.....	47
OBJETO DE DATOS.....	47
VARIABLES Y CONSTANTES.....	48
TIEMPO DE VIDA.....	50
TIPOS DE DATOS.....	51
ESPECIFICACIÓN E IMPLEMENTACIÓN DE UN TIPO DE DATOS.....	51

TIPOS DE DATOS ELEMENTALES	52
DECLARACIONES	54
VERIFICACIÓN DE TIPOS	55
<i>Verificación de tipos y lenguaje C</i>	56
<i>Conversión de tipos</i>	56
CLASIFICACIÓN	58
TIPOS DE DATOS ENTEROS	58
<i>Operación de asignación</i>	59
<i>Valores lvalue y rvalue</i>	59
TIPOS DE DATOS REALES	61
TIPOS DE DATOS BOOLEANOS	61
TIPO DE DATOS CARACTER	62
TIPO DE DATO PUNTERO (APUNTADOR)	63
TIPOS DE DATOS ESTRUCTURADOS	70
ESPECIFICACIÓN DE TIPO DE DATOS ESTRUCTURADOS	71
REPRESENTACIÓN DE TIPOS DE DATOS ESTRUCTURADOS	72
DECLARACIONES Y VERIFICACIONES DE TIPO	74
CLASIFICACIÓN DE LOS TIPOS DE DATOS ESTRUCTURADOS	74
ARREGLOS UNIDIMENSIONALES	74
ARREGLOS BIDIMENSIONALES Y MULTIDIMENSIONALES	77
ARREGLOS Y PUNTEROS EN LENGUAJE C	78
CADENAS DE CARACTERES EN C	80
REGISTROS	82
ACCESO A LOS MIEMBROS DE UNA ESTRUCTURA	84
OPERACIONES SOBRE ESTRUCTURAS	86
PUNTEROS A ESTRUCTURAS	89
ACCESO A LOS COMPONENTES DE UNA VARIABLE STRUCT	89
REGISTROS VARIANTES	91
STRUCT Y UNION EN LENGUAJE C	92
BIBLIOGRAFÍA	93
UNIDAD 3: FUNCIONES	95
INTRODUCCIÓN	95
DEFINICIÓN DE TIPOS	95
LENGUAJE C Y TYPEDEF	97
SISTEMA DE TIPOS	98
VENTAJAS DE LA DEFINICIÓN DE TIPOS	98
SUBPROGRAMAS	98
ESPECIFICACIÓN DE UN SUBPROGRAMA	99
IMPLEMENTACIÓN DE UN SUBPROGRAMA	100
DEFINICIÓN, INVOCACIÓN Y ACTIVACIÓN DE SUBPROGRAMAS	101
FUNCIONES EN LENGUAJE C	103
DEFINICIÓN	103
INVOCACIÓN O LLAMADA A UNA FUNCIÓN	106
DECLARACIÓN DE UNA FUNCIÓN O PROTOTIPO DE FUNCIÓN	108
ORGANIZACIÓN DE LA MEMORIA EN C	109
EJECUCIÓN DE UN PROGRAMA EN C	110
TIPOS DE ALMACENAMIENTO	112
VARIABLES AUTOMÁTICAS	112

VARIABLES EXTERNAS	113
VARIABLES ESTÁTICAS	114
DECLARACIONES, BLOQUES Y ALCANCE	116
DECLARACIONES – SU IMPORTANCIA DURANTE LA TRADUCCIÓN	116
BLOQUES EN LENGUAJE C.	116
ALCANCE DE UN VÍNCULO EN LENGUAJE C:	116
APERTURA EN EL ALCANCE:.....	117
VISIBILIDAD DE UNA DECLARACIÓN:	117
PASAJE DE PARÁMETROS A UNA FUNCIÓN	118
PASAJE POR VALOR	118
PASAJE DE DIRECCIONES	120
PASAJE POR REFERENCIAS	121
FUNCIONES QUE DEVUELVEN MÁS DE UN RESULTADO	122
ARREGLOS COMO PARÁMETROS DE FUNCIONES	123
PASAJE CONSTANTE EN ARREGLOS	126
TRADUCCIÓN Y TABLA DE SÍMBOLOS EN LENGUAJE C	127
OPERACIONES CON LA TS	127
MANEJO DE LA TS EN PILA	132
BIBLIOGRAFÍA:.....	133
PRACTICO 2.....	134
LENGUAJE C - FUNCIONES - TABLA DE SÍMBOLO	134
UNIDAD 4: RECURSIVIDAD.....	139
INTRODUCCIÓN.....	139
FUNCIONES RECURSIVAS.....	139
FUNCIONES RECURSIVAS QUE DEVUELVEN MÁS DE UN RESULTADO	145
RECURSIVIDAD USANDO ARREGLOS.....	146
ORDENACIÓN RÁPIDA (QUICKSORT).....	149
RECURSIÓN VERSUS ITERACIÓN	151
PRACTICO 3.....	152
RECURSIVIDAD.....	152
UNIDAD 5: ESTRUCTURAS DINÁMICAS.....	157
INTRODUCCIÓN.....	157
EL MONTÍCULO O MONTÓN(HEAP)	157
MANEJO DEL MONTÍCULO EN LENGUAJE C	158
LENGUAJE C: VARIABLES DINÁMICAS SIMPLES	160
LENGUAJE C: ARREGLOS DINÁMICOS	162
ARREGLOS DINÁMICOS USADOS COMO CADENA DE CARACTERES	166
ARREGLOS DE CADENAS DINÁMICAS	167
LISTAS.....	170
IMPLEMENTACIÓN DE LISTAS	170
IMPLEMENTACIÓN DE LISTAS MEDIANTE ARREGLOS.....	170
IMPLEMENTACIÓN DE LISTAS MEDIANTE PUNTEROS	171
DEFINICIÓN Y DECLARACIÓN DE LAS DISTINTAS IMPLEMENTACIONES	172
MANIPULACIÓN DE LISTAS ENLAZADAS	173
<i>Creación</i>	<i>173</i>
<i>Inserción en una lista</i>	<i>173</i>

<i>Gestión de almacenamiento.....</i>	<i>175</i>
<i>Recorrido de la lista.....</i>	<i>177</i>
<i>Búsqueda de una componente de una lista.....</i>	<i>177</i>
<i>Modificación de una componente de la lista.....</i>	<i>178</i>
<i>Inserción de un nodo en cualquier lugar de la lista.....</i>	<i>180</i>
<i>Supresión de un elemento en una lista.....</i>	<i>182</i>
<i>Ordenamiento de una lista.....</i>	<i>184</i>
<i>Eliminación de todas las componentes de una lista.....</i>	<i>184</i>
LISTAS ENLAZADAS QUE ALMACENAN DATOS ESTRUCTURADOS	185
PROBLEMAS DE GESTIÓN DE ALMACENAMIENTO.....	186
<i>Referencias Bamboleantes</i>	<i>186</i>
<i>Basura</i>	<i>186</i>
MANIPULACIÓN DE LISTAS CON FUNCIONES RECURSIVIDAD.....	187
ARREGLO DE LISTAS.....	188
BIBLIOGRAFÍA.....	190
PRACTICO 4	191
ESTRUCTURAS DINÁMICAS.....	191
UNIDAD 6: ARCHIVOS.....	195
INTRODUCCIÓN	195
DEFINICIÓN	195
ORGANIZACIÓN DE ARCHIVOS	195
CLASIFICACIÓN DE ARCHIVOS	196
VENTAJAS Y DESVENTAJAS DEL USO DE ARCHIVOS.....	197
DECLARACIÓN DE UN ARCHIVO EN EL LENGUAJE C.....	197
OPERACIONES BÁSICAS EN EL MANEJO DE ARCHIVOS	198
ARCHIVOS ORGANIZADOS SECUENCIALMENTE	200
ARCHIVOS DE CARACTERES.....	200
ARCHIVOS DE TEXTOS	202
ARCHIVOS SIN FORMATO	206
ARCHIVOS ORGANIZADOS SECUENCIALMENTE CON ACCESO DIRECTO.....	213
ELIMINACIÓN DE INFORMACIÓN DEL ARCHIVO.....	218
CÁLCULO DE LA LONGITUD DE UN ARCHIVO.....	220
BIBLIOGRAFÍA.....	220
PRACTICO 5	221
ARCHIVOS	221
UNIDAD 7: DISEÑO MODULAR.....	225
INTRODUCCIÓN	225
DISEÑO MODULAR	226
DISEÑO Y ARQUITECTURA DE SOFTWARE.....	226
OBJETIVOS DEL DISEÑO MODULAR.....	227
DESCOMPOSICIÓN MODULAR	227
INDEPENDENCIA FUNCIONAL.....	227
OCULTAMIENTO DE LA INFORMACIÓN	228
COHESIÓN	228
ACOPLAMIENTO	229
COMPREENSIBILIDAD	229
ADAPTABILIDAD	229

MODULARIDAD.....	230
COHESIÓN MODULAR Y ACOPLAMIENTO INTER-MODULAR	231
COHESIÓN MODULAR	232
COHESIÓN POR COINCIDENCIA.....	233
COHESIÓN LÓGICA	233
COHESIÓN TEMPORAL.....	234
COHESIÓN DE PROCEDIMIENTO.....	235
COHESIÓN SECUENCIAL	237
COHESIÓN FUNCIONAL.....	237
COHESIÓN ABSTRACCIONAL	238
ÁRBOL DE DECISIÓN PARA HALLAR LA COHESIÓN DE UN MÓDULO	239
ACOPLAMIENTO INTER - MODULAR.....	243
ACOPLAMIENTO POR CONTENIDO.....	244
ACOPLAMIENTO COMÚN	244
ACOPLAMIENTO EXTERNO	245
ACOPLAMIENTO DE CONTROL.....	246
<i>Inversión de Autoridad.....</i>	246
ACOPLAMIENTO ESTAMPADO	247
ACOPLAMIENTO DE DATOS	247
<i>Pasaje de Punteros como Parámetros</i>	248
ACOPLAMIENTO NORMAL.....	248
BIBLIOGRAFÍA	249
PRACTICO 6.....	250
DISEÑO MODULAR	250

Programa de Examen 2020

Unidad 1

a) Construcción de programas. Introducción. Verificación y Derivación. Verificación de una iteración. Derivación de algoritmos. Cálculo del Invariante. Ejemplos de Derivación de Programas.

b) Introducción. Lenguaje de programación. Algunas cuestiones de diseño e implementación: computadora distintos tipos. Traductores y computadoras simuladas por software. Modelos de computación. Lenguajes imperativos: Definición de un lenguaje de programación. Concepto de estado de máquina. Implementación de un lenguaje de programación. Eficiencia y Regularidad. Ejemplificación en lenguaje C.

Unidad 2

Objetos de Datos. Variables y Constantes. Tiempo de Vida. Enlace (ligadura) y Tiempo de enlace. Tipos de Datos. Especificación e Implementación.

a) Tipos de Datos Elementales. Especificación e Implementación. Representación de almacenamiento. Implementación de algoritmos y procedimientos que definen las operaciones. Declaraciones y Verificación de tipo. Verificación de tipos en lenguaje C. Conversión y coerción de tipos. Tipo de datos Entero. Semántica de la operación de Asignación: Valor l y Valor r. Números Reales de punto flotante. Enumeraciones. Tipos Booleanos. Tipos de datos caracteres.

Tipo Apuntador. Punteros a variables simples en lenguaje C. Operadores de apuntadores. Inicialización. Asignación de punteros. Ejercicios de Aplicación.

b) Tipos De Datos Estructurados: Introducción. Especificación de Tipo de Datos Estructurados. Implementación de Tipos de Datos Estructurados. Vectores. Implementación de Operaciones sobre Estructuras de Datos. Declaraciones y Verificaciones de Tipo. Arreglos Bidimensionales y Multidimensionales. Arreglos y Punteros en Lenguaje C.

Cadenas De Caracteres. Distintas formas. Cadenas de caracteres en C. Uso De Funciones de Cadena de la Biblioteca Estándar.

Registros: Especificación e Implementación. Operación de Selección de Componentes. Manejo de Struct en Lenguaje C. Punteros a Struct.

Registros Variantes: Implementación. Union y Struct en Lenguaje C. Ejercicios de Aplicación.

Unidad 3

Encapsulamiento a través de subprogramas. Subprogramas: Especificación e implementación de subprogramas. Definición, invocación y activación de subprogramas.

Concepto de función en C: Función main(). Declaración y definición de funciones. Organización de memoria: Almacenamiento Estático, Pila y Montículo. Ejecución de un programa en C.

Pasajes de parámetros: Pasajes de parámetros: valor, constante, por dirección. Variables referenciadas. Pasaje de parámetro por referencia. Arreglos como parámetros. Funciones que devuelven más de un valor. Ejercicios de aplicación.

Traducción y Tabla de Símbolos (TS): Compilación de un programa fuente. Lenguaje C como un lenguaje estructurado en bloque. Importancia de las declaraciones en C. Alcance de un vínculo en lenguaje C. Generación de la TS en un programa en C. Ejercicios de aplicación.

Clasificación de las variables según alcance y tiempo de vida: Variables automáticas, estáticas y externas. Ejercicios de aplicación.

Unidad 4

Recursividad. Funciones Recursivas en C. Definición. Caso base y caso general. Distintas combinaciones. Manejo de memoria (pila). Aplicaciones. Recursión e Iteración. Ejercicios de aplicación.

Unidad 5

Estructuras dinámicas. Almacenamiento estático y dinámico. Variables dinámicas. El montículo: Operaciones sobre el montículo o heap. Manejo del montículo en lenguaje C: funciones malloc y free. Variables dinámicas simples. Basura y referencias desactivadas. Arreglos dinámicos en lenguaje C. Arreglos dinámicos uni y bi-dimensionales. Mapa de memoria. Cadena de caracteres dinámicas. Arreglo de cadenas dinámicas. Ejercicios de Aplicación.

Listas: listas secuenciales y listas enlazadas. Manipulación de listas enlazadas: creación, inserción. Búsqueda, recorrido, modificación, supresión de elementos. Ordenamiento y eliminación de una lista. Gestión de almacenamiento. Problemas de gestión de almacenamiento: referencias bamboleantes y basura. como cola. Recursividad en listas. Ejercicios de Aplicación.

Unidad 6

Archivos. Concepto de archivo en C. Distintas clasificaciones. Archivos secuenciales: Archivos de caracteres y sin formato. Acceso secuencial de archivos: Funciones para la creación y utilización de archivos secuenciales. Acceso directo de archivos secuenciales. Funciones para el uso de archivos. Ejercicios de aplicación.

Unidad 7

Metodología de Diseño Modular: Concepto de módulo en C. Diseño Modular: Concepto de módulo. Ventajas del Diseño Modular. Independencia Funcional. Criterios que miden la independencia modular. Cohesión y Acoplamiento Modular. Distintos Tipos.

Construcción de programas que incluyan manejo avanzado de archivos: Corte de control y merge. Ejercicios de aplicación.

Dr. Mario Diaz

Prof. Titular Programacion Procedural

Unidad 1: Verificación y Derivación de Programas

Introducción

En esta primera parte del Tema 1, profundizaremos los conceptos trabajados sobre verificación de programas, vistos en la asignatura Algoritmos y Resolución de Problemas. Aplicaremos estos conceptos a programas que incluyan estructuras repetitivas e introduciremos los elementos necesarios para abordar el tema de derivación de programas.

Como se ha visto, para la verificación de programas imperativos se utiliza un sistema de reglas basado en la **tripla de Hoare**. La lógica de Hoare es una extensión de la lógica de predicados de primer orden (reglas de inferencia) para razonar sobre la corrección o construcción de algoritmos imperativos.

Tripla de Hoare: Para especificar formalmente un programa se utiliza una expresión del tipo $\{P\} A \{Q\}$, para indicar que si P es cierto antes de la ejecución del programa y dicho programa termina, entonces Q es cierto tras la ejecución del programa. En el programa la precondition P y la postcondition Q se especifican mediante fórmulas denominadas aserciones que relacionan las entradas y salidas del programa. Se garantiza que si la entrada actual satisface las restricciones de entrada (precondiciones) la salida satisface las restricciones de salida (postcondiciones).

Un **Aserto** es un predicado donde se expresa la relación que tienen que cumplir los valores de ciertas variables en un momento determinado de la ejecución de un algoritmo. La precondition es un aserto que se debe cumplir al inicio, y la poscondition es un aserto que se debe cumplir cuando finaliza un algoritmo. Un aserto se escribe entre los signos " $\{ \}$ ".

El Aserto $\{\text{True}\}$ o $\{\text{Verdadero}\}$ representa el conjunto universal de estados, es decir todos los posibles valores de las variables. El aserto $\{\text{False}\}$ o $\{\text{Falso}\}$ representa el conjunto vacío de estados, esto significa que no se verifica para ningún valor de las variables.¹

A partir de lo expuesto, se interpreta que una **Especificación Formal de Algoritmos** es un conjunto de instrucciones que tiene dos asertos: la Precondition y la Poscondition. Tiene como **objetivo** expresar en forma correcta y sin ambigüedades que es lo que debe hacer un algoritmo y bajo qué condiciones se puede ejecutar.

Por ejemplo en el siguiente algoritmo que permita sumar los elementos de un arreglo, el aserto $\{P\}$ refiere al tamaño del arreglo y la postcondition $\{Q\}$ expresa formalmente la sumatoria requerida como salida del algoritmo, S es el algoritmo.

Especificación

```
Const int N
int a[0..N),m
{P: N>0}
  S
{Q: m=<math>\sum_{i=0}^{N-1} a[i]>math>}
```

Precondiciones y Postcondiciones

Durante el desarrollo de una tarea es importante determinar qué **condiciones** deben darse al comienzo o entrada de la tarea para que se cumplan las que se establece que deben ser ciertas al finalizar la misma, es decir a la salida.

¹M.^a Teresa González de Lena Alonso, Isidoro Hernán Losada y otros (2005) Introducción a la programación: problemas resueltos en Pascal

Los asertos que deben cumplirse a la entrada de una tarea reciben el nombre de Precondiciones. Si la operación, tarea o instrucción se realiza sin que la precondición se cumpla no tendremos garantía de los resultados obtenidos.

Los asertos acerca de los resultados que se esperan a la salida, se llaman postcondiciones.

Verificación vs Derivación

Es importante distinguir entre verificar y derivar o deducir programas. En ambos casos los programas son tratados como fórmulas lógicas.

Verificar consiste en demostrar que el programa construido es correcto respecto de la especificación dada.

Derivar un programa permite construir un programa a partir de su especificación, de forma que se obtiene un algoritmo correcto por construcción. –

Se debe distinguir entre corrección total y parcial.

Corrección parcial: se dice que $\{P\} A \{Q\}$ es parcialmente correcto si comienza en un estado que satisface $\{P\}$ y en caso de que termine satisface $\{Q\}$.

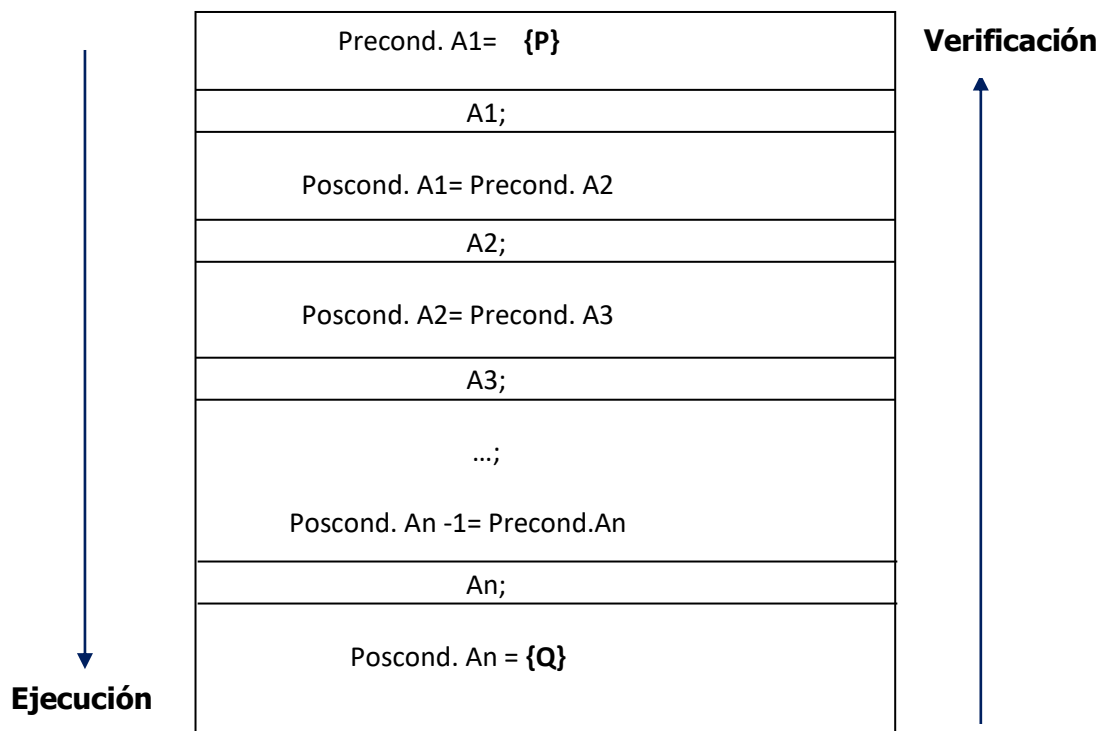
Corrección total: Se da cuando un código además de ser correcto parcialmente, termina.

Los algoritmos sin bucles siempre terminan por lo que la corrección parcial implica la corrección total. Esta distinción es esencial sólo en el caso de códigos que incluyan bucles o recursiones.

En la asignatura Algoritmos y Resolución de Problemas se ha trabajado la **verificación** de algoritmos que utilizan estructuras de secuencia y alternativas, como así también sentencias de asignación.

Vimos que dada una especificación $\{P\} A_1; A_2; A_3; \dots; A_n \{Q\}$ la verificación de la misma se inicia desde la postcondición, y a partir de ahí se deduce la precondición. Esto es, el código se verifica en sentido contrario a como se ejecuta.

Para $\{P\} A_1; A_2; A_3; \dots; A_n \{Q\}$ se cumple:



El programa A actúa como una **función de estados en estados**: comienza su ejecución en un estado inicial válido, descrito por el valor de los parámetros de entrada, y termina en un estado final en el que los parámetros de salida contienen los resultados esperados

Es necesario recordar que **salvo en el caso de la iteración, las otras sentencias utilizan sólo la precondición más débil como método para verificar y derivar programas.**

La **precondición más débil** es el conjunto de todos los estados tales que desde ellos y ejecutando el programa A se llega a Q.

Cuando se trabaja con algoritmos iterativos, es necesario introducir el concepto de **invariante**.

Verificación

Comenzaremos a trabajar la verificación de algoritmos iterativos, para luego abocarnos a la **derivación** de algoritmos.

Iteración: La instrucción de iteración corresponde a la siguiente sintaxis en pseudocódigo:

Mientras (B)

A

finMientras

Siendo **A** una instrucción (simple o compuesta) y **B** una expresión booleana.

La instrucción A o cuerpo del ciclo se ejecuta mientras la condición B es verdadera, se vuelve a evaluar la expresión B, repitiendo el proceso hasta que la condición B es falsa, caso en el que el ciclo termina.

Para verificar un bucle necesitamos un predicado llamado **invariante** que describa los distintos estados por los que pasa el bucle, estableciendo la relación que existe entre las variables que participan en él. *Es decir que si bien las variables cambian su valor, se mantienen invariables ciertas relaciones entre ellas.*

El invariante de un bucle es un predicado que captura las características que no varían al ejecutarse un bucle, es el precursor de la postcondición por lo que debe ser parecido a ella.

Por tanto para especificar una iteración además de la pre y postcondición, es necesario introducir un nuevo predicado llamado invariante.

{P} precondición

Mientras (B) {I} invariante

A

finMientras

{Q} postcondición

El invariante es un predicado lógico que tiene la propiedad de ser cierto antes de entrar al bucle, tras cada iteración (durante el bucle) y después del bucle.

Estas condiciones que debe cumplir el invariante, se pueden expresar como sigue:

- 1) **{P} A₀ {I}** El invariante se satisface antes de la primer iteración , A₀ representa los valores iniciales de las variables
- 2) **{I ^ B} A {I}** El invariante se mantiene al ejecutar el cuerpo A del bucle.
- 3) **{I ^ ~B} => {Q}** El invariante se cumple al salir del bucle (cuando B es falsa) y debe llevarnos a la postcondición.

4) **Además se debe probar que el algoritmo termina.** Para ello debemos buscar una **función de cota C** que tome valores enteros. Esta función se construye a partir de una expresión con todas o algunas de las variables que intervienen en la expresión de la condición del bucle y que son modificadas en el cuerpo del bucle.

La idea es que esa expresión debe dar idea del número de iteraciones que quedan por realizar cada vez que se ejecuta el ciclo, de forma que se cumpla que:

- La función cota es mayor o igual que 0 cuando se cumple la condición B

4-a) $\{I \wedge B\} \Rightarrow C \geq 0$

- La función cota decrece al ejecutar el cuerpo A del bucle.

4-b) $\{I \wedge B \wedge C=T\} A \{C < T\}$

Esta **función C** es una cota superior al número de iteraciones que quedan por realizar, de ahí el nombre de función cota. Sintetizando, la regla para la verificación de una instrucción iterativa es:

$\{P\} A_0 \{I\}$
$\{I \wedge B\} A \{I\}$
$\{I \wedge \sim B\} \Rightarrow \{Q\}$
$\{I \wedge B\} \Rightarrow C \geq 0$
$\{I \wedge B \wedge C=T\} A \{C < T\}$
<hr/>
$\{P\} \text{ Mientras } (B) \text{ A finMientras } \{Q\}$

Ejemplo: Suma elementos de un arreglo- de derecha a izquierda-.

El siguiente algoritmo calcula la suma de los elementos de un arreglo, con un recorrido de derecha a izquierda. Verificar que cumple la especificación dada suponiendo que el invariante es:

I: $x = \sum_{i: n \leq i \leq N} V[i] \wedge 0 \leq n \leq N^2$

Constante N

entero V [N], x, n //declaración de variables

$\{P: N \geq 0\}$

n=N

x=0

Mientras (n ≠ 0)

x = x+V[n-1]

n = n-1

finMientras

$\{Q : x = \sum_{i: 0 \leq i < N} V[i] \wedge\}$

En este algoritmo la condición de entrada del bucle es (n ≠ 0), su negación es la condición de salida, que permite que el bucle termine. La relación **x=x+V[n-1]** es la **invariante del bucle** .

² Las variables x y n se llaman variables de programa, ya que describen el estado del programa. La variable i, que aparece en un cuantificador se llama variable de cuantificación, variable dummy o ficticia.

Probemos que se cumple las reglas de verificación:

1) El invariante se satisface antes de la primer iteración $\{P\} A_0 I$

Se debe probar que la terna $\{P\} n=N; x=0; \{I\}$ es correcta, para ello se debe demostrar que $\{P\} \Rightarrow \text{pmd}(n=N; x=0, I)$

Calculemos la precondition más débil (pmd) utilizando las reglas de verificación de la composición secuencial y de la asignación.

$$\begin{aligned} \text{pmd}(n=N; x=0, I) &\equiv ((I) \text{ }) \text{ }_x^0 \text{ }_n^N \\ (I) \text{ }_x^0 &\equiv ((0 \leq n \leq N) \wedge (x = \sum_{i: n \leq i < N} V[i])) \text{ }_x^0 \equiv (0 \leq n \leq N \wedge 0 = \sum_{i: n \leq i < N} V[i]) \\ (I) \text{ }_n^N &\equiv (0 \leq n \leq N \wedge 0 = \sum_{i: n \leq i < N} V[i]) \text{ }_n^N \\ &\equiv ((0 \leq n \leq N \wedge 0 = \sum_{i: n \leq i < N} V[i])) \equiv n \geq 0 = \{P\} \end{aligned}$$

Por ser el rango vacío, la sumatoria es 0 (elemento neutro) por lo que el segundo término es True

Dado que $\{P\} \Rightarrow \text{pmd}$ se ha demostrado que las inicializaciones $n=N$ y $x=0$ son correctas.

2) El invariante se mantiene al ejecutar el cuerpo A del bucle $\{I \wedge B\} A \{I\}$

Para demostrar que la terna $\{I \wedge B\} A \{I\}$ se verifica, se debe probar que $\{I \wedge B\} \Rightarrow \text{pmd}(A, I)$

$$\begin{aligned} \text{pmd}(x=x + V[n-1]; n=n-1, I) \\ &\equiv ((0 \leq n \leq N \wedge x = \sum_{i: n \leq i < N} V[i])) \text{ }_n^{n-1} \text{ }_x^{x+V[n-1]} \\ &\equiv ((0 \leq n-1 \leq N \wedge x = \sum_{i: n-1 \leq i < N} V[i])) \text{ }_x^{x+V[n-1]} \\ &\equiv (0 \leq n-1 \leq N) \wedge (x + V[n-1] = \sum_{i: n-1 \leq i < N} V[i]) \\ &\equiv (0 \leq n-1 \leq N) \wedge (x + V[n-1] = V[n-1] + \sum_{i: n \leq i < N} V[i]) \quad (1) \end{aligned}$$

Hemos descompuesto la sumatoria separando el primer término de la sumatoria.

Ahora debemos probar que $I \wedge B$ es más fuerte que la aserción (1), es decir $I \wedge B \Rightarrow (1)$

Recordemos que $I: (0 \leq n \leq N) \wedge (x = \sum_{i: n \leq i < N} V[i])$ y $B: n \neq 0$.

Por tanto, de $I \wedge B$ se puede inferir, considerando en forma separada las condiciones relacionadas con la variable n y la que corresponde a la variable x :

- a) $(0 \leq n \leq N) \wedge (n \neq 0) \equiv (0 < n \leq N) \equiv (0 \leq n-1 < N) \Rightarrow 0 \leq n-1 \leq N$
- b) $(x = \sum_{i: n \leq i < N} V[i]) \equiv (x + V[n-1] = V[n-1] + \sum_{i: n \leq i < N} V[i])$

Luego, $I \wedge B \Rightarrow (1)$ como queríamos demostrar.

Esto significa que las acciones que constituyen el cuerpo del ciclo son correctas.

3) El invariante se cumple al salir del bucle (cuando B es falsa) y debe llevarnos a la postcondición $I \wedge \sim B \Rightarrow Q$.

$$I \wedge \sim B \equiv I \wedge \sim (n \neq 0) \equiv (0 \leq n \leq N) \wedge (x = \sum_{i: n \leq i < N} V[i]) \wedge (n = 0)$$

$$\equiv (N \geq 0) \wedge (x = (\sum_{i: 0 \leq i < N} V[i])) \Rightarrow x = (\sum_{i: 0 \leq i < N} V[i]) \equiv Q$$

4) Definamos la función cota y probar que el algoritmo termina

Para probar que la iteración termina, debe encontrarse la función cota. Como se expuso, esta función debe definirse en función de las variables que aparecen en la condición B. Como en B la única variable que aparece es n, entonces podemos tomar:

$$C(n) = n; \text{ n natural } n \geq 0$$

Se elige n como cota ya que decrece en cada iteración e indica el número de iteraciones que quedan por realizar en una iteración.

Para que el algoritmo termine deben cumplirse las condiciones siguientes:

a - La función cota es mayor o igual que 0 cuando se cumple condición B

Esto es: $I \wedge B \Rightarrow C \geq 0$

Debemos probar $I \wedge n \neq 0 \Rightarrow n \geq 0$

$$(I \wedge n \neq 0) \Rightarrow (0 \leq n \leq N) \wedge n \neq 0 \Rightarrow n > 0 \Rightarrow n \geq 0$$

b - La función cota decrece al ejecutar el cuerpo A del bucle.

Esto es probar que es correcta la siguiente especificación:

$$\{I \wedge B \wedge C = T\} A \{C < T\} \equiv \{I \wedge n \neq 0 \wedge n = T\} x = x + v[n-1]; n = n-1 \{n < T\}$$

Para probar que esta terna es correcta se debe demostrar que

$$\{I \wedge B \wedge C = T\} \Rightarrow \text{pmd}(A, n < T)$$

Utilizando las reglas de verificación de la composición secuencial y de la asignación, se tiene:

$$\text{pmd}(A, n < T) \equiv \text{pmd}(x = x + v[n-1]; n = n-1, n < T)$$

$$\equiv (n < T) \begin{pmatrix} n-1 \\ n \end{pmatrix} \begin{matrix} x + v[n-1] \\ x \end{matrix} \equiv (n-1 < T) \begin{matrix} x + v[n-1] \\ x \end{matrix} \equiv n-1 < T \quad (2)$$

Debemos probar que $I \wedge n \neq 0 \wedge n = T \Rightarrow n-1 < T$

$$(I \wedge n \neq 0) \wedge (n = T) \Rightarrow n = T \equiv n-1 < T \quad (2)$$

De esta manera se prueba que la cota decrece, es decir el ciclo termina su ejecución.

A partir de los 4 pasos anteriores hemos verificado que este algoritmo que suma los elementos de un arreglo de N enteros, a partir del último elemento, cumple con la especificación dada.

Ejempl: Verificación algoritmo potencia

Demostrar que el siguiente programa que calcula la potencia de base **a** y exponente **b**, es correcto; siendo a, b y p números enteros.

$\{ P: x=a \wedge y=b \wedge b \geq 0 \mid b \in \mathbb{Z} \}$

$p=1;$

Mientras($y \neq 0$)

$p = p * x;$

$y = y - 1;$

finMientras

$\{ Q: p=a^b \}$

Dado que la variable **a** no cambia su valor y la variable **y** comienza con el valor **b** y finaliza en cero, se puede elegir como invariante: $\{ I \equiv x=a \wedge y \geq 0 \wedge p=a^{b-y} \}$

Entonces se debe probar:

1) El invariante se satisface antes de la primer iteración $\{ P \mid A_0 \mid \{ I \}$

Se debe probar $\{ P \mid p=1 \mid \{ I \}$, es correcta, es decir $\{ P \} \Rightarrow \text{pmd}(p=1, I)$

$$\text{Pmd}(p=1, I) \equiv (I) \stackrel{p}{=} x=a \wedge y \geq 0 \wedge p=a^{b-y} \equiv \left(\frac{1}{p} = a \wedge y \geq 0 \wedge 1 = a^{b-y} \right)$$

$$\text{Si } b-y=0 \Rightarrow b=y$$

$$\text{Pmd}(p=1, I) \equiv (x=a \wedge y \geq 0 \wedge b=y) \equiv \{ P \}$$

Por tanto la inicialización $p=1;$ es correcta.

2) El invariante se mantiene al ejecutar el cuerpo A del bucle $\{ I \wedge B \mid A \mid \{ I \}$

Para demostrar que la terna $\{ I \wedge B \mid A \mid \{ I \}$ es correcta, se debe probar que $\{ I \wedge B \} \Rightarrow \text{pmd}(A, I)$

$$\begin{aligned} \text{pmd}(p=p * x; y=y-1, I) &\equiv ((x=a \wedge y \geq 0 \wedge p=a^{b-y})_{y^{y-1}})_p^{p*x} \\ &\equiv ((x=a \wedge y-1 \geq 0 \wedge p=a^{b-(y-1)})_p^{p*x}) \equiv ((x=a \wedge y-1 \geq 0 \wedge p*x=a^{b-(y-1)}) \equiv \\ &\equiv ((x=a \wedge y \geq 1 \wedge p*x=a^{b-y+1})) \equiv (x=a \wedge y \geq 1 \wedge p*x=a*a^{b-y}) \end{aligned}$$

$$\text{pmd} \equiv (x=a \wedge y \geq 1 \wedge p*x = x*a^{b-y}) \equiv (x=a \wedge y \geq 1 \wedge p = a^{b-y}) \quad (1)$$

Debemos demostrar que $\{ I \wedge B \} \Rightarrow (1)$

$$\{ I \wedge B \} \equiv (x=a \wedge y \geq 0 \wedge p=a^{b-y}) \wedge (y \neq 0) \equiv (x=a \wedge y > 0 \wedge p=a^{b-y}) \equiv$$

$$(x=a \wedge y \geq 1 \wedge p=a^{b-y}) \equiv (1)$$

Por tanto las acciones que están dentro del cuerpo del bucle son correctas.

3) El invariante se cumple al salir del bucle (cuando B es falsa) y debe llevarnos a la postcondición $I \wedge \neg B \Rightarrow Q$.

$$I \wedge \neg B \equiv I \wedge (y=0) \equiv (x=a \wedge y \geq 0 \wedge p=a^{b-y}) \wedge (y=0) \equiv (x=a \wedge y=0 \wedge p=a^b) \Rightarrow p=a^b \equiv \{Q\}$$

4) Definamos la función cota y probar que el algoritmo termina

Para probar que la iteración termina, debe encontrarse la función cota. Ésta debe definirse en función de las variables que aparecen en la condición B y debe determinar la cantidad de iteraciones que falta realizar en cada iteración. Como en B la única variable que aparece es y, que cumple con las condiciones señaladas, podemos tomar:

$$C = y$$

Para que el algoritmo termine deben cumplirse las condiciones siguientes

a.-La función cota es mayor o igual que 0 cuando se cumple condición B

$$\text{Esto es: } I \wedge B \Rightarrow C \geq 0$$

Debemos probar $I \wedge y \neq 0 \Rightarrow y \geq 0$

$$(I \wedge y \neq 0) \equiv (x=a \wedge y \geq 0 \wedge p=a^{b-y}) \wedge y \neq 0 \Rightarrow y > 0 \Rightarrow y \geq 0$$

b.-La función cota decrece al ejecutar el cuerpo A del bucle.

Se debe probar que es correcta la siguiente especificación: $\{I \wedge B \wedge C=T\} A \{C < T\}$

Esta terna es correcta si se verifica que:

$$\{I \wedge B \wedge C=T\} \Rightarrow \text{pmd}(A, y < T)$$

Utilizando las reglas de verificación de la composición secuencial y de la asignación, se tiene:

$$\begin{aligned} \text{pmd}(A, y < T) &\equiv \text{pmd}(p=p * x; y=y-1, y < T) \equiv ((y < T)_{y=y-1}) \equiv \left(\frac{p * x}{y-1} < T \right)_{\frac{p * x}{p}} \\ &\equiv (y-1 < T) \quad (1) \end{aligned}$$

Debemos probar $\{I \wedge B \wedge C=T\} \Rightarrow (1)$

$$(I \wedge y \neq 0 \wedge C=T) \equiv (x=a \wedge y \geq 0 \wedge p=a^{b-y}) \wedge (y \neq 0) \wedge (y=T) \Rightarrow (y=T) \equiv (y-1 < T)$$

Derivación

La derivación de un algoritmo es un proceso que permite construir las instrucciones a partir de la especificación preocupándose a lo largo de este proceso de su corrección. Así, al terminar la derivación el algoritmo encontrado será correcto o sea cumple su especificación, por construcción.

En la especificación, la postcondición describe el estado que se desea alcanzar, de ahí que el proceso de construcción está guiado por la postcondición.

Si recordamos, la regla de verificación de algoritmos es la siguiente:

$\{P\} A_0 \{I\}$ *El invariante se satisface antes de la primera iteración*
 $\{I \wedge B\} A \{I\}$ *El invariante se mantiene al ejecutar el cuerpo A del bucle*
 $\{I \wedge \neg B\} \Rightarrow \{Q\}$ *El invariante se cumple al salir del bucle*
 $\{I \wedge B\} \Rightarrow C \geq 0$ *La función cota es ≥ 0 cuando se cumple condición B*
 $\{I \wedge B \wedge C = T\} A \{C < T\}$ *La función cota decrece al ejecutar el cuerpo del bucle*

 $\{P\} \text{Mientras } (B) \ A \ \text{finMientras } \{Q\}$

Para el caso de la derivación, se conoce únicamente la precondition y la postcondición, por tanto debe determinarse:

- 1- El invariante I.
- 2- La condición B del ciclo.
- 3- Las sentencias de inicialización de variables, esto es las condiciones iniciales A_0 .
- 4- El cuerpo A del ciclo: el cuerpo está constituido por dos instrucciones fundamentales A_1 y A_2 . La Instrucción A_1 es parte del cuerpo del bucle y se encarga de mantener cierto el invariante, A_2 representa la instrucción que hace que las variables que intervienen en la expresión B avancen hacia la condición de salida del bucle.
- 5- La función cota.

Por tanto a la hora de derivar un bucle, se propone seguir el siguiente esquema³:

$\{P\}$

A_0 condiciones iniciales

$\{I, C\}$ I es el invariante y C es la cota

Mientras (B)

A_1 instrucción del cuerpo del bucle que se encarga de mantener cierto el invariante

A_2 representa las instrucciones que hacen que las variables que intervienen en la expresión B avancen hacia la condición de salida del bucle, es decir hacia $\neg B$ (hacen que decrezca la función cota)

finMientras

$\{Q\}$

A continuación describiremos como se realizan los pasos enumerados.

³Adaptación de Narciso Martí Oliet

Derivación de bucles a partir de su especificación

Pasos 1 y 2 - Obtener el invariante I y la condición del bucle a partir de la postcondición

Como sabemos:

$$\boxed{I \wedge \sim B \Rightarrow} \quad \text{garantiza el cumplimiento de la postcondición al finalizar el ciclo}$$

Existen 2 posibilidades:

- Si la **postcondición está en forma conjuntiva**, se pueden ensayar distintas posibilidades eligiendo una parte ella como invariante y el resto como negación de la condición del bucle.
- Se puede **introducir una nueva variable que sustituya en la postcondición a una constante**. En este caso el invariante será la postcondición generalizada con esa nueva variable y la condición del bucle será la negación de la igualdad entre la variable y la constante sustituida

Paso 3 - Asignar los valores iniciales de las variables (Construir A_0)

Para asegurar que el invariante sea cierto al comienzo del bucle, se debe cumplir:

$$\boxed{\{P\} A_0 \{I\}} \quad \text{Garantiza el cumplimiento del invariante al comenzar el ciclo}$$

Paso 4 - Obtener el cuerpo del bucle A

Esto es un programa que mantenga el invariante y permita que la función cota decrezca.

$\{I \wedge B\} A \{I\}$ donde A es el cuerpo del bucle y está formado por A_1 y A_2

Para ello se sugiere:

- Diseñar la instrucción A_2 para *avanzar*, y que *decrezca el valor de la función cota*.

Para esto, se debe construir un aserto o **condición intermedia R** que haga correcta la condición $\{R\} A_2 \{I\}$; es decir habrá que determinar la precondición más débil de A_2 respecto de I, $\text{pmd}(A_2, I)$

- Diseñar A_1 , comparando $\{I \wedge B\}$ con $\{R\}$, de manera que se cumpla $\{I \wedge B\} A_1 \{R\}$

A_1 representa las instrucciones del cuerpo que se encargan de mantener cierto el invariante.

Las condiciones a y b, equivalen por regla de composición secuencial a:

- $\{I \wedge B\} A_1 \{R\}$
 - $\{R\} A_2 \{I\}$
- $$\frac{}{\{I \wedge B\} A_1, A_2 \{I\}}$$

Paso 5 - Diseñar la función cota $C \geq 0$

Esta función debe diseñarse de tal manera que se verifique:

a- Sea positiva dentro del bucle

$$I \wedge B \Rightarrow C \geq 0$$

b- La función decrezca dentro del bucle

$$\{I \wedge B \wedge C = T\} A \{C < T\}$$

Asegura que la guarda B sea falsa en algún momento, lo que garantiza la finalización del ciclo.

Para ejemplificar estos pasos veamos cómo realizar la derivación del algoritmo que calcula la suma de los elementos de un arreglo, con un recorrido de derecha a izquierda, del cual fue realizada la verificación.

Ejemplo: Derivar un programa que calcule la suma de los elementos de un arreglo que contiene N componentes enteras, recorriendo el arreglo de derecha a izquierda.

Especificación

constante entero N

entero x, $V[0..N]$

{P: $N \geq 0$ }

.....

Mientras (B)

A

finMientras

{Q: $x = \sum_{i: 0 \leq i < N: V[i]}$ }



1. Diseñar el invariante I y la condición del bucle o guarda B a partir de la postcondición, sabiendo que se debe cumplir que $I \wedge \neg B \Rightarrow Q$.

Como la postcondición no está en forma conjuntiva, optamos por introducir una nueva variable que sustituya en la postcondición a una constante. En este caso el invariante será la postcondición generalizada con esa nueva variable y la condición del bucle será la negación de la igualdad entre la variable y la constante sustituida.

En este ejemplo, el cuantificador que aparece en la postcondición involucra las constantes 0 y N. Por tanto, utilizando la técnica de reemplazo de constantes por variables y dado que el arreglo debe ser recorrido empezando por el valor N del índice, reemplazamos la constante **0 por la variable n**.

Es importante tener en cuenta que si el recorrido del arreglo fuera de izquierda a derecha, es decir la variable i comienza en cero, entonces en este caso el reemplazo sería la constante **N por la variable n**.

Entonces reemplazando la constante 0 por n se obtiene:

$$I: x = \sum_{i: n \leq i < N: V[i]}$$

Con esta definición resulta $I \wedge (n=0) \Rightarrow Q$.

Por tanto se elige como guarda $n \neq 0$. Por otra parte los valores de i están definidos en el rango $0 \leq i < N$, esto permite reforzar el invariante definiendo los valores que puede tomar la variable n a fin de evitar indefiniciones.

B: $n \neq 0$

I : $x = \sum_{i: n \leq i < N: V[i]} \wedge (0 \leq n \leq N)$

2. Inicializar las variables, es decir diseñar A_0 de modo que el invariante sea cierto; esto es $\{P\} A_0 \{I\}$

Se debe inicializar x y n , variables que aparecen en el invariante, de tal manera que la terna $\{P\} A_0 \{I\}$ sea correcta.

Para ello se debe elegir expresiones enteras E y F con las cuales inicializaremos a x y n , de modo que se garantice que

$\{P\} \Rightarrow \text{pmd } (x = F, n = E, I)$

$\text{pmd } (x = F, n = E, I) \equiv ((I_n)^E)_x^F \equiv F = \sum_{i: E \leq i < N: V[i]} \wedge (0 \leq E \leq N) \quad (1)$

Como E representa el valor del índice del arreglo y debemos recorrerlo de derecha a izquierda se inicializa en N $\Rightarrow E = N$

reemplazando en (1) $F = \sum_{i: N \leq i < N: V[i]} \wedge (0 \leq N) \quad (2)$

\equiv Por rango vacío, el elemento neutro de la suma es 0

Si $F = 0$, en (2) $\text{true} \wedge (0 \leq N) \equiv (0 \leq N) \equiv \{P\}$

Hemos probado que $\{P\} \Rightarrow \text{pmd } (x, n = F, E, I)$, y por tanto son válidas las inicializaciones $n = E = N$, y $x = F = 0$

Por tanto $n = N$, y $x = 0$ son los valores iniciales de las variables n y x

Entonces hasta el momento el algoritmo nos queda del siguiente modo:

```

{P:  $N \geq 0$ }
 $n = N$ ,
 $x = 0$ 
Mientras ( $n \neq 0$ )
  A1
  A2
finMientras
{Q:  $x = \sum_{i: 0 \leq i < N: V[i]}$ }
```

3. Determinar la función cota del tal manera que $I \wedge B \Rightarrow C \geq 0$

La cota debe definirse teniendo en cuenta las variables que intervienen en la guarda de tal manera de asegurar su decrecimiento con cada ejecución del bucle. Como n es la única variable que interviene en la guarda y su valor inicial es $n = N \geq 0$, deberá decrecer dentro del bucle

Elegimos como cota $C(n) = n \geq 0$, se debe probar que es positiva dentro del bucle

$$\begin{aligned} I \wedge B &\equiv (x = \sum_{i: n \leq i < N: V[i]} \wedge (0 \leq n \leq N)) \wedge (n \neq 0) \\ &\equiv (x = \sum_{i: n \leq i < N: V[i]} \wedge (0 < n \leq N)) \Rightarrow (0 < n \leq N) \Rightarrow (n > 0) \Rightarrow (n \geq 0) \end{aligned}$$

4. Determinar el cuerpo del ciclo

El cuerpo del ciclo consistirá de una instrucción A_1 que se encarga de mantener cierto el invariante y la instrucción A_2 que permita avanzar hacia $\sim B$ para garantizar la finalización del bucle.

Esto es, se debe encontrar A_1 y A_2 de manera que las ternas siguientes sean válidas

$$\begin{aligned} \text{a- } &\{I \wedge B\} A_1 \{R\} \\ \text{b- } &\{R\} A_2 \{I\} \\ &\overline{\{I \wedge B\} A_1, A_2 \{I\}} \end{aligned}$$

La función A_2 depende de la variable n , la cual deberá decrecer por ejemplo en 1 unidad para garantizar la finalización del bucle, entonces se elige $A_2: n = n - 1$.

Debemos encontrar además A_1 que además de mantener el invariante cumpla que:

$$\{I \wedge B\} A_1 A_2 \{I\}$$

Como A_1 es función de x , se buscará la expresión E , tal que:

$$I \wedge B \Rightarrow \text{pmd}(x = E, A_2, I), \text{ es decir } I \wedge B \Rightarrow \text{pmd}(x = E, n = n - 1, I)$$

Si el invariante es $I: x = \sum_{i: n \leq i < N: v[i]} \wedge (0 \leq n \leq N)$

$$\text{pmd} \equiv (x = \sum_{i: n \leq i < N: v[i]} \wedge (0 \leq n \leq N)) \stackrel{n-1}{n} \stackrel{E}{x}$$

$$E = \sum_{i: n-1 \leq i < N: v[i]} \wedge (0 \leq n-1 \leq N)$$

$$\equiv E = v[n-1] + \underbrace{(\sum_{i: n \leq i < N: v[i]} \wedge (0 \leq n-1 \leq N))}_x \{ \text{por partición de rango} \}$$

$$\text{pmd} \equiv x = x + v[n-1] \wedge (0 \leq n-1 \leq N) \quad (1)$$

debemos probar que $I \wedge B \Rightarrow (1)$

$$I \wedge B \equiv x = \sum_{i: n \leq i < N: v[i]} \wedge (0 \leq n \leq N) \wedge n \neq 0$$

$$\equiv (x = \sum_{i: n \leq i < N: v[i]} \wedge (0 < n \leq N)) \quad (2)$$

$$(2) \quad (3)$$

Analizamos que pasa con cada una de las variables:

$$x \Rightarrow x + V[n-1] \text{ por tanto } (2) \Rightarrow (1)$$

$$\text{Además } (0 < n \leq N) \equiv (0 \leq n-1 < N) \Rightarrow (0 \leq n-1 \leq N); (3) \Rightarrow (1)$$

Como $I \wedge B \Rightarrow (1)$ entonces $A_1: x = x + v[n-1]$ y $A_2: n = n - 1$ son asignaciones correctas

Por tanto el Algoritmo obtenido es:

```

{P: N ≥ 0}
n = N,
x = 0
Mientras (n ≠ 0)
  x = x + a[n-1]
  n = n-1
finMientras
{Q: x = ∑i=0N-1 a[i]}
```

5. Probar que el algoritmo termina, es decir la cota decrece dentro del ciclo

Para probar que la función cota decrece al ejecutar el cuerpo A del bucle, debe ser correcta la siguiente especificación: $\{I \wedge B \wedge C=T\} A \{C < T\}$

Para probar que esta terna es correcta se debe demostrar que

$\{I \wedge B \wedge C=T\} \Rightarrow \text{pmd}(A, n < T)$

Utilizando las reglas de verificación de la composición secuencial y de la asignación, se tiene:

$$\begin{aligned} \text{pmd}(A, n < T) &\equiv \text{pmd}(x = x + v[n-1]; n = n-1, n < T) \equiv ((n < T) \wedge n^{n-1}) \wedge x^{x+v[n-1]} \\ (n < T) \wedge n^{n-1} &\equiv (n-1 < T) \quad (1) \end{aligned}$$

Se debe probar que $(n=T) \Rightarrow (1)$

Si $(n=T) \Rightarrow (n-1 < T) \equiv 1$

De esta manera se demuestra la corrección total del algoritmo, es decir que además de cumplir las especificaciones, el ciclo termina.

```

constante entero N
entero x, n, a[0..N]
n = N,
x = 0
mientras (n ≠ 0)
  x = x + a[n-1]
  n = n-1
finMientras
```

Ejemplo: Dados dos números enteros no negativos a y b, derivar un programa que calcule la potencia a^b

{P: $a \geq 0 \wedge b \geq 0, a, b \in \mathbb{Z}$ }

A

{Q: $p = a^b$ }

1. Diseñar el invariante I y la condición del bucle o guarda B a partir de la postcondición, sabiendo que se debe cumplir que $I \wedge \neg B \Rightarrow Q$.

Como la postcondición no está en forma conjuntiva, optamos por introducir una nueva variable que sustituya en la postcondición a una constante. Como en la postcondición aparecen los valores a y b , debemos determinar cuál se reemplaza por la variable.

De ellas, a no modifica su valor dentro del proceso iterativo, por lo que reemplazamos b por la variable n , $b=n$ y se refuerza el invariante determinado el rango para n . Por tanto el invariante y la guarda son:

$$\mathbf{I} : (p=a^n) \wedge (0 \leq n \leq b)$$

$$\mathbf{B} : n \neq b$$

2. Inicializar las variables, es decir diseñar A_0 de modo que el invariante sea cierto; esto es $\{P\} A_0 \{I\}$

Se debe inicializar las variables p y n , que aparecen en el invariante, de tal manera que la terna $\{P\} A_0 \{I\}$ sea correcta.

Para ello se debe elegir expresiones enteras E y F tales que se garantice que

$$\{P\} \Rightarrow \text{pmd } (p = F; n = E, I) \quad (1)$$

$$\begin{aligned} \text{pmd } (p = F; n = E, I) &\equiv ((I)_n^E)_p^F \equiv ((p=a^n) \wedge (0 \leq n \leq b))_n^E)_p^F \\ &\equiv (p=a^E) \wedge (0 \leq E \leq b)_p^F \equiv (F=a^E) \wedge (0 \leq E \leq b) \end{aligned}$$

Si tomamos el valor inicial del rango de variación de E , entonces $E=0$, por tanto $F=1$ ya que $(1=a^0)$

Entonces los valores iniciales para las variables n y p podrían ser: $n=0$ y $p=1$, para demostrarlo se debe verificar (1)

$$\text{pmd} \equiv (1=a^0) \wedge (0 \leq b) \quad (2)$$

Como $\{P: a \geq 0 \wedge b \geq 0\} \Rightarrow (2)$ entonces $n=0$ y $p=1$ es una inicialización correcta

Entonces hasta ahora el algoritmo queda:

```
n=0
p=1
Mientras (n ≠ b)
```

3. Determinar la función cota, de tal manera que $I \wedge B \Rightarrow C \geq 0$

La cota debe definirse teniendo en cuenta las variables que intervienen en la guarda de tal manera de asegurar su decrecimiento con cada ejecución del bucle. Como n es la única variable que interviene en la guarda y su valor inicial es $n=0$, deberá crecer dentro del bucle, por tanto se elige como cota $C(n)=b-n$. Se debe probar que la cota es ≥ 0 al ejecutarse el bucle, es decir

$$I \wedge B \equiv (p=a^n) \wedge (0 \leq n \leq b) \wedge n \neq b \Rightarrow n < b \Rightarrow b-n > 0 \Rightarrow b-n \geq 0 \equiv C \geq 0$$

4. Determinar el cuerpo del ciclo

El cuerpo del ciclo consistirá de una instrucción avanzar A_2 que permita avanzar hacia $\sim B$ para garantizar la finalización del bucle, y la instrucción restablecer A_1 de tal manera que asegure el cumplimiento del invariante dentro del bucle.

Esto es, se debe encontrar A_1 y A_2 de manera que la siguiente terna sea válida:

$$\{I \wedge B\} A_1, A_2 \{I\}$$

La función A_2 depende de la variable n , la cual deberá crecer por ejemplo en 1 unidad.

Debemos encontrar además A_1 , es decir determinar qué valor tomará p al incrementar el valor de n , de tal forma que se verifique el invariante.

Esto es, se buscará una expresión E tal que $I \wedge B \Rightarrow \text{pmd}(p = E; n = n+1, I)$

$$\text{pmd}((p = a^n) \wedge (0 \leq n \leq b)) \stackrel{n \rightarrow n+1}{p \rightarrow p^E} \equiv ((p = a^{n+1}) \wedge (0 \leq n+1 \leq b)) \stackrel{p}{p^E}$$

$$\equiv ((E = a^{n+1}) \wedge (0 \leq n+1 \leq b)) \equiv (E = a * a^n) \wedge (0 \leq n+1 \leq b)$$

$$\text{pmd} \equiv (p = a * a^n) \wedge (0 \leq n+1 \leq b) \quad (1)$$

Debemos demostrar que $\{I \wedge B\} \Rightarrow (1)$

$$\{I \wedge B\} \equiv (p = a^n) \wedge (0 \leq n \leq b) \wedge (n \neq b) \equiv (p = a^n) \wedge (0 \leq n < b)$$

$$(2) \quad (3)$$

$$\text{Como } (p = a^n) \Rightarrow (p = a * a^n) \quad (2) \Rightarrow (1)$$

$$(0 \leq n < b) \equiv (0 < n+1 \leq b) \Rightarrow (0 \leq n+1 \leq b) \quad (3) \Rightarrow (1)$$

Por tanto $\{I \wedge B\} \Rightarrow (1)$ y son válidas las siguientes asignaciones dentro del cuerpo del ciclo: $A_2: n = n+1; A_1: p = p * a$

```

n=0
p=1
Mientras (n ≠ b)
  p=p*a
  n= n+1
finMientras

```

5. Probar que el algoritmo termina, es decir la cota decrece dentro del ciclo

Para probar que la función cota decrece al ejecutar el cuerpo A del bucle, debe ser correcta la siguiente especificación: $\{I \wedge B \wedge C = T\} A \{C < T\}$

Para probar que esta terna es correcta se debe demostrar que

$$\{I \wedge B \wedge C = T\} \Rightarrow \text{pmd}(A, b - n < T)$$

$$\text{pmd}(A, b - n < T) \equiv \text{pmd}(p = p * a; n = n+1; b - n < T) \equiv ((b - n < T) \wedge n^{n+1}) \stackrel{p \rightarrow p*a}{p} \equiv (b - (n+1)) < T \quad (1)$$

Debemos probar $\{I \wedge B \wedge C = T\} \Rightarrow (1)$

$$(0 \leq n \leq b) \wedge (n \neq b) \wedge (b - n = T) \Rightarrow (b - n = T) \equiv (b - (n+1)) < T \quad (1)$$

Entonces el algoritmo buscado es:

```

entero a, b, n
{ P: a ≥ 0 ^ b ≥ 0 ; a, b ∈ Z }
n=0
p=1
Mientras ( n ≠ b )
    p=p*a
    n= n+1;
finMientras
{ Q: p=ab }

```

Ejemplo: Búsqueda Secuencial

Especificar y derivar un algoritmo que permita obtener la posición de la primer componente con valor **k**, de un arreglo **V[0..N]** de números enteros. Si ninguna componente tiene el valor **k**, el algoritmo devuelve el índice o posición **N**.

La respuesta del algoritmo de búsqueda del valor **k** dependerá de que **k** sea o no una componente del arreglo **V**. Si **k** no es una componente, el resultado es **N** y si **k** es una componente, el resultado es el primer índice de la componente cuyo valor es **k**.

Estas dos posibilidades deben reflejarse en la postcondición, y se puede expresar el algoritmo de la siguiente forma:

```

Constante k
entero x
{P: N ≥ 0}

A0

Mientras (B)

    :::

finMientras

{Q : <(( 0 ≤ x ≤ N) ^ (∀j: 0 ≤ j < x : v[j] ≠ k)) ^ (V[x] == k) v (x == N)>

```

Esta postcondición dice que **x** es el primer valor entre 0 y **N** que cumple que todas las componentes anteriores no son la constante **k**.

1. Diseñar el invariante **I y la condición del bucle o guarda **B** a partir de la postcondición, sabiendo que se debe cumplir que $I \wedge \sim B \Rightarrow Q$.**

Una de las pautas que se proponen para construir el invariante y la condición del bucle **B**, es analizar la postcondición y si es una conjunción tomar un miembro como el invariante y el otro como la negación de **B**.

Por lo tanto escribamos la poscondición de la siguiente forma:

$$\{Q: \underbrace{(0 \leq x \leq N \wedge \forall j: 0 \leq j < x : v[j] \neq k)}_{\text{Disyunción entre los casos de éxito o fallo}} \wedge \underbrace{V[x] == k \vee (x == N)}_{\text{Disyunción entre los casos de éxito o fallo}}\}$$

Esta expresión dice: que el valor de **x** varía en el intervalo **[0..N]**; y para todo índice menor que **x** la correspondiente componente no es **k** y para el índice **x** la componente es **k** o **x** es **N**.

Como la postcondición está expresada como una conjunción de dos expresiones elegiremos a una como invariante y la otra como la negación de la condición del bucle.

$$\mathbf{I: (0 \leq x \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k)}$$

$$\mathbf{B: \sim (V[x] == k) \vee (x == N)} \equiv (V[x] \neq k) \wedge (x \neq N), \text{ luego:}$$

$$\mathbf{B: (x \neq N) \wedge (V[x] \neq k)}$$


2. Diseñar A_0 de modo que el invariante sea cierto; esto es $\{P\} A_0 \{I\}$

Como x es la única variable de programa que aparece en la condición y en el invariante, debe ser inicializada. Entonces debemos determinar A_0 , esto es buscar una expresión E tal que $x=E$; de manera que $\{P\} \Rightarrow \text{pmd}(A_0, I)$

$$\begin{aligned} \text{pmd}(A_0, I) &\equiv \text{pmd}(x=E, I) \equiv (0 \leq x \leq N \wedge (\forall j: 0 \leq j < x : v[j] \neq k)) \wedge x=E \\ &\equiv (0 \leq E \leq N) \wedge (\forall j: 0 \leq j < E : v[j] \neq k) \end{aligned}$$

Teniendo en cuenta el rango de variación de E , lo inicializamos en 0, de esta manera la segunda expresión de la conjunción es TRUE por rango vacío.

$$\text{Luego, } \text{pmd}(A_0, I) \equiv (0 \leq N) \wedge (\forall j: 0 \leq j < 0 : v[j] \neq k) \equiv 0 \leq N \equiv \{P\}$$



 Por ser rango vacío, este término es True

Como $\{P\} \Rightarrow \text{pmd}(A_0, I)$, es válida la inicialización $x=0$

3. Construir una función cota C de modo que $I \wedge B \Rightarrow C \geq 0$

Como sabemos, la función C es una cota superior que indica el número de iteraciones que quedan por realizar. En este caso, como N es cte, x que se inicializa en 0 y deberá crecer dentro del bucle, elegimos la función cota $C(x)=N-x$, para que pueda decrecer dentro del bucle.

Probemos $I \wedge B \Rightarrow C \geq 0$.

$$\begin{aligned} I \wedge B &\equiv (0 \leq x \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k) \wedge (x \neq N \wedge V[x] \neq k) \\ &\quad (1) \qquad \qquad \qquad (2) \end{aligned}$$

$$\text{De (1) y (2) } 0 \leq x < N \text{ entonces } (N > x) \equiv (N - x) > 0 \Rightarrow (N - x) \geq 0 \equiv C \geq 0$$

4. Determinar el cuerpo del ciclo.

Como el algoritmo de búsqueda consiste en comparar el elemento a buscar con los distintos elementos del arreglo avanzando en él, y la comparación está en la guarda, el cuerpo sólo consistirá de la acción que permita avanzar en el arreglo. Se debe entonces diseñar la instrucción A_2 para avanzar de modo que decrezca la función cota $(N-x)$. Elegimos entonces como instrucción A_2 de avance a $x=x+1$.

Entonces se debe probar que $\{I \wedge B\} A_2 \{I\}$ es correcto, es decir:

$$\{I \wedge B\} \Rightarrow \text{pmd}(A_2, I)$$

$$\begin{aligned} \text{pmd}(A_2, I) &\equiv \text{pmd}(x=x+1, I) \equiv (I)_{x^{x+1}} \equiv ((0 \leq x \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k))_{x^{x+1}} \\ &\equiv (0 \leq x+1 \leq N) \wedge \forall j: 0 \leq j < x+1 : v[j] \neq k \end{aligned}$$

por partición de rango

$$\equiv ((0 \leq x+1 \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k) \wedge (v[x] \neq k)) \quad (1)$$

Se debe probar que $I \wedge B \Rightarrow (1)$

$$I \wedge B \equiv ((0 \leq x \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k) \wedge ((x \neq N) \wedge V[x] \neq k)) \equiv$$

$$(2)$$

$$(3)$$

$$\text{Por (2) y (3) } I \wedge B \equiv (0 \leq x < N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k) \wedge (V[x] \neq k) \quad (4)$$

Se debe probar entonces que $(4) \Rightarrow (1)$

Comparando ambas expresiones tienen los dos últimos términos iguales, por tanto sólo se debe probar $(0 \leq x < N) \Rightarrow (0 \leq x+1 \leq N)$

$$(0 \leq x < N) \equiv (0 < x+1 \leq N) \Rightarrow (0 \leq x+1 \leq N)$$

Como $I \wedge B \Rightarrow (2)$, $A_2: x=x+1$ es una asignación válida.

5. Probar que el algoritmo termina, es decir la cota decrece dentro del ciclo

Para probar que la función cota decrece al ejecutar el cuerpo A del bucle, debe ser correcta la siguiente especificación: $\{I \wedge B \wedge C == T\} A \{C < T\}$

Para probar que esta terna es correcta se debe demostrar que

$$\{I \wedge B \wedge C == T\} \Rightarrow \text{pmd}(A, N - x < T)$$

$$\text{pmd}(A, N - x < T) \equiv \text{pmd}(x=x+1, N - x < T) \equiv (N - x < T)_x^{x+1} \equiv (N - (x+1) < T) \quad (1)$$

Se debe probar que $\{I \wedge B \wedge C == T\} \Rightarrow (1)$

$$\text{Si } N - x == T \Rightarrow N - (x+1) < T$$

Por lo tanto el algoritmo completo es:

entero x,k

{P: $N \geq 0$ }

x=0

Mientras $((x \neq N) \wedge (V[x] \neq K))$

x=x+1

finMientras

{Q: $< (0 \leq x \leq N) \wedge (\forall j: 0 \leq j < x : v[j] \neq k) \wedge V[x] == k \vee (x == N) >$ }

Ejemplo: Derivar un programa iterativo que satisfaga la siguiente especificación

entero v[N],i

{P: $N \geq 0$ }

S

{x= (#i: $0 \leq i < N : v[i] \bmod 2 == 0$)}

Es decir un algoritmo que permita contar los elementos pares de una arreglo.

1. Diseñar el invariante I y la condición del bucle o guarda B a partir de la postcondición, sabiendo que se debe cumplir que $I \wedge \sim B \Rightarrow Q$.

Como la postcondición no está en forma conjuntiva, optamos por introducir una nueva variable que sustituya en la postcondición a una constante. En este caso el invariante será la postcondición generalizada con esa nueva variable y la condición del bucle será la negación de la igualdad entre la variable y la constante sustituida.

En este ejemplo, el cuantificador que aparece en la postcondición involucra las constantes 0 y N. Por tanto, utilizando la técnica de reemplazo de constantes por variables, reemplazando la constante N por la variable n, dado que se va a recorrer el arreglo comenzando desde la posición 0, se obtiene:

$$I : x = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge (0 \leq n \leq N)$$

Con esta sustitución, el arreglo se recorrerá de izquierda a derecha y la guarda es:

$$B: n \neq N$$

2. Inicializar las variables, es decir diseñar A_0 de modo que el invariante sea cierto; esto es $\{P\} A_0 \{I\}$

Se debe inicializar las variables de programa x y n, que aparecen en el invariante, de tal manera que la terna $\{P\} A_0 \{I\}$ sea correcta.

Para ello se debe elegir expresiones enteras E y F tales que se garantice que

$$\{P\} \Rightarrow \text{pmd } (x = F, n = E, I) \equiv ((x = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge (0 \leq n \leq N)) \cdot_n^E) \cdot_x^F$$

$$\equiv F = (\#i: 0 \leq i < E : v[i] \bmod 2 == 0) \wedge (0 \leq E < N) \quad (3)$$

(1)

(2)

Teniendo en cuenta el rango de E, expresado en (2), lo inicializamos $E=0$, entonces en (1) el rango es vacío y el elemento neutro de un contador es 0.

Por tanto si $F=0$; (1) es true; entonces

$$\text{true} \wedge (0 \leq E < N) \equiv (0 \leq E < N) \equiv N \geq 0$$

$$\text{pmd} \equiv N \geq 0 \equiv \{P\}$$

Entonces los valores iniciales para las variables son $x=0, n=0$

3. Construir una función cota C de modo que $I \wedge B \Rightarrow C \geq 0$

Como N es constante y la variable n que se inicializa en 0 y deberá crecer dentro del bucle, entonces elegimos la función cota como $C(x)=N-n$, para que pueda decrecer dentro del bucle.

Probemos $I \wedge B \Rightarrow C \geq 0$.

$$I \wedge B \equiv (I : x = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge 0 \leq n \leq N) \wedge (n \neq N) \equiv n < N \Rightarrow N - n > 0 \Rightarrow C \geq 0$$

4. Determinar el cuerpo del ciclo

Como n se inicializa en cero, debe crecer dentro del bucle, por ejemplo en una unidad, esto es, $n=n+1$.

Para encontrar el cuerpo del ciclo debemos determinar el valor apropiado E , de modo tal que se mantenga el invariante, es decir se debe cumplir que:

$\{I \wedge B\} (x=E; n=n+1) \{I\}$. Para que esta terna sea correcta, debemos demostrar que

$$I \wedge B \Rightarrow \text{pmd}(x=E; n=n+1, I) \quad (1)$$

$$\text{pmd}(x=E; n=n+1, I) \equiv ((I)_n^{n+1})_x^E \equiv (x = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge 0 \leq n \leq N))_n^{n+1} x^E$$

$$\equiv (E = (\#i: 0 \leq i < n+1 : v[i] \bmod 2 == 0) \wedge 0 \leq n+1 \leq N))$$

$$\equiv \{\text{por partición de rango}\}$$

$$\equiv (E = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge (E = (\#i: n \leq i < n+1 : v[i] \bmod 2 == 0) \wedge (0 \leq n+1 \leq N)))$$

$$\equiv (E = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) \wedge (E = (i=n : v[n] \bmod 2 == 0) \wedge (0 \leq n+1 \leq N))) \quad (2)$$

Expresado en forma algorítmica a $(E = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) + \#(v[n] \bmod 2 == 0))$

$$E = (\#i: 0 \leq i < n : v[i] \bmod 2 == 0) + \#(v[n] \bmod 2 == 0)$$

$$x = x + \begin{cases} 1 & \text{si } v[n] \bmod 2 == 0 \\ 0 & \text{si } v[n] \bmod 2 \neq 0 \end{cases}$$

Con esto se concluye que hay que sumar 1 al valor de x si $v[n]$ es par, por lo que en el algoritmo debe aparecer una expresión condicional.

El cuerpo del ciclo es :

Si $(v[n] \bmod 2 == 0)$ entonces $x=x+1$ finsi

$n=n+1$

5. Probar que el algoritmo termina, es decir la cota decrece dentro del ciclo

Para probar que la función cota decrece al ejecutar el cuerpo A del bucle, debe ser correcta la siguiente especificación: $\{I \wedge B \wedge C == T\} A \{C < T\}$

Es decir que $\{I \wedge B \wedge C == T\} \Rightarrow \text{pmd}(A, N - n < T)$

$$\text{pmd}(A, N - n < T) \equiv \text{pmd}(\text{Si } (v[n] \bmod 2 == 0) \text{ entonces } x=x+1 \text{ finsi; } n=n+1, N - n < T)$$

$$\text{pmd} \equiv ((N - n < T)_n^{n+1} \equiv (N - (n+1)) < T) \quad (1)$$

Se debe probar que $(N - n == T) \Rightarrow \quad (1)$

$$(N - n == T) \Rightarrow (N - (n+1)) < T) \quad (1)$$

Por lo tanto el algoritmo completo es:

```

entero v[N], i
{P: N ≥ 0}
n=0
x=0
Mientras ( n ≠ N)
  Si (v[n] mod 2 == 0 )
    entonces x=x+1
  finsi
  n=n+1
finMientras
{Q: x = (#i: 0 ≤ i < N : v[i] mod 2 == 0)}
```

Bibliografía

- Blanco, J.; Smith, S. y Barsotti, D. (2008) Cálculo de Programas. Facultad de Matemática, Astronomía y Física. Universidad Nacional de Córdoba.
- Martí Oliet, N.; Segura Díaz, C. y Verdejo López, J. (2006) Especificación, derivación y análisis de algoritmos. Pearson Educación.SA. Madrid.
- Silva Ramírez, E. (2010) Verificación Formal de Algoritmos. Ejercicios Resueltos. Universidad de Cádiz. Servicio de Publicaciones.
- M.^a Teresa González de Lena Alonso, Isidoro Hernán Losada y otros (2005) Introducción a la programación: problemas resueltos en Pascal. Editorial Centro de Estudios Ramón Areces SA. España. Recuperado de:
https://books.google.com.ar/books?id=25jqDQAAQBAJ&pg=PA517&lpg=PA517&dq=un+aserto+es+un+predicado&source=bl&ots=f1oeCmkhAh&sig=8RT4t_rYMBbrZ6hFgDCImVRKPPc&hl=es-419&sa=X&ved=0ahUKEwja3qaf07vVAhVEf5AKHQuQBkMQ6AEIKzAB#v=onepage&q=un%20aserto%20es%20un%20predicado&f=false

Práctico 1

Verificación y Derivación

Ejercicio 1

Dado los siguientes fragmentos de programa, encontrar la precondition más débil que satisfaga la especificación dada, suponiendo que x, y, z son variables enteras. Indique en cada caso una precondition que haga válido el algoritmo y otra para que el algoritmo no sea válido.

- a- $\{P_{md}\} (x,y)=(x+1,y-1) \{ x+y>0 \}$
- b- $\{P_{md}\} (x,y,z)=(x+1,y-1,x+y) \{ x+y=z \}$
- c- $\{P_{md}\} x = x*x, x = x+1 \{ x>0 \}$
- d- $\{P_{md}\} x = x+y, y = 2*y-x \{ y>0 \}$

Ejercicio 2

Verificar si el siguiente algoritmo que suma los elementos de un arreglo recorriéndolo de izquierda a derecha, cumple esta especificación utilizando el invariante:

$I = \{0 \leq n \leq N \wedge x = \sum_{i: 0 \leq i < n} a[i]\}$

Algoritmo

const int N

int a[0..N), n, x

$\{P: N \geq 0\}$

n=0

x=0

Mientras (n < N)

$x = x + a[n]$

$n = n+1$

Finmientras

$\{Q: x = \sum_{i: 0 \leq i < N} a[i]\}$

Ejercicio 3

El siguiente algoritmo calcula el mínimo recorriendo un arreglo de derecha a izquierda, verificar si cumple la especificación utilizando como invariante:

$I = \{0 \leq n \leq N \wedge x = \min_{i: n \leq i < N} a[i]\}$

Algoritmo

const int N

int a[0..N), n, x

$\{P: N > 0\}$

n=N - 1

x= a[N-1]

Mientras (n ≠ 0)

$x = x \min a[n-1]$

$n = n-1$

finmientras

$\{Q: x = \min_{i: 0 \leq i < N} a[i]\}$

Ejercicio 4

Derivar un algoritmo que permita sumar los elementos de un arreglo, recorriéndolo de izquierda a derecha.

Especificación

```
const int N
int a[0..N),m
{P: N>0}
S
{Q: m=<=∑i: 0 ≤ i <N : a[i] >}
```

Ejercicios Propuestos**Ejercicio 1**

Derivar un algoritmo que calcule el producto de los elementos de un vector, empezando de izquierda a derecha.

Especificación

```
const int N
int a[0..N),m
{P: N>0}
S
{Q: m=<=∏ i: 0 ≤ i <N : a[i] >}
```

Ejercicio 2

Derivar un programa que indique si todos los valores de un arreglo de N componentes enteras son positivos.

La idea es recorrer el arreglo mientras los elementos sean positivos y cortar el ciclo cuando se encuentra un valor ≤ 0 . El programa devuelve un valor que será N si todas las componentes son positivas (indica que recorrió el arreglo hasta el final), en caso contrario el valor de la posición donde encontró un número ≤ 0 ($<N$)

Especificación

```
const int N
int x
int a[0..N)

{P: N ≥ 0}
S
{Q: ( 0 ≤ x ≤ N) : ∀ i: 0 ≤ i < x: a[i]>0) ∧ (( x=N) ∨ (a[x] ≤ 0)) }
```

Esta expresión indica que el valor de x varía en el intervalo $[0..N)$; que todo índice menor que x tiene componente positiva, y que el ciclo finaliza cuando $x=N$ o si encuentra un elemento no positivo.

Unidad 1: Lenguajes Procedurales

Introducción

Para 1969 ya se habían contabilizado más de 120 lenguajes de programación, hoy en día hay quienes afirman que la lista supera los 2000. Wikipedia nombra y describe alrededor de 671⁴.

Si bien un programador profesional usa efectivamente no más de tres lenguajes, en función de las posibilidades y las tendencias de la empresa o medio en que está inserto, es importante que se explore más profundamente en el diseño de los lenguajes y en su efecto sobre la implementación, por muchas razones, entre ellas:

- Mejorar la habilidad en el desarrollo de algoritmos eficaces.
- Mejorar el uso de lenguaje de programación que el profesional utiliza.
- Hacer posible una mejor elección del lenguaje de programación, cuando del profesional dependa.
- Facilitar el estudio de nuevos lenguajes
- Permitir el diseño de nuevos lenguajes.

Concepto de Computación

Desde un punto de vista amplio, que incluya la definición, diseño e implementación de lenguajes de programación, el **concepto de computación** se puede interpretar como "*cualquier proceso que puede ser realizado por una computadora*". No solo cálculos matemáticos sino también manipulación de datos, procesamiento de textos, almacenamiento y recuperación de la información.

Lenguaje de Programación

Para que exista comunicación, debe existir una comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje. En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas.

Estos programas se ejecutarán sobre una computadora que realizará las tareas descritas.

La utilización de un lenguaje de programación requiere, por tanto, una comprensión mutua por parte de personas y máquinas.

*Un lenguaje de programación es un sistema notacional para describir computaciones en una forma legible tanto para la máquina como para el ser humano.*⁵

Algunas cuestiones de Diseño

En el diseño de lenguajes, la meta es lograr la potencia, expresividad y comprensión que requiere la legibilidad del ser humano, mientras se conservan la precisión y simplicidad necesarias para la traducción de máquina.

La *legibilidad* por parte del ser humano es un requisito complejo y sutil. Depende en gran parte de las posibilidades de abstracción que tiene un lenguaje de programación.

La *abstracción* permite concentrarse en un problema al mismo nivel de generalización, dejando de lado los detalles irrelevantes. La abstracción permite trabajar con conceptos y términos familiares al entorno del problema, sin tener que transformarlos a una estructura no familiar⁶.

Abstracciones procedimentales

Es una secuencia nombrada de *instrucciones* que tienen una función específica y limitada. (Ejemplo de esto es el uso de funciones en lenguaje C).

Abstracciones de datos

⁴ Enlace Web https://es.wikipedia.org/wiki/Anexo:Lenguajes_de_programación - 01/08/2017.

⁵ Louden, Kenneth C. (2004) Lenguajes de Programación Principios y Práctica. Editorial Thomson. México, D.F. Pág 3.

⁶ Ibidem. Pág 26.

Técnica de inventar *nuevos tipos de datos* que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. (Ejemplo: uso de struct en lenguaje C).

Abstracciones de control

Resume propiedades de la transferencia de control, es decir, la modificación de la trayectoria de ejecución de un programa en una determinada situación. (Ejemplo: for, if, while, etc son ejemplo de abstracciones de control en lenguaje C).

El principal objetivo de la abstracción en el diseño de lenguajes de programación *es el control de la complejidad*. Se controla la complejidad elaborando abstracciones que esconden los detalles cuando es apropiado. En general todos los mecanismos de abstracción se diseñan para que los seres humanos puedan entenderlos.

Lenguajes Imperativos

Un lenguaje procedural o imperativo es un lenguaje controlado por mandatos o instrucciones.

El proceso de ejecución de programas procedurales por la computadora, tiene lugar a través de una serie de estados, cada uno definido por el contenido de la memoria, los registros internos y los almacenes externos durante la ejecución.

El almacenamiento inicial de estas áreas define el **estado inicial** de la computadora. Cada paso en la ejecución transforma este estado en otro nuevo, a través de la modificación de alguna de estas áreas. Esta transformación se denomina **transición de estado**. Cuando termina la ejecución del programa, el **estado final** viene dado por el contenido final de las áreas antes mencionadas. Por lo tanto, la ejecución de un programa puede verse como una serie de transiciones de estados que realiza la computadora.

En este modelo de computación, un programa es un conjunto de enunciados y la ejecución de cada enunciado hace que cambie el valor de una o más localidad de almacenamiento en memoria, es decir que la memoria pase a un nuevo estado.

Si se considera a la memoria como un conjunto de cajas, la construcción de un programa consiste en construir los estados de máquina sucesivos (cánicas en cajas o valores en variables) que se necesitan para llegar a la solución.

Este tipo de modelo de computación, tiene características propias del modelo de Von Newmann, variables que representan valores de memoria y asignación que permite que el programa opere sobre dichos valores.

Existen otros paradigmas que no son tan dependientes del modelo de computadora de Von Newmann, y serán estudiados en cursos superiores.

El lenguaje C, lenguaje que se usará en este curso responde al paradigma imperativo.

Definición de un lenguaje de programación

Un lenguaje natural es un instrumento de comunicación utilizado de forma común entre personas. Para que exista comunicación, debe existir una comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas. Estos programas se ejecutarán por una computadora que realizará las tareas descritas. El programa debe ser comprendido tanto por personas como por la computadora. La utilización de un lenguaje de programación requiere, por tanto, una comprensión mutua por parte de personas y máquinas. Este objetivo es difícil de alcanzar debido a la naturaleza diferente de ambos.

En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia. Sin embargo, los lenguajes de programación se definen habitualmente por una autoridad, que puede ser el diseñador individual del lenguaje o un determinado comité.



Para que el computador pueda comprender un lenguaje humano, es necesario diseñar métodos que traduzcan tanto la estructura de las frases como su significado a código máquina. Los diseñadores de lenguajes de programación construyen lenguajes que saben cómo traducir o que creen que serán capaces de traducir. Si las computadoras fuesen la única audiencia de los programas, estos se escribirían directamente en código máquina o en lenguajes mucho más mecánicos.

Sin embargo, el programador debe ser capaz de leer y comprender el programa que está construyendo y las personas no son capaces de procesar información con el mismo nivel de detalle que las máquinas.

Los lenguajes de programación son, por tanto, una solución de compromiso entre las necesidades del emisor (programador - persona) y del receptor (computadora - máquina).

Las declaraciones, tipos, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito en un programa. Por otro lado, la utilización de un vocabulario limitado y de unas reglas estrictas son concesiones para facilitar el proceso de traducción.

La definición de un lenguaje de programación necesita una descripción precisa y completa, además de un traductor que permita aceptar el programa en el lenguaje en cuestión y que, lo ejecute directamente o bien lo transforme en una forma adecuada para su ejecución.

Dos partes se pueden distinguir en la definición de un lenguaje. La estructura o **sintaxis** y la **semántica** o significado.

La **sintaxis** estudia las reglas de formación de frases. Las reglas de sintaxis nos dicen cómo se escriben los enunciados, declaraciones y otras construcciones del lenguaje.

La **semántica**, sin embargo, hace referencia al significado de esas construcciones.

Traductores y computadoras simuladas por software⁷

La programación se hace más a menudo en un lenguaje de alto nivel “muy alejado” del lenguaje de máquina, (lenguaje muy cercano al del hardware).

Para que un lenguaje de programación sea útil debe tener un traductor, es decir un programa que acepte el programa que escribe el programador - en lenguaje de alto nivel - y que lo ejecute directamente o lo transforme en una forma adecuada para su ejecución.

Existen por tanto dos soluciones básicas para estas cuestiones: traducción e interpretación. Un traductor que produce otro programa equivalente pero adecuado para su ejecución se llama **compilador**, y un traductor que ejecuta el programa directamente se llama **intérprete**.

Simulación de software (interpretación)

En lugar de traducir los programas de alto nivel, a programas equivalentes en lenguaje máquina, se podrían simular a través de programas ejecutados en otra computadora anfitrión, una computadora cuyo lenguaje máquina sea de alto nivel. Esto se construye con software que se ejecuta en la computadora anfitrión, la computadora con lenguaje de alto nivel que de otra manera se podría haber construido en hardware. A esto se le conoce como una simulación por software (o interpretación) de la computadora con lenguaje de alto nivel en la computadora anfitrión.

Un **intérprete** es un software que recibe un programa en lenguaje de alto nivel, lo analiza y lo ejecuta. Para analizar el programa completo, va traduciendo sentencias de código y ejecutándolas si están bien, así hasta completar el programa origen.

⁷ Cueva Lovelle, Juan (1998) Conceptos Básicos de Procesadores de Lenguajes. Ed. SERVITEC. España. Capítulo 2. Pág. 8.

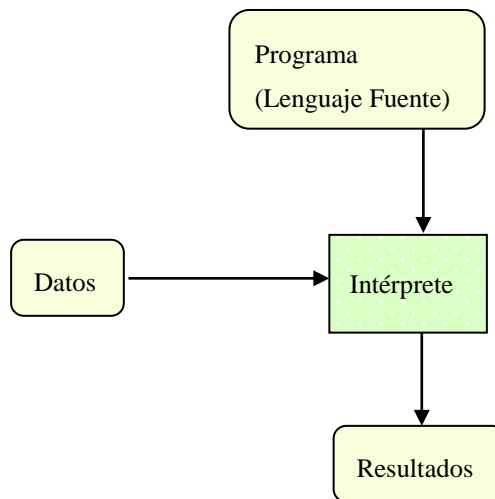


Fig. 3: Esquema general de un intérprete

Traductores

Por **Traductor** se denota a cualquier “*procesador de lenguajes*”⁸ que acepta programas en cierto lenguaje fuente (que puede ser de alto o bajo nivel) como entrada y produce programas funcionalmente equivalentes en otro lenguaje objeto.

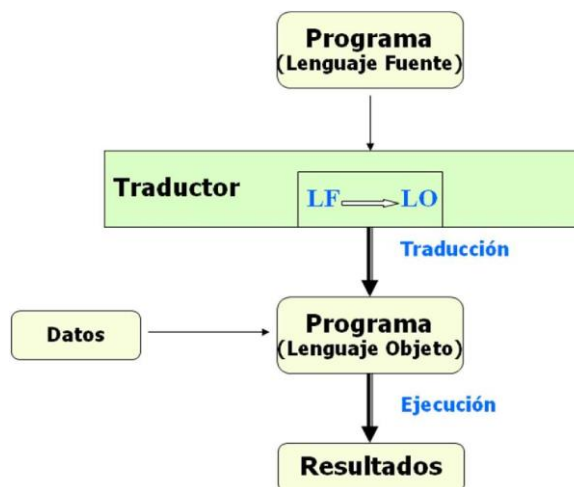


Fig. 1: Esquema general de un traductor⁹

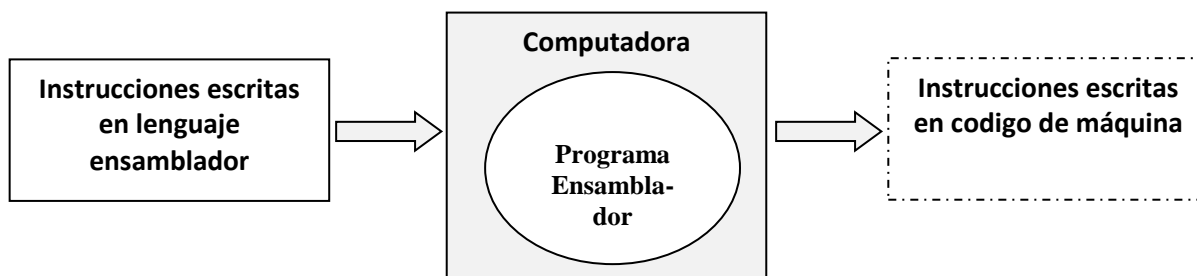
⁸ Procesador de lenguaje es un nombre genérico que se aplica a traductores, compiladores, intérpretes, y otros programas que realizan operaciones con los lenguajes.

⁹ Pratt Terrence y Marvin Zelkowitz (1987) Lenguajes de Programación Diseño e Implementación. Editorial Prentice Hall. Hispano americana, S.A. 2° Edición. Capítulo 2. Pág. 41.

Distintos tipos de traductores

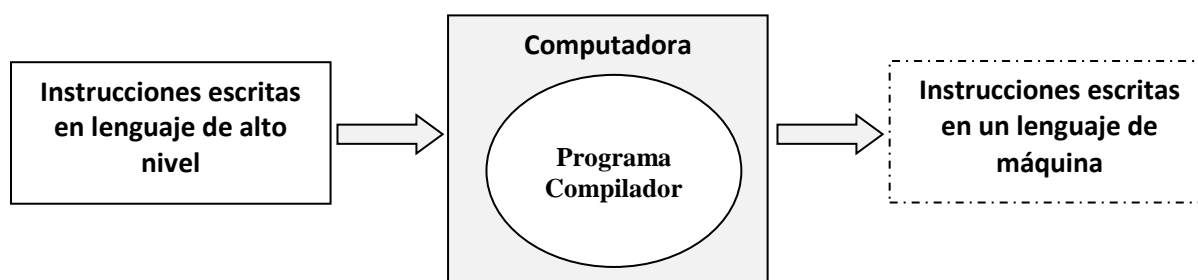
Ensamblador

El término ensamblador (del inglés assembler) se refiere a un tipo de programa que se encarga de traducir un archivo fuente escrito en un lenguaje ensamblador, a un fichero objeto que contiene código máquina, ejecutable directamente por el microprocesador.



Compilador

Para traducir las instrucciones de un programa escrito en un lenguaje de alto nivel a instrucciones de un lenguaje máquina, hay que utilizar un programa llamado **compilador**. Así pues, el compilador es un programa que recibe como datos de entrada el código fuente de un programa escrito por un programador, y genera como salida un conjunto de instrucciones escritas en el lenguaje binario de la computadora donde se van a ejecutar.



Fases de la Compilación

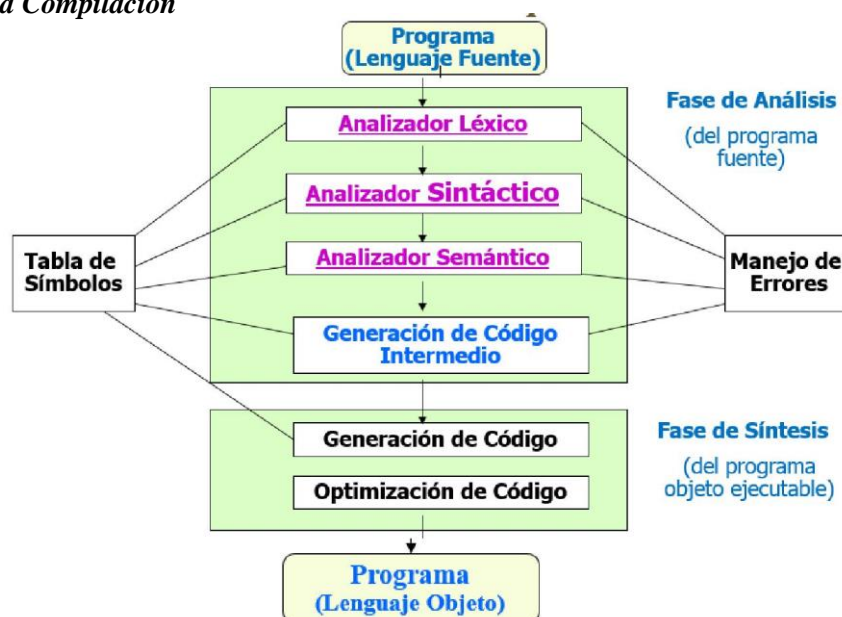


Fig. 2: Esquema general de un compilador¹⁰

¹⁰ Ibidem. Pratt. - Capítulo 3. Pág. 72.

Analizador léxico (o revisor) es un programa que lee la secuencia de caracteres del programa fuente, de izquierda a derecha y agrupa esas secuencias de caracteres en unidades con significado propio (componentes léxicos o “tokens” en inglés). Es decir que identifica el tipo de cada elemento léxico: identificador, número, operador, delimitador y adjunta una marca de tipo, para enviárselo al analizador sintáctico.

Analizador sintáctico o parsing identifica estructuras más grandes del programa como enunciados, declaraciones, llamadas a funciones, expresiones, etc., a partir de los elementos léxicos o tokens producidos por el analizador léxico. Analiza las relaciones estructurales entre los componentes léxicos, esto es semejante a realizar el análisis gramatical sobre una frase en lenguaje natural

Analizador semántico: programa que detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. Las tareas básicas a realizar son la verificación de tipos en asignaciones y expresiones, la declaración del tipo de variables y funciones antes de su uso, el correcto uso de operadores, el ámbito de las variables y la correcta llamada a funciones.

Se procesan las estructuras sintácticas reconocidas por el analizador sintáctico y comienza a tomar forma la estructura del código ejecutable

En este curso se observará al análisis semántico estático (en tiempo de compilación), utilizando la Tabla de Símbolos.

Traducción e Interpretación

Los traductores y los interpretes proporcionan ventajas diferentes, por ejemplo una ventaja de los traductores es que los códigos referentes a ciclos que se repiten gran cantidad de veces y son ejecutados muchas veces se traducen sólo una vez a diferencia de los interpretes que tienen que decodificarlo la misma cantidad de veces que se ejecute.

Una ventaja para la interpretación es que en caso de un error proporcionan más información acerca de donde fue la falla que en un código objeto traducido.

Por lo tanto, si el programa de entrada está en un lenguaje de alto nivel, el traductor procesará los enunciados del programa en el orden físico de entrada, una vez cada uno. El intérprete podrá procesar los enunciados siguiendo el orden lógico de control y algunos enunciados podrá procesarlos más de una vez y otros omitirlos (según que sean parte de un bucle o que el control no los alcance).

No siempre se dan por separado, la traducción y la interpretación. Generalmente se mezclan estos dos mecanismos; según cuál de ellos prevalezca se habla de **lenguajes compilados** cuando prevalece la traducción (ejemplo: C, C++, Pascal, Fortran, Ada, Cobol) o **interpretados** cuando predomina la simulación (ejemplo: LISP, PROLOG, SmallTalk, ML).

En general cuando hay enunciados repetitivos es mejor la traducción pues se decodifica una vez, aunque se ejecuten muchas veces. La desventaja es la pérdida de información acerca del programa.

Actividad 1: Investigue la bibliografía e indique claramente las ventajas y desventajas de la traducción y la simulación.

Implementación de lenguajes

Cuando se implementa un lenguaje de programación, las estructuras de datos y algoritmos que se utilizan en tiempo de ejecución de un programa escrito en ese lenguaje, definen un **modelo de ejecución** definido por la implementación del lenguaje.

Por ejemplo, para la implementación de las operaciones de suma de enteros y raíz cuadrada, el implementador puede optar por representar la suma de enteros a partir de la suma de enteros proporcionada directamente por el hardware, y para la operación de raíz cuadrada puede optar por una simulación por software.

De ahí que es importante entender que la implementación de un lenguaje está determinada por múltiples decisiones que debe tomar el implementador en función de los recursos de hardware y software disponibles en la computadora subyacente y los costos de su uso.

Criterios de diseño de lenguajes de programación

Al diseñar lenguajes de programación, a menudo es necesario tomar decisiones sobre las características que se incluyen de forma permanente, las características que no se incluyen (aunque existen mecanismos que facilitan su inclusión) y las que no se permiten. Estas decisiones pueden afectar al diseño final del lenguaje y posiblemente entrar en conflicto con otros aspectos del lenguaje.

A continuación se presenta una síntesis de criterios de diseño, extraídos de autores como Pratt y Loudon, Kenneth.

Existen dos principios importantes a tener en cuenta: la *eficiencia* y la *regularidad*.

Eficiencia

La **eficiencia** debe evaluarse en distintas dimensiones (código, traducción, etc.):

- **Eficiencia de código:** Hace referencia a un diseño del lenguaje que permite que el traductor genere un código ejecutable eficiente. Por ejemplo el uso de declaraciones anticipadas de las variables.
- **Eficiencia de traducción:** Hace referencia a un diseño del lenguaje que permite que el código fuente se traduzca con rapidez y con un traductor de tamaño razonable. Ejemplo: C standard permite un compilador de una sola pasada pues las variables se declaran antes de usarse. C++, el compilador debe realizar una segunda pasada para resolver referencias del identificador.
- **Eficiencia de implementación:** Hace referencia a la eficiencia con la que se puede escribir un traductor. Esto está relacionado con la eficiencia de la traducción y con la complejidad de la definición del lenguaje.
- **Eficiencia de la programación:** Hace referencia a la facilidad para escribir programas en el lenguaje. Entre otras, esto involucra la capacidad de expresión del lenguaje y la capacidad de darle mantenimiento al programa.

Regularidad

La **regularidad** es una cualidad que implica que hayan pocas restricciones en el uso de sus constructores particulares, pocas interacciones raras entre dichos constructores, y menos sorpresas en la forma en la que se comportan las características del lenguaje.

Distintos principios básicos miden la regularidad, entre otros la *generalidad*, la *ortogonalidad* y la *uniformidad*.

- **Generalidad:** Un lenguaje alcanza generalidad evitando casos especiales en cuanto a disponibilidad o uso de constructores.

La no generalidad está ligada a restricciones de los constructores con independencia del contexto en que estén.

En lenguaje C el operador de igualdad `==` carece de generalidad. Dos arreglos no pueden compararse a través de este operador, por ejemplo.

En cuanto a la longitud de los arreglos, lenguaje C admite generalidad pues maneja arreglos de longitud variable, mientras que esta generalidad no está presente en lenguaje Pascal, que admite sólo arreglos de longitud fija.

- **Ortogonalidad:** Se refiere al atributo de combinar varias características de un lenguaje de manera que la combinación tenga significado.
Ejemplo: Si un lenguaje provee de expresiones que pueden devolver un valor y también posee un enunciado condicional que compara valores; estas dos características son ortogonales si se puede usar dentro del enunciado condicional cualquier expresión.
- **Uniformidad:** Hace referencia a la consistencia de la apariencia y comportamiento de los constructores del lenguaje.

Las no-uniformidades son de dos tipos:

- a) dos cosas similares que no parecen serlo
- b) cosas no similares que parecen serlo o se comportan similarmente cuando no debieran.

Ejemplo de falta de uniformidad en lenguaje C los operadores & (and bit a bit) y && (and lógico) tienen una apariencia confusamente similar y producen resultados muy distintos.

Modelos de computación¹¹

Entendemos por una computación a cualquier proceso que puede ser realizado por una computadora, con esto nos estamos refiriendo no solo a cálculos matemáticos sino que incluimos la manipulación de datos, el procesamiento de textos y el almacenamiento y recuperación de la información.

Estos modelos de computación dan lugar a distintas familias de lenguajes o paradigmas. En informática, es posible observar varias comunidades, cada una hablando su propio lenguaje y utilizando sus propios paradigmas. De hecho los lenguajes de programación suelen fomentar el uso de ciertos paradigmas y disuadir el uso de otros.

Paradigma de Programación¹²

Un paradigma de programación "*consiste en un método para llevar a cabo cómputos y la forma en la que deben estructurarse y organizarse las tareas que debe realizar un programa*".

Se trata de una propuesta tecnológica adoptada por una comunidad de programadores, y desarrolladores cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados; la resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software. Representa un enfoque particular o filosofía para diseñar soluciones.

Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo.

Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Es un estilo de programación empleado.

Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que lo sustituyen parcial o totalmente.

Clasificación de los Paradigmas

En general, la mayoría de paradigmas son variantes de los dos tipos principales de programación:

- Imperativa
- Declarativa.

En la **programación imperativa** se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

En la **programación declarativa** las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.



¹¹ Ibidem. Louden. Capítulo 1. "Paradigmas de computación". Pág. 12 a 18.

¹² https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n#Paradigma_de_programaci%C3%B3n

Variantes de los Paradigmas de Programación

Programación imperativa o procedural

Es el más usado en general, se basa en dar instrucciones al ordenador de cómo hacer las cosas en forma de algoritmos, en lugar de describir el problema o la solución. Las recetas de cocina y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; donde cada paso es una instrucción. Es la forma de programación más usada y la más antigua, el ejemplo principal es el lenguaje de máquina. Ejemplos de lenguajes puros de este paradigma serían el C, BASIC o Pascal.



- **Programación orientada a objetos:** está basada en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones. Está representado por C++, C#, Java o Python entre otros, pero el más representativo sería el Smalltalk que está completamente orientado a objetos.
- **Programación dinámica:** está definida como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, busca resolver problemas en $O(n)$ sin usar por tanto métodos recursivos. Este paradigma está más basado en el modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo.
- **Programación orientada a eventos:** la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

Programación declarativa

está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan solo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando). No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial. Los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas. Unos de los primeros lenguajes funcionales fueron Lisp y Prolog.

- **Programación funcional:** basada en la definición los predicados y es de corte más matemático, está representado por Scheme (una variante de Lisp) o Haskell. Python también representa este paradigma.
- **Programación lógica:** basado en la definición de relaciones lógicas, está representado por Prolog.
- **Programación con restricciones:** similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.
- **Programación multiparadigma:** es el uso de dos o más paradigmas dentro de un programa. El lenguaje Lisp se considera multiparadigma. Al igual que Python, que es orientado a objetos, reflexivo, imperativo y funcional.⁴ Según lo describe Bjarne Stroustrup, esos lenguajes permiten crear programas usando más de un estilo de programación. El objetivo en el diseño de estos lenguajes es permitir a los programadores utilizar el mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la forma más fácil y eficiente posible. Por ejemplo, lenguajes de programación como C++, Genie, Delphi, Visual Basic, PHP o D5 combinan el paradigma imperativo con la orientación a objetos. Incluso existen lenguajes multiparadigma que permiten la mezcla de forma natural, como en el caso de Oz, que tiene subconjuntos (particularidad de los lenguajes lógicos), y otras características propias de lenguajes de programación funcional y de orientación a objetos. Otro ejemplo son los lenguajes como Scheme de paradigma funcional o Prolog (paradigma lógico), que cuentan con estructuras repetitivas, propias del paradigma imperativo.

- **Programación reactiva:** Este paradigma se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos que se "suscriben" a los primeros (es decir, quedan a la escucha de la emisión de eventos de estos) y *reaccionan* a los valores que reciben. Es muy común usar la librería Rx de Microsoft (Acrónimo de Reactive Extensions), disponible para múltiples lenguajes de programación.
- **Lenguaje específico del dominio o DSL:** se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. El más representativo sería SQL para el manejo de las bases de datos, de tipo declarativo, pero los hay imperativos, como el Logo.

Lenguaje C

Para hablar del lenguaje C, es bueno recordar que los primeros lenguajes de programación se denominaban lenguajes de bajo nivel o lenguajes de máquina (ceros y unos). El uso de éstos implicaba un conocimiento exhaustivo del computador y un manejo preciso de las direcciones de memoria asociadas a las distintas operaciones y operadores. Para resolver los problemas que surgían de esta tediosa forma de programar, surgieron los lenguajes de programación ensambladores que usaban símbolos para expresar las distintas operaciones y las direcciones de memoria donde se almacenaban los valores de las variables.

Finalmente, surgen los lenguajes de alto nivel, que distan mucho del lenguaje de máquina y se asemejan más al lenguaje que utilizan las personas para comunicarse (el inglés).

Debido al notable abaratamiento de las computadoras alrededor de los años 70, fue evidente el progreso de la tecnología de compiladores, dando lugar al surgimiento de muchos lenguajes líderes sobre distintos dominios: comercial, científico, militar etc. Si bien muchos de ellos ya prácticamente no se usan, y muchos más se han desarrollado, en la actualidad el lenguaje C tiene la característica de estar aún vigente como consecuencia de las continuas revisiones que en él se han realizado y de su capacidad de reflejar los cambios que se produjeron en otras áreas de la computación.

El lenguaje C fue desarrollado en 1972 por Dennis Ritchie y Ken Thompson en los laboratorios Bell Telephone de AT&T. El lenguaje C comienza a desarrollarse como un lenguaje para escribir software y compiladores, específicamente para desarrollar el sistema operativo UNIX, pero su uso se ha generalizado y hoy en día muchos sistemas están escritos en C o en C++. Además, es un lenguaje que no está ligado a ningún sistema operativo ni máquina, lo que permite la portabilidad entre distintas plataformas. El lenguaje C es un lenguaje de propósitos generales y, si bien es un lenguaje de alto nivel, provee sentencias de bajo nivel que permite facilidades similares a las que se encuentran en los lenguajes ensambladores.

Etapas en la obtención de programa ejecutable en lenguaje C

Antes de ejecutar un programa en lenguaje C, existen ciertas etapas que se deben realizar, ya que todas ellas contribuyen a la obtención del programa ejecutable. Ellas son: edición, compilación, enlace, carga y ejecución.

Edición

Esta etapa consiste en la escritura de un programa en C, utilizando un programa editor. Este programa, que se conoce como **programa en código fuente**, deberá ser almacenado en el disco con un nombre que lo identifique para su posterior compilación. Los programas fuentes se almacenan o graban con extensión **.c** o **.cpp** según se utilice lenguaje C ó C++ (C plus plus).

Este es el **Preprocesador**, un editor de texto, que toma como entrada una forma ampliada de un lenguaje fuente y su salida es una forma estándar del mismo lenguaje fuente.



Compilación

El proceso de compilación consiste en la traducción del programa fuente a código binario, para que la computadora pueda interpretar las órdenes. Esto se logra dando la orden *compilar* del ambiente de desarrollo integrado (IDE) o bien desde la línea de órdenes.

Durante esta etapa, se detectan los posibles **errores sintácticos** cometidos por el programador, cuando no ha respetado las reglas de sintaxis del lenguaje. En este caso, es necesario volver a editar el programa, corregir los errores señalados y compilar nuevamente. Este proceso se repite hasta que no se detectan errores, obteniéndose el **programa en código objeto** que es almacenado en disco.

Se debe tener en cuenta que antes de la fase de traducción mencionada, el compilador invoca automáticamente a un programa *preprocesador* que obedece directrices especiales que indican ciertas operaciones que deben realizarse antes de la compilación. En general, esas operaciones consisten en la inclusión de otros archivos en el programa a compilar o en el reemplazo de símbolos especiales por textos de programa.

Enlace

El programa objeto no es ejecutable, esto se debe a que generalmente los programas contienen referencias a funciones definidas en otro lugar, una biblioteca estándar o en bibliotecas definidos por algún programador, por ejemplo. El enlazador es el que se encarga de vincular el código objeto con las bibliotecas, obteniendo así el **programa ejecutable**, que es almacenado en disco.

Carga

El **Cargador** es un procesador de lenguajes cuya entrada es un lenguaje objeto, un programa en lenguaje de máquina en manera reubicable; las modificaciones que realiza son a tablas de datos que especifican las direcciones de memoria donde el programa necesita estar para ser ejecutable.

El cargador toma el programa ejecutable del disco y los transfiere a memoria.

Finalmente la computadora, bajo el control de la UCP (Unidad Central de Procesamiento) toma cada una de las instrucciones y las ejecuta.

El siguiente es un esquema de las etapas descriptas:

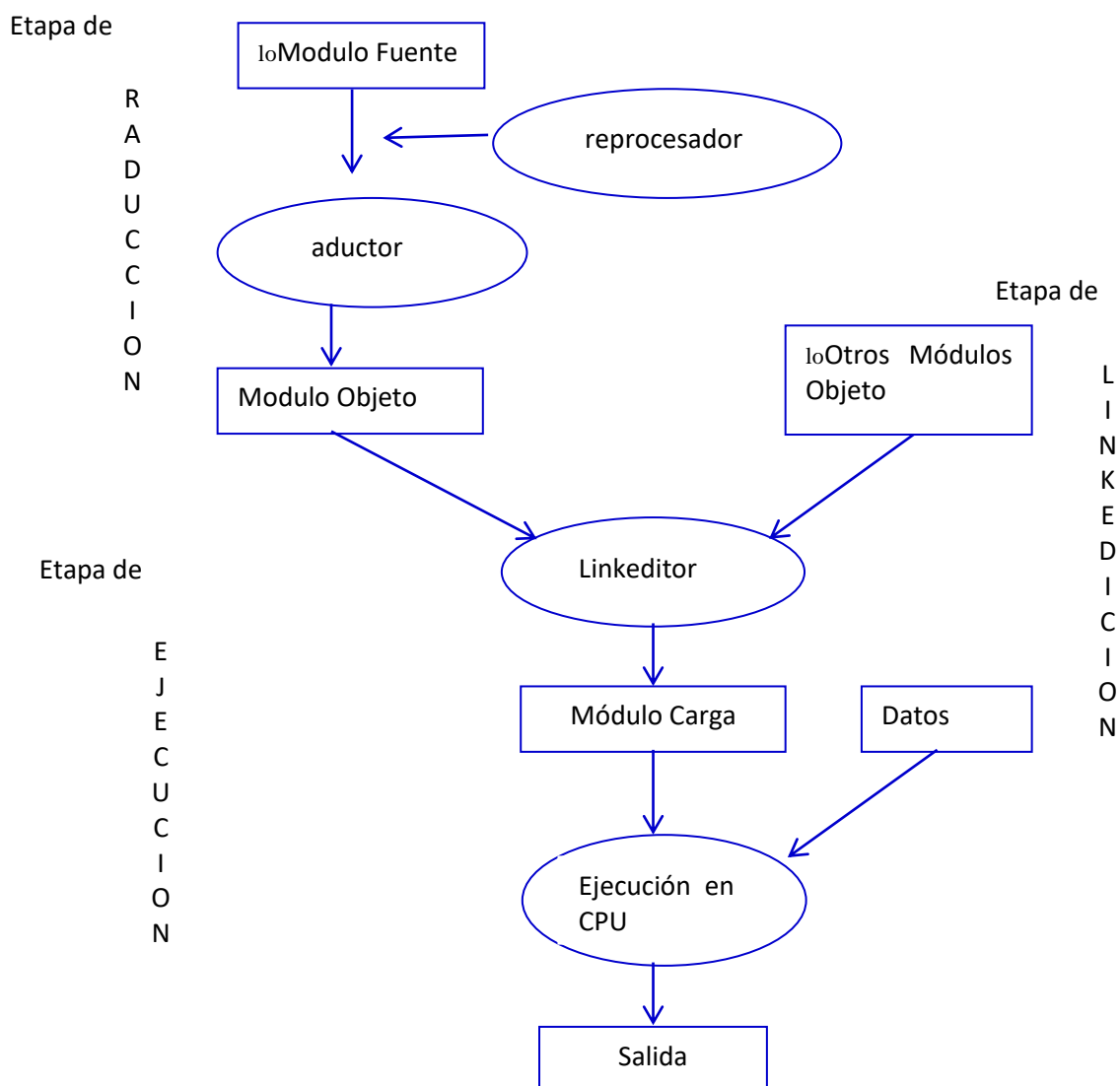


Fig. 4: Etapas para la obtención de un programa ejecutable en C

Bibliografía

- Cueva Lovelle, Juan (1998) Conceptos Básicos de Procesadores de Lenguajes. . Ed. SERVITEC. España.
- Fontela, Carlos (2003) Programación Orientada a Objetos. Técnicas Avanzadas de Programación. Editorial Nueva Librería. Buenos Aires.
- Louden, Kenneth C. (2004) Lenguajes de Programación Principios y Práctica. Editorial Thomson. México, D.F.
- Pratt Terrence y Marvin Zelkowitz (1987) Lenguajes de Programación Diseño e Implementación. Editorial Prentice Hall. Hispano americana, S.A. 2º Edición. México.

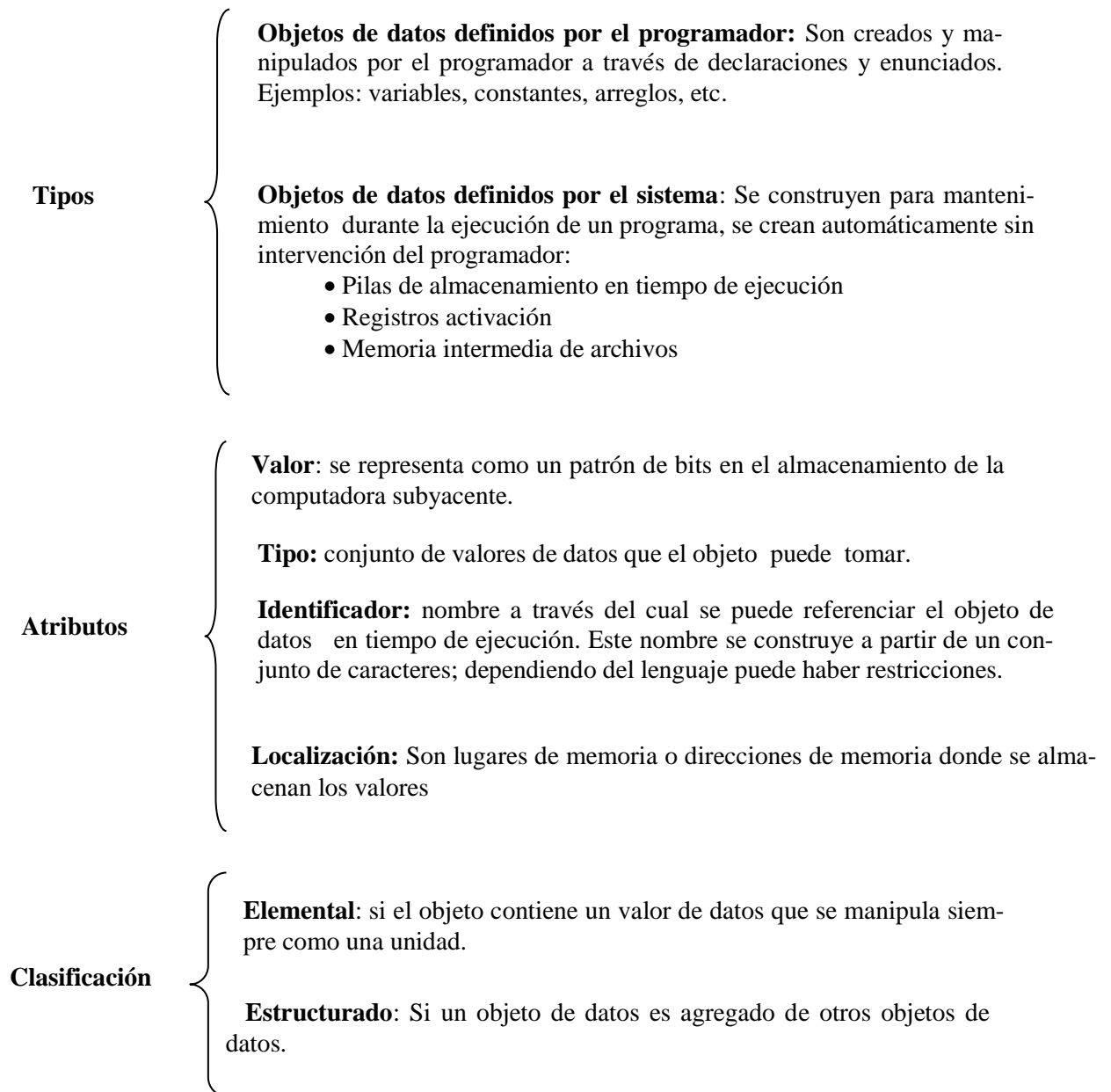
Unidad 2: Objeto de Datos

Introducción

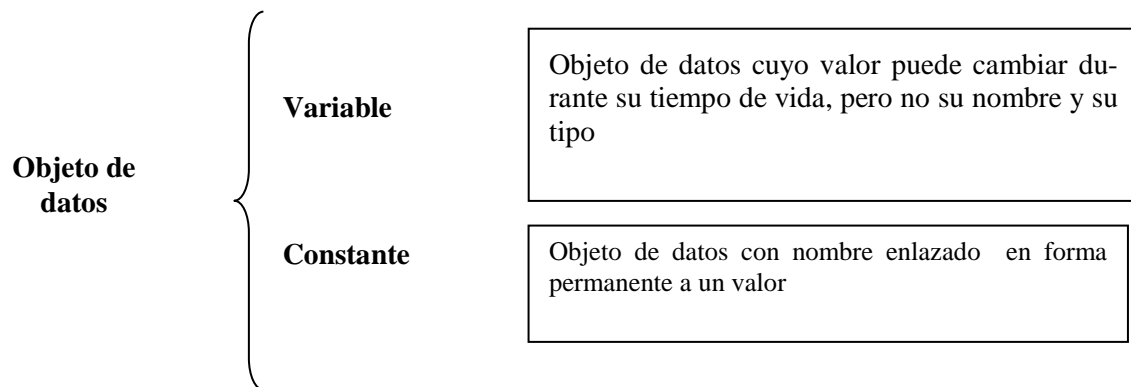
Si tenemos en cuenta que un programa se puede considerar como un conjunto de operaciones que se van a realizar sobre ciertos datos en un orden determinado, los puntos sobresalientes a tener en cuenta en el estudio de los lenguajes de programación son: datos, operaciones y control.

Objeto de Datos

Un **objeto de datos** es un recipiente que se usa para guardar y recuperar valores de datos



Variables y Constantes



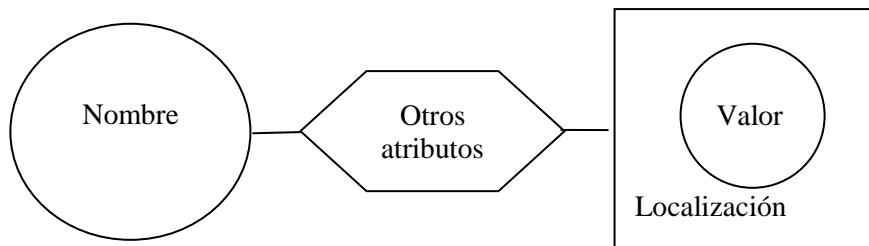
Almacenamiento de variables y constantes en memoria

Antes de usar una variable hay que declararla, al hacerlo se debe dar su nombre (identificador) y su tipo. Para el identificador se sugiere un nombre significativo, de modo de lograr una mejor legibilidad del programa.

Variable

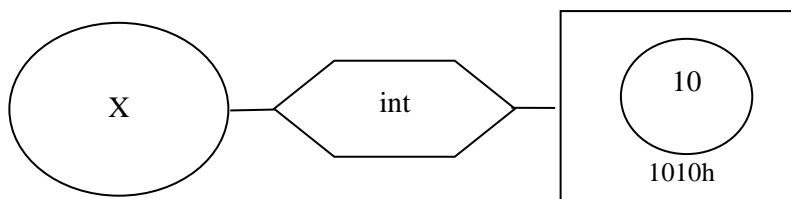
Una variable se puede especificar a través de sus atributos, que incluyen nombre, localización, valor y otros atributos como tipo de datos y tamaño.

Una representación esquemática con **diagramas sintácticos** (estos son una alternativa gráfica para la Forma de Backus-Naur (BNF)) es:



Ejemplo

int X=10;

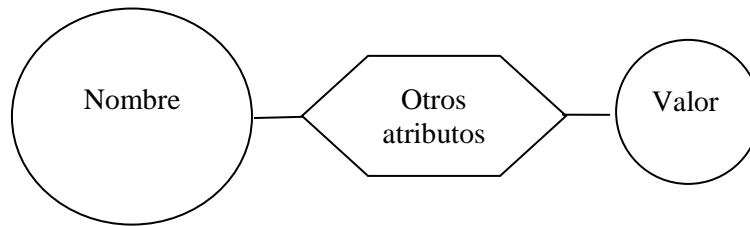


Si hacemos referencia a los atributos principales de una variable, nombre, localización y valor, podemos representarla de manera más sencilla como sigue:



Constante

Una constante es una entidad del lenguaje que tiene un valor fijo mientras existe dentro de un programa.



Una constante es similar a una variable, excepto que no tiene un atributo de localización. Esto no significa que una constante no se almacene en la memoria en algunas ocasiones.

En C las constantes se declaran con la directiva `#define`, esto significa que esa constante tendrá el mismo valor a lo largo de todo el programa.

El identificador de una constante así definida será una cadena de caracteres que deberá cumplir los mismos requisitos que el de una variable (sin espacios en blanco, no empezar por un dígito numérico, etc).

Ejemplo

```
#include <stdio.h>
#define PI 3.1415926
```

```
int main()
{
    printf("Pi vale %f", PI);
    return 0;
}
```

Lo cual mostrará por pantalla:

```
Pi vale 3.1415926
```

Hay lenguajes como C++ en los que el valor de las constantes se computa en tiempo de carga o al comienzo de la ejecución, por lo tanto su valor necesita ser almacenado en la memoria.

Ejemplo de constantes

```
#include<stdio.h>
main()
{ const int a=5;
  const int b=2 +a;

  printf("%d", b);
  getchar();
}
```

La diferencia entre el uso de **const** y el uso de **#define** está en que mediante **const** se declara una constante que tiene un tratamiento similar a una variable (por ejemplo, la constante es de un tipo de dato) mientras que mediante **#define** se indica que escribir el nombre especificado equivale a escribir el valor, con una correspondencia directa y sin tratamiento análogo al de una variable.

Ejemplo

```
const int JUGADORES = 5;      ...      #define JUGADORES 5
```

En la primera declaración se indica que JUGADORES es una constante de tipo int mientras que en la segunda se indica que donde aparezca en el código la palabra JUGADORES deberá ser reemplazada por 5 directamente. En general, usar **#define** supone que la compilación sea más rápida al no tener el compilador que realizar el tratamiento y verificaciones propias de variables. Por ello su uso resultará recomendable cuando existan ciertos valores numéricos que tengan un significado especial, valor constante y uso frecuente dentro del código. Las constantes definidas con **#define** se denominan constantes simbólicas.

Tiempo de Vida¹³

Durante la ejecución de un programa algunos objetos de datos existen desde el comienzo de la ejecución, otros pueden crearse dinámicamente durante la misma. Algunos objetos persisten durante toda la ejecución del programa, otros se destruyen durante la ejecución del mismo.

Por lo tanto, definimos como tiempo de vida de un objeto, al tiempo durante el cual el objeto puede usarse para guardar valores de datos.

Enlace (ligadura) y Tiempo de enlace

El **enlace o** ligadura de un elemento de programa a una característica o propiedad particular, hace referencia a la elección de una propiedad dentro de un conjunto de propiedades posibles.

El momento del programa durante el cual se realiza este enlace se denomina **tiempo de enlace o tiempo de ligadura**. Existen una gran variedad de tipos de enlace, algunos de ellos se presentan a continuación, a través de ejemplos en lenguaje C.

Ejemplo: Sea el enunciado $X=X+10$; podemos distinguir distintos tipos de enlaces:

1. La variable X puede tener asociados distintos tipos de datos, el conjunto de los posibles tipos de datos para X se fijan en **tiempo de definición del lenguaje**.

Por ejemplo, hay lenguajes como Pascal que admiten un tipo de dato boolean. Mientras que lenguaje C no lo admite, si lo admite C++. Esto se determina cuando se define el lenguaje.

2. Por otro lado, si el tipo de dato asociado a X fuese un tipo definido por el programador (tal cual lo admite Pascal o C), los posibles valores de X se fijan en **tiempo de traducción**.
3. La decisión de asignarle como nombre X a la variable, así como el tipo de dato de la misma, son enlaces elegidos por el programador y corresponden al tiempo de **traducción o compilación**.

Ejemplo: `int X;`

Hay lenguajes en los que el programador no especifica un tipo de datos para las variables, sino que las mismas van tomando el tipo de datos en función del valor que se le va asociando. En estos casos una misma variable puede ir tomando distintos tipos de datos en tiempo de ejecución. Ejemplo de esto lo brinda SmallTalk y Prolog.

4. El conjunto de los posibles valores que puede tomar la variable X se determina en **tiempo de implementación del lenguaje**.
5. La asignación $X=X + 10$, cambia el valor asociado a la variable X en **tiempo de ejecución**, asignándole como nuevo valor el valor que tenía más 10.
6. Además, el número o constante **10** se representa como una serie de bits en tiempo de implementación, mientras que su representación decimal en el programa se hace en **tiempo de definición del lenguaje**.
7. La elección del + para representar la adición se hace en tiempo de definición del lenguaje, mientras que la determinación de cuál es la operación que se debe realizar se hace en **tiempo de compilación** (recordemos que el + es un operador homónimo u homonímico, es decir el mismo operador puede usarse para distintos tipos de sumas, cada uno tiene una definición según el contexto).
8. La definición en detalle de la función + se realiza en tiempo de implementación del lenguaje y ésta depende de la definición de adición del hardware subyacente.

¹³ Capítulo 5. Pág. 114 a 117. Kenneth Loudon.

Cuando los enlaces de un lenguaje se realizan antes de la ejecución se dicen que son enlaces tempranos o **ligaduras estáticas**¹⁴, mientras que cuando se realizan en tiempo de ejecución hablamos de enlaces tardíos o **ligaduras dinámicas**. Las ventajas y desventajas de unos y otros giran en torno al conflicto entre eficiencia y flexibilidad.

De todos los tipos de enlaces expuestos, salvo los enlaces en tiempo de ejecución, todos los otros enlaces son **ligaduras estáticas**.

En lenguajes como C o Pascal, donde la eficiencia de ejecución es primordial, es común que la mayoría de los enlaces sean tempranos.

En lenguaje LISP, donde la flexibilidad es su objetivo, los enlaces se retardan hasta la ejecución para que los mismos puedan hacerse dependientes de los datos.

Es importante hacer notar, que el traductor debe conservar los enlaces de tal manera que se realice la operación apropiada durante la traducción y la ejecución. Esto es, el traductor debe conservar las ligaduras de modo que se den significados apropiados a los nombres en tiempo de traducción y ejecución.

Para esto, el traductor crea una estructura de datos, llamada **Tabla de Símbolos**. A grandes rasgos, se puede decir que esta estructura contiene una entrada por cada identificador diferente declarado en el programa fuente, además para cada identificador se introduce información adicional (tipo de variable, parámetro formal, nombre del subprograma, entorno de referencia, etc.).

Conforme se avanza en la traducción la Tabla de Símbolos va cambiando de modo que quedan siempre reflejadas la eliminación o inserción de enlaces.

En temas siguientes, una vez desarrollado el concepto de función retomaremos la tabla de símbolos, y mostraremos ejemplos que permitan ver el funcionamiento de la misma.

Tipos de datos

Tipo de Datos: En un programa, los objetos de datos como un flotante F, un arreglo A, una struct T, tienen un tipo de datos que está asociado al conjunto de valores que puede tomar dicho objeto. No obstante, para un lenguaje de programación podemos dar una definición más precisa:

Un tipo de datos es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

De ahí, podemos hacer referencia a un tipo de datos como la clase de los arreglos, la clase de los flotantes, etc.

Los tipos de datos pueden ser: **Primitivos:** datos que están integrados al lenguaje o **Definidos por el programador:** el lenguaje provee recursos para definir nuevos tipos.

Especificación e implementación de un tipo de datos

Elementos básicos de una especificación de un tipo de datos

Atributos: distinguen a los objetos de datos de ese tipo. Por ejemplo, para un tipo de dato arreglo los atributos son: nombre, dimensiones, tipo de dato de las componentes, etc.

Valor: especifica los valores de datos que pueden tomar los objetos de este tipo de dato.

Operaciones: hacen referencias a las distintas formas de manipular los objetos de datos de ese tipo de datos.

¹⁴Algunos autores hablan de enlaces, otros de ligaduras.

Elementos básicos de la **implementación** de un tipo de datos:

Representación de almacenamiento: hace referencia a la forma que se usa para representar los objetos de datos de un tipo de dato determinado de datos en memoria.

Algoritmos y procedimientos: definen las operaciones del tipo, en términos de manipulaciones de su representación de almacenamiento.

Tipos de datos elementales

Especificación

Tipo elemental de datos, hace referencia a una clase de objetos de datos que contienen **un solo valor**.

- Los atributos de un objeto de un tipo elemental de datos, como por ejemplo su tipo, se mantienen invariables durante su tiempo de vida.
- El conjunto de valores que puede tomar un objeto de un tipo de dato elemental está generalmente ordenado, hay un máximo, un mínimo y dados dos valores distintos del dato, uno es mayor que el otro.
- Las operaciones asociadas pueden ser primitivas o definidas por el programador. Las operaciones primitivas son operaciones que se especifican como parte de la definición del lenguaje, mientras que las definidas por el programador son subprogramas o procedimientos que el programador define sobre objetos de un tipo elemental de datos.

Representación de almacenamiento

Ordinariamente, representación de almacenamiento hace referencia al tamaño de bloque de memoria que se requiere, a la disposición de los valores y atributos dentro de ese bloque y no a su ubicación dentro de la memoria.

El almacenamiento está influido por la computadora subyacente que va ejecutar el programa. Por ejemplo, la representación de almacenamiento para valores enteros o reales generalmente utiliza la representación binaria que se usa en el hardware para esos valores. Por lo tanto, las operaciones básicas sobre datos de este tipo se pueden implementar a través de las operaciones de hardware. Caso contrario, las operaciones deben simularse por software, de manera mucho menos eficiente.

En lenguajes como **C**, Pascal, Fortran los **atributos** no forman parte del objeto de datos en tiempo de ejecución, es decir no se guardan en la representación de almacenamiento. Son determinados por el compilador. Esto permite mayor eficiencia en el uso del almacenamiento y mayor velocidad de ejecución, objetivos primordiales de estos lenguajes.

En lenguajes como Lisp y Prolog, los atributos forman parte del objeto de datos en tiempo de ejecución. Estos atributos se guardan como un descriptor.

Implementación de tipos de datos elementales

La implementación de un tipo elemental de datos hace referencia a la representación de almacenamiento para objetos de datos y valores de ese tipo, y a un conjunto de operaciones y procedimientos que definen las operaciones del tipo en términos de manipulaciones de la representación de almacenamiento.

Implementación de las operaciones

Implementación de algoritmos y procedimientos que definen las operaciones	$\left\{ \begin{array}{l} \text{Como operación de hardware} \\ \text{Simulada por software a través de subprogramas} \\ \text{Simulada por software como una secuencia de código de línea} \end{array} \right.$
--	---

Ejemplos

- Como operación de hardware: Por ejemplo, si los números enteros usan la representación de hardware para enteros, entonces la suma puede implementarse usando la operación de suma integrada en el hardware.
- Simulada por software a través de subprogramas: Una operación de potenciación generalmente no es suministrada directamente como una operación de hardware. En este caso se puede implementar un subprograma que calcule esta operación.

Si los objetos de datos no se representan usando la representación de hardware, entonces las operaciones se deben simular comúnmente, a través de subprogramas que se suministran a través de bibliotecas.

Ejemplo

```
#include <stdio.h>
#include <math.h>
main()
{ int base, exponente, r;
  scanf("%d %d", &base, &exponente);
  r=pow(base, exponente);
  printf("\n Potencia: %d", r);
}
```

En este ejemplo en Lenguaje C, la función potencia (pow) está simulada por software y suministrada por el lenguaje en el archivo <math.h>.

-Lenguaje C también permite que ciertas operaciones puedan simularse por macros. Esto puede ser beneficioso cuando el subprograma es muy corto, directamente se reemplaza por él, cada vez que este aparece en el programa.

```
#include <stdio.h>
#include <conio.h>
#define maximo(A,B) A<B?B:A
main()
{
  int x=20, y=30,z;
  z= maximo(x,y);
  printf("\n el mayor es %d",z);
  getch();
}
```

Declaraciones

Una declaración es un enunciado escrito por el programador que sirve para comunicar al traductor información primordial para establecer ligaduras.

En general, podemos decir que las declaraciones contribuyen a:

- Ofrecer al traductor información que permite el almacenamiento para un objeto de datos, reduciendo el tiempo de ejecución del programa que se está traduciendo.
- Cuando hay operadores homónimos, las declaraciones permiten que el traductor, en tiempo de compilación, determinar cuál es la operación particular a la que se hace referencia el operador homónimo.

Nota: Los operadores +, *, -, /, etc. son operadores homónimos pues denotan operaciones genéricas que tienen distintas definiciones en función de los argumentos de las mismas.

Ejemplo

```
void main (void)
{ int A, B, C;
  float P, Q, R;
  A = B + C;
  P = Q + R;
  .....
}
```

En este ejemplo, el operador + se ha utilizado para realizar una suma de enteros y una de reales, operaciones que matemáticamente se realizan de manera distinta.

Las declaraciones de las variables permiten al traductor del lenguaje determinar en tiempo de compilación cual es la operación particular que designa el símbolo homónimo +, esto es, o suma de enteros o suma de flotantes.

Las declaraciones informan sobre el tiempo de vida de un objeto lo que hace más eficiente la gestión de almacenamiento durante la ejecución de un programa.

Por ejemplo, en **lenguaje C**, a las variables locales a un subprograma se les asigna espacio en un bloque de memoria en forma automática al entrar a la ejecución del subprograma. Al terminar ese bloque también en forma automática se libera. Sin embargo, para variables dinámicas (creadas con malloc), como no se declaran explícitamente en el subprograma, se les debe asignar almacenamiento en forma individual en un área de almacenamiento por separado, a un costo mayor porque el pedido lo hace el programador.

```
double calculo (double a )
{ return a * 1.20; }

void main (void)
{
  double sueldo,s;
  .....
  s=calculo(sueldo);
  int *p;
  p=malloc(30 * sizeof(int));
  .....
  free(p);
  .....
}
```

La declaración de la función **calculo** indica una operación que puede ser usada en el programa.

La variable **a** de tipo double, indica que para su almacenamiento se usará 8 bytes. Además por ser una variable local a la función se le asigna memoria automáticamente durante la ejecución del subprograma y automáticamente se libera al terminar la ejecución del mismo.

Las variables **sueldo** y **s**, son locales al main y por ser del tipo double necesitan para su almacenamiento 8 bytes.

La variable **p** es un puntero a entero, por lo tanto necesita para su almacenamiento 4 bytes.

Con la función malloc se habilitará el uso de un área de almacenamiento de $30 \times 4 = 120$ bytes.

La función free liberará el espacio generado por malloc.

El **propósito más importante** de las declaraciones es permitir la verificación estática de tipos.

Verificación de tipos

La verificación de tipos es el proceso que realiza el intérprete o el compilador para verificar que todas las construcciones en un programa tengan sentido en términos de los tipos declarados.

Ejemplo

Como la verificación de tipos implica, entre otras, comprobar que cada operación que un programa ejecuta recibe el número de argumentos apropiados y del tipo de datos correcto, el caso de la expresión $A = B + C$ si C es tipo `char`, las variables A y B son tipo `float`, entonces el compilador detectará un error de tipo de argumento en la operación.

La verificación de tipos se puede realizar en tiempo de ejecución (**verificación dinámica de tipos**) o en tiempo de compilación (**verificación estática de tipos**).

Verificación dinámica de tipos: Si la información de tipos se conserva y se verifica en tiempo de ejecución, la verificación es dinámica. Por definición los intérpretes llevan a cabo verificación dinámica. No obstante, existen algunos compiladores que proveen mecanismos para que la verificación de tipos se realice en tiempo de ejecución; tal es el caso del lenguaje LISP.

Para implementar este tipo de verificación se guarda una marca de tipo en cada objeto, el cual indica el tipo de datos asociado al mismo. Por lo tanto, cuando se realiza la verificación de tipos, lo primero que se efectúa es la verificación de marcas de tipo de argumentos. La operación se realiza si los tipos de argumentos son correctos, de lo contrario se indica un error. También, para cada operación, se deben anexar las marcas de tipo apropiadas a sus resultados, para que las operaciones subsiguientes se puedan verificar.

Este tipo de verificación es propia de lenguajes como LISP y PROLOG, ya que en ellos no se dan declaraciones de variables, y por lo tanto para dos variables A , y B , su tipo de datos puede cambiar durante la ejecución del programa.

Verificación estática de tipos: La verificación estática de tipos se realiza durante la traducción de un programa. En general, la información necesaria para esta verificación, es proporcionada por el programador a través de las declaraciones.

¿Cómo se realiza la verificación de tipos?

- El traductor recoge información de las declaraciones desde la tabla de símbolos, que contiene información de variables y operaciones.
- Después de reunir la información de tipos, se verifica cada operación que invoca el programa (verifica tipo de argumento válido). Si los argumentos son válidos, se determina el tipo de datos del resultado y esta información se guarda para verificar operaciones posteriores. Si la operación es homonimia, el nombre de la operación se puede reemplazar por el nombre de la operación específica de tipo particular que usa esos argumentos.
- Como la verificación estática incluye todas las operaciones que aparecen en cualquier subprograma, se verifican todas las rutas de ejecución. No se necesita una revisión adicional en busca de errores. Se evitan por lo tanto, marcas de tipo en los objetos en tiempo de ejecución, como en la verificación dinámica, se gana almacenamiento y tiempo de ejecución

Uno de los problemas más significativos respecto a los tipos de datos, es que no hay consenso entre los diseñadores de lenguajes respecto de hasta qué punto la información de tipos debe explicitarse y utilizarse para la verificación de tipos, antes de la ejecución de un programa.

Hay quienes ponen más énfasis, en una mayor flexibilidad en el uso de tipos, y que no procuran una verificación de tipos estricta, y hay quienes procuran la implementación de la verificación estricta de tipos en tiempo de traducción.

Un lenguaje **fuertemente tipificado** es por tanto, un lenguaje con una verificación de tipos estricta. O lo que es lo mismo, **un lenguaje de tipo fuertes** detecta en forma estática, los *errores de tipo de un programa*.

Verificación de tipos y lenguaje C

Los compiladores de C aplican una verificación estática de tipos durante la traducción, no obstante muchas inconsistencias en los tipos no causan errores de compilación, sino que son automáticamente eliminados, presentando o no un mensaje de advertencia.

La mayoría de los compiladores modernos, tienen configuraciones de nivel de error que aportan un tipificado más fuerte. C++ agrega una verificación de tipos más fuerte a C, pero bajo la forma de advertencias en vez de errores, con fines de compatibilidad con C. Este tipo de mensaje no impide la ejecución. Ignorar estas advertencias puede ser un desatino peligroso (Stroustrup, 1994, Página 42).

Conversión de tipos

Cuando en la verificación de tipos hay una discordancia entre el tipo real de un argumento y el tipo esperado, el lenguaje puede:

1. marcar el error y ejecutar una acción de error apropiada
2. se puede realizar una conversión de tipos.

Una conversión de tipos es una operación, de la forma:

Conversión: tipo1 \longrightarrow tipo2

Esta conversión toma un objeto de un tipo y produce el objeto de datos correspondiente de un tipo distinto.

Conversión implícita o coerciones¹⁵: son invocadas automáticamente en ciertos casos de discordancia. El principio básico que gobierna las coerciones es no perder información.

Ejemplo

```
void main(void)
{ char c;
  int i;
  float f;
  double d, result;
  result = (c / i) + (f * d) - (f + i)
  ....
}
```

Si durante la verificación de tipos de una operación como la planteada en el ejemplo anterior, ocurre una discordancia entre los tipos de los operandos y/o del tipo de resultado esperado, el compilador busca la operación de conversión para que de **manera automática** se realicen los cambios al tipo de dato ade-

¹⁵ K. Loudon, en el libro Lenguajes de Programación las denomina coerciones. Pág. 211

cuado. En este caso, estamos frente a una **conversión implícita** o **coerción**. En C, las coerciones son reglas.

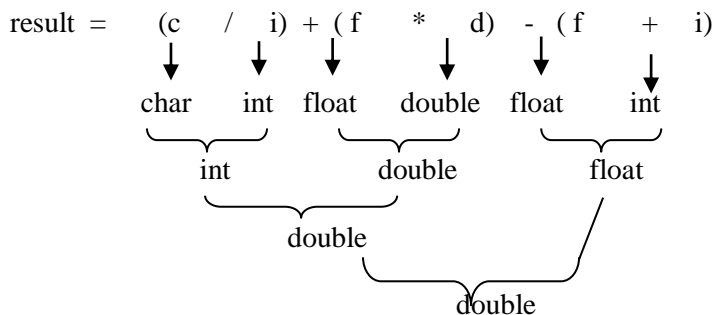
Por ejemplo, en expresiones aritméticas el lenguaje C permite utilizar operandos de distintos tipos en una expresión, teniendo en cuenta las siguientes reglas:

a- Si los dos operandos son de tipo coma flotante con distinta precisión, el operando de menor precisión se transforma a la precisión del otro operando y el resultado se expresa en esta precisión (la más alta). Por ejemplo entre un float y un double, el float se convierte a double y el resultado será double.

b- Si un operando es de tipo coma flotante y el otro es un char o un int, el char/int se convertirá al tipo coma flotante y el resultado se expresará en este tipo. Por ejemplo una operación entre un int y un double tendrá como resultado un double.

c- Si ninguno de los operandos es de tipo coma flotante, el compilador convierte los char y short int a int antes de evaluar, salvo que en la expresión aparezca un long int. En este caso los operandos y el resultado se convertirán a long int.

Para el ejemplo anterior, la conversión implícita es la siguiente:



En todos los casos la conversión se ha realizado de tal forma que el tipo de datos final puede contener toda la información que se convierte sin perder información. En este caso la conversión es de **ensanchamiento o extensión**.

No obstante, si la variable **result** fuese de tipo **int**, entonces habría una conversión de un tipo real a un entero, y en tal caso puede ser que se pierda información. Mientras el valor real 15.0 es igual al entero 15, no sucede lo mismo con el real 15.20, el cual no es representable como entero. Este real será convertido al entero 15, perdiendo información. Este tipo de coerción se llama de **estrechamiento o restricción**.

Conversión explícita o forzada: conjunto de funciones integradas que el programador puede llamar para realizar una conversión de tipos.

En C, se puede forzar a una expresión a ser de un tipo específico, usando el operador unario **cast**.

Formato: **(tipo) expresión** donde **tipo**, es el tipo al que se desea forzar el resultado

Ejemplo

```
#include <stdio.h>

void main(void)
{
    int x=7;
    printf("\n %f", x/2);
}
```

Para este programa la salida es 3.0, sin embargo si forzamos a x a un tipo flotante, tal cual lo expresa el siguiente programa:

```
void main(void)
{ int x=7;
  printf("\n %f", (float) x/2);
}
```

La salida es 3.50.

En estos casos de conversión explícita, para la verificación estática de tipos, se inserta código adicional en el programa compilado para invocar la operación de conversión en el punto apropiado durante la ejecución.

El lenguaje Pascal casi no provee coerciones, salvo excepciones, todas las discordancias son tratadas como errores.

En C, las coerciones son la regla. Cuando se encuentra una discordancia de tipo, el compilador busca una operación de conversión apropiada para insertarla en el código compilado y de ese modo proveer el cambio de tipo adecuado. Sólo si no se encuentra una conversión posible, la discordancia se presenta como un error.

En otros capítulos, se presentarán nuevos ejemplos donde revisaremos los distintos tipos de coerciones.

Clasificación

Los tipos de datos elementales (también llamados primitivos) son los tipos de datos originales de un lenguaje de programación, esto es, aquellos que proporciona el lenguaje y con los que se puede construir estructuras de datos y tipos de datos abstractos. Estos se pueden clasificar de la siguiente manera:

- Carácter
- Entero
- Real
- Booleano o Lógico
- Puntero o Apuntador

Algunos lenguajes no proporcionan los tipos de datos booleanos y puntero. A continuación se describirá cada uno de ellos.



Tipos de datos enteros

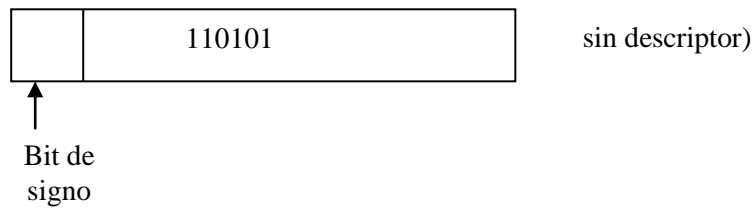
Especificación: Un objeto de datos de tipo entero tiene como atributo solo su tipo. Los valores que puede tomar un objeto de datos es un subconjunto ordenado de números enteros, que tiene límites inferior y superior. El máximo valor es elegido por el implementador de manera que refleje el valor entero máximo representable en el hardware subyacente.

Por lo tanto, los valores posibles varían en un intervalo como:

[-maximo-entero , maximo-entero]

Implementación: La implementación de un tipo de dato entero usa la representación de almacenamiento definida por el hardware y las operaciones primitivas relacionadas a enteros.

En lenguaje C (también el lenguaje Pascal) las representaciones de almacenamiento carecen de descriptor puesto que ambos lenguajes ofrecen declaraciones y verificación estática de tipos.



Otros lenguajes incluyen descriptores en tiempo de ejecución, tal es el caso de SmallTalk .

Operaciones sobre enteros: Las operaciones típicas que un lenguaje de programación define sobre objetos de datos enteros son las siguientes: Operaciones aritméticas, Operaciones relacionales, Operación de asignación.

Además, en lenguaje C, los enteros representan valores booleanos, lo cual permite operaciones de bit, tales como & (and), | (or) y <<(desplazamiento) entre otras. Estas operaciones no serán trabajadas en este curso.

Operación de asignación

Dentro de las operaciones antes destacadas, resulta importante entender **la asignación**. La asignación es una operación de casi todos los tipos de datos elementales, y es la operación básica para cambiar el valor de una variable. Ejemplos de asignación son:

X=2+8; para X entera

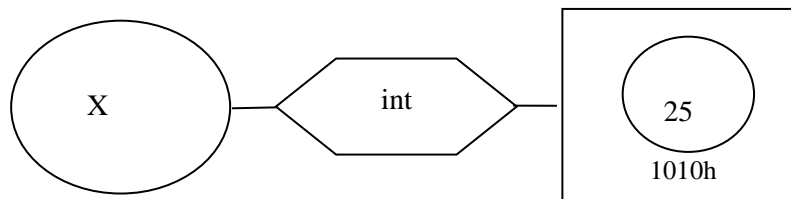
X=Y; para X e Y enteras.

Valores lvalue y rvalue

Para comprender de manera más clara como funciona esta operación, se analiza el concepto de **valor l** y **valor r** de una variable.

Dado que una variable tiene una localización y un valor almacenado en dicha localización, para distinguir claramente a ambos, se denomina **valor r** de una variable al valor que contiene dicha variable, y **valor l** se refiere a la localidad de la variable.

Supongamos que la variable X tiene almacenado un valor 25, su representación esquemática es:



Por lo tanto **el valor l (X) es 1010h**, mientras que **el valor r(X) es 25**.

Luego de la operación X= 35, cambia el valor r (X) por 35.

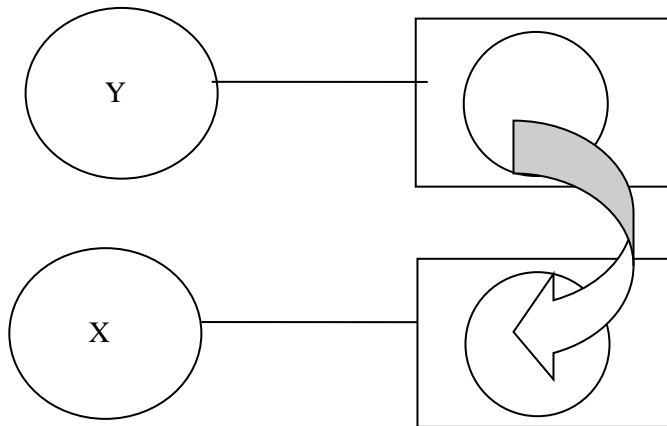
El **l** representa la izquierda (left), aludiendo a que un **valor l** o **lvalue** debe estar a la izquierda de una operación de asignación. Del mismo modo la **r** representa la derecha (right), aludiendo a que un **valor r** o **rvalue** debe estar a la derecha de una operación de asignación.

Como puede inferirse una constante no tiene **lvalue**.

La semántica de una operación de asignación puede cambiar en distintos lenguajes. Se analiza a continuación la operación de asignación en lenguaje Pascal y en lenguaje C.

Esquemáticamente:

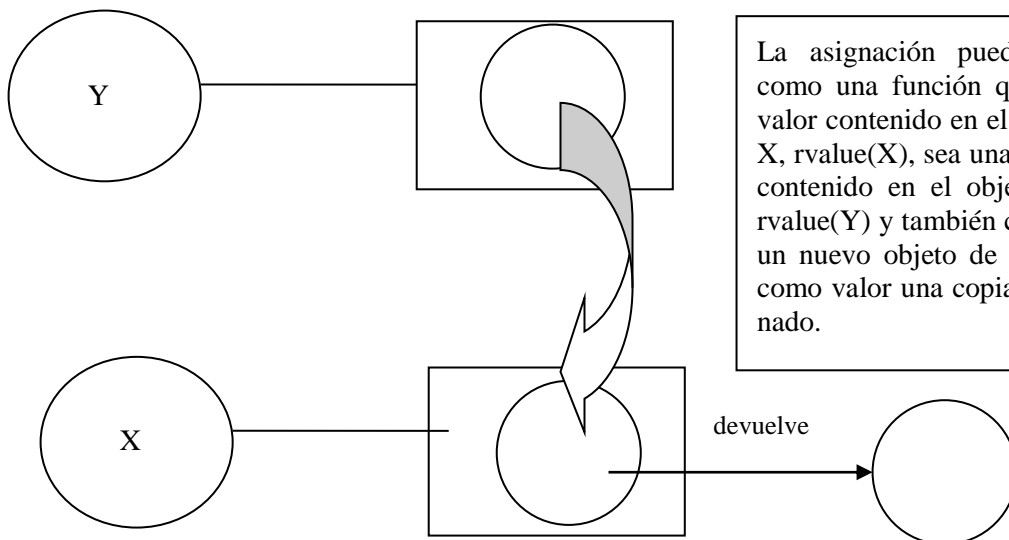
En lenguaje Pascal, la sintaxis de la asignación: **X:=Y**



La asignación se interpreta como una función que hace que el valor contenido en el objeto de datos X es una copia del valor del objeto de datos Y, no devolviendo esta función ningún resultado explícito.

De ahí que la asignación es un efecto colateral de la función.

En lenguaje C, la sintaxis de la asignación es: **X=Y**



La asignación puede interpretarse como una función que hace que el valor contenido en el objeto de datos X, $rvalue(X)$, sea una copia del valor contenido en el objeto de datos Y $rvalue(Y)$ y también crear y devolver un nuevo objeto de datos que tiene como valor una copia del valor asignado.

Por lo tanto, si en lenguaje C se tiene la siguiente asignación:

X= Y + 3*2;

La operación que se realiza sería la siguiente:

- 1) Computar el valor l de la variable X
- 2) Computar el valor r de la expresión $Y + 3*2$, es decir, computar el valor r de Y y finalmente computar el valor r de toda la expresión.
- 3) Asignar el valor r calculado al objeto de datos del valor l computado.
- 4) Devolver el valor r del objeto de datos antes asignado.

Esta manera de interpretarse la semántica de la operación de asignación, se infiere que las asignaciones múltiples en lenguaje C son válidas.

Ejemplo

```
int a=10, b, c;
c=b+a*2;
```

Tipos de datos reales

Especificación: Un objeto de este tipo puede tomar cualquier valor real dentro de una serie ordenada de valores, y limitada por un valor mínimo y un valor máximo determinado por el hardware. Cabe tener en cuenta que los valores no están distribuidos uniformemente en ese intervalo, como en el caso de los enteros.

La precisión en cuanto al número de dígitos para la representación decimal puede ser especificada por el programador tanto en **Pascal** como en **C**.

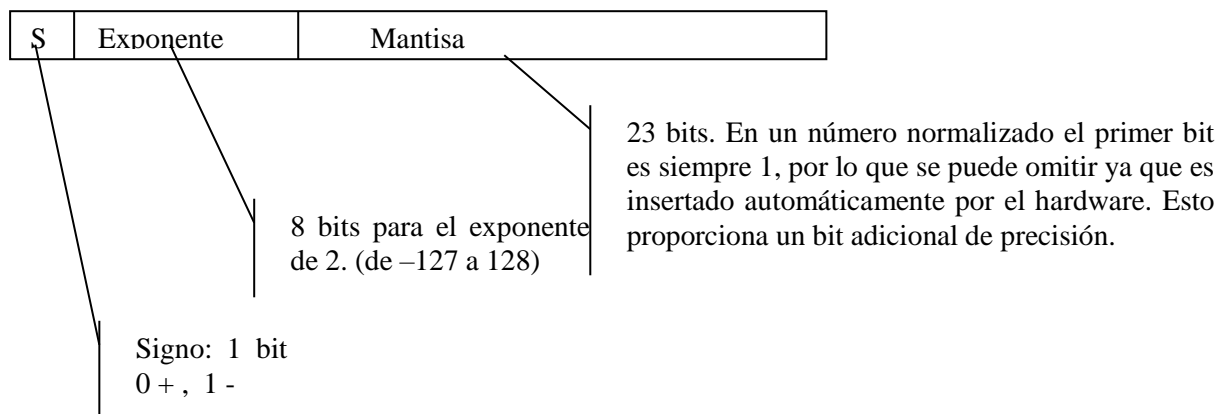
Las operaciones relacionales y aritméticas provistas para enteros, son válidas para reales, aunque algunas operaciones booleanas están restringidas debido a cuestiones de redondeo.

Implementación: Para representar valores reales de punto flotante, utiliza la misma representación de hardware, la que emula a la notación científica.

La norma **754 de IEEE** (1985) se ha convertido en la definición aceptada para la implementación de números reales de punto flotante de precisión simple o doble.

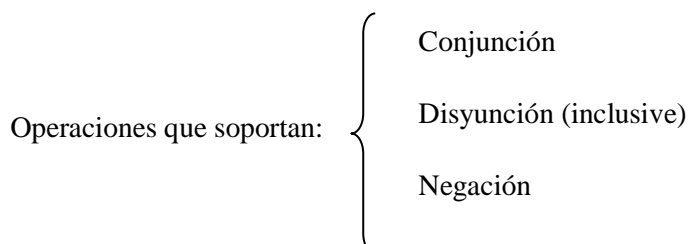
Utiliza 32 o 64 bits para números de punto flotante. Para números de punto flotante de precisión doble se agrega a la mantisa una palabra más (mantisa extendida).

La representación de número normalizado es la siguiente:

**Tipos de datos booleanos**

Especificación: Este tipo representa a objetos de datos que pueden tomar uno de dos valores posibles: True o False. En Lenguaje Pascal, el tipo boolean se considera como un tipo enumerado definido por el lenguaje, donde los valores simbólicos son TRUE y FALSE.

Esto es, Type boolean=(FALSE, TRUE);



Implementación: Si bien se necesita un solo bit para su representación (0 o 1), como los bit no son individualmente direccionables, se usa un byte o una palabra que es una unidad direccionable.

Hay dos maneras distintas de representar estos valores dentro de la unidad de almacenamiento:

1. Se usa un solo bit del byte para representar con 0 o 1 según sea *verdadero* o *falso*. Los bits restantes se ignoran. (*generalmente se usa el bit de signo*)
2. Un valor 0 en toda la unidad de almacenamiento representa *falso*, cualquier otro valor representa *verdadero*.

En lenguaje Pascal: Utiliza un byte False se representa con 0, True se representa con 1.

Lenguaje C no provee de tipos boolean pero se usan objetos del tipo de dato entero para representar valores verdadero o falso. En general con 0 se representa un valor falso, cualquier valor distinto de 0 representa verdadero, aunque lo óptimo es usar el 1 como verdadero para no tener problemas cuando se realizan operaciones de bits como el &.

Podría suceder por ejemplo:

Bandera=7; en binario es: 0000111

!Bandera; en binario es: 1110000 seguiría siendo verdadero.

El lenguaje C estandar no admite este tipo, pero C++ admite el tipo booleano (bool) y por lo tanto la siguiente declaración es válida:

`bool Band = false; // Band es una variable booleana, el valor verdadero se escribe true.`

Tipo de datos caracter

Especificación: Este tipo de dato toma como valor un solo carácter. El conjunto de posibles valores se corresponde generalmente con el conjunto de caracteres que maneja el hardware y el Sistema Operativo subyacente. Este conjunto, de caracteres tiene un orden, lo cual resulta importante cuando se construyen cadenas de caracteres. Es casualmente este orden el que permite el orden alfabético entre cadenas y las operaciones referidas a las mismas.

Pascal y C no poseen cadenas de caracteres, sino que se construyen a partir de tipo de dato estructurado arreglo.

En C, el tipo carácter es un subtipo del tipo entero. Cada carácter ocupa un byte y puede ser manipulado como un entero.

Respecto al tipo de dato caracter, prácticamente todas las computadoras utilizan para representar caracteres el código ASCII extendido (Código estándar Americano para el Intercambio de Información). Este código es una extensión del ASCII standard (128 caracteres), y si bien no es un código standard ofrece la posibilidad de trabajar con 256 caracteres. Estos caracteres están ordenados de 0 a 255, lo que permite la comparación de caracteres entre sí. Los dígitos están ordenados en su propia secuencia numérica y las letras dispuestas en orden alfabético precediendo las mayúsculas a las minúsculas.

En general, todas las computadoras manipulan los siguientes caracteres:

Conjunto de letras minúsculas: a..z. (excepto ch, ñ, ll)

Conjunto de letras mayúsculas: A..Z. (excepto CH, Ñ, LL)

El conjunto de dígitos decimales: 0..9.

Carácter de espacio en blanco.

Caracteres especiales: +, -, %, ... @

Signos de puntuación: , ; :

Una variable de tipo caracter puede tomar uno de los valores de este conjunto, el que se debe escribir entre apóstrofes para evitar confundirlo con el nombre de una variable, operador o número.

Ejemplo

```
char sexo
```

```
sexo='m'
```

En el ejemplo, sexo es una variable de tipo carácter que se utiliza para representar el sexo de un atleta; el carácter 'm' es un valor constante que se asigna a la variable sexo para indicar que el atleta es varón.

Implementación: Generalmente los objetos de datos de tipo carácter son manejados por el hardware y el Sistema Operativo (S.O) subyacente, debido a su uso en la entrada y salida de información. Generalmente la implementación de un lenguaje usa la misma representación de almacenamiento para caracteres que el hardware. Si se usa una cadena distinta a la que soporta el hardware (distinta generalmente en el orden de los caracteres) la implementación del lenguaje debe proveer las conversiones apropiadas a la representación del nuevo conjunto o implementar las operaciones relacionales en las cual incide la secuencia de ordenación.

Tipo de dato puntero (apuntador)

Un objeto del tipo apuntador, también llamado tipo de referencia o de acceso, posee la dirección o localidad de otro objeto (valor L del objeto) o puede contener un apuntador nulo es decir a ninguna dirección.

No todos los lenguajes proveen este tipo de datos, pero entre los tipos de datos elementales que provee el lenguaje C, el tipo de datos puntero adquiere relevancia al momento de trabajar con gestión de memoria de manera dinámica. Esto es, a través del uso de punteros el programador puede, en tiempo de ejecución crear variables y destruirlas cuando las considere innecesarias.

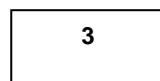
En lenguaje C, los punteros se utilizan para manejar estructuras de datos tales como arreglos y listas enlazadas. También son útiles para devolver resultados a través de los parámetros de una función. En capítulos posteriores se analizan estos temas.

Punteros a variables simples

Cuando se declara una variable, el sistema se encarga de reservar el espacio de memoria necesario para almacenar un valor del tipo declarado.

Así por ejemplo, dada la siguiente declaración `int x;` el sistema reserva 4 bytes para la variable entera x. Con `x=3` por ejemplo, se almacena el valor 3 en el espacio reservado a esa variable.

x



1002h

Se debe distinguir entre la dirección de memoria de una variable y su contenido.

El siguiente esquema de memoria representa esta situación:

1000h		
1001h		
1002h	3	x
1003h		
:		
1005h		
Dirección de memoria	Contenido	Identificador

En el caso presentado, la variable x está almacenada en la dirección de memoria 1002h y su valor o contenido es 3.

En la memoria de la computadora, cada dato almacenado ocupa varias celdas contiguas, dependiendo del tipo de dato de la variable declarada. Así, un entero ocupa 2 ó 4 bytes consecutivos según la representación de almacenamiento de entero definido por el hardware, un carácter 1 byte, etc.

Las celdas adyacentes de memoria están numeradas en forma consecutiva desde el principio al final de la misma. Este número, conocido como dirección de memoria se representa por lo general en sistema hexadecimal; F96 y EC2 son ejemplos de direcciones válidas de memoria. Por razones didácticas se simbolizará de ahora en más las direcciones con números decimales consecutivos, seguidos o no de la letra h de hexadecimal, así 1002h ó 1002 serán direcciones válidas.

Declaración de Punteros

Si se quiere definir un puntero p que contenga la dirección de memoria de una variable entera, los pasos a seguir son los siguientes:

Declarar las variables

`int *p, x;` Esto se lee: p es un puntero a un entero, x es una variable entera

Se debe notar que en la declaración, el nombre de una variable puntero es precedido por un asterisco.

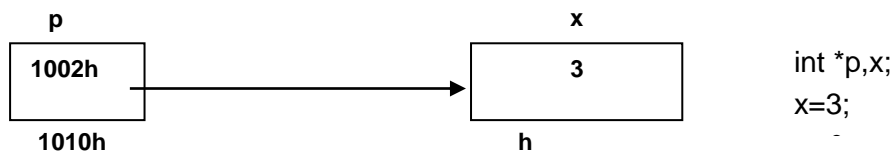
Asignar al puntero la dirección de memoria de la variable

Una forma de asignar un valor a una variable puntero es a través del uso del operador &, que devuelve la dirección de una variable.

Por lo tanto la asignación,
`p=&x;` guarda en p la dirección de la variable entera x.

Como el puntero contiene en este caso la posición de otra variable, se suele decir que la variable x está apuntada por el puntero p.

Si la variable x tiene asignado el valor 3, lo expuesto se representa:



Un puntero a una variable toma como valor, el valor L de la variable apuntada.

La siguiente tabla muestra el estado de la memoria después de la ejecución de las sentencias:

1000h		
1001h		
1002h	3	x
1003h		
:		
1005h		
:		
.		
1010	1002h	p
Dirección de memoria	Contenido	Identificador

Como el puntero p es también una variable, se almacena en memoria, en este caso en la posición 1010h.

Se debe tener en cuenta que, en la mayoría de los compiladores una variable puntero ocupa 4 bytes.

Formato:

En forma general, la declaración de un puntero es:

< tipo > * < nombre variable >

Donde: nombre variable es el identificador de la variable puntero, y tipo es el tipo de dato al que apunta el puntero.

Ejemplo

```
int x , *p;  
float puntajes[15] , *punt ;
```

Se declara la variable p y punt como punteros a variables entera y real respectivamente.

Es bueno destacar que una variable puntero es un tipo de dato simple, que puede apuntar a cualquier variable cuyo tipo sea simple o estructurado, tal como Arreglo, Struct, etc.

Operadores de punteros

Existen dos operadores que permiten trabajar con apuntadores: * y &.

Operador de dirección &

Este operador se utiliza para obtener la dirección de memoria de una variable.

&x se expresa “*la dirección de memoria de x*”.

Operador de indirección *

Este operador se aplica a una variable puntero y se utiliza para obtener el contenido de la posición de memoria apuntada por un puntero, es conocido también con el nombre de operador indirección o contenido.

***p** se expresa “*el contenido de lo apuntado por p*”.

Por tanto, en el ejemplo considerado, para acceder al valor de la variable x se puede utilizar directamente su nombre **x** o la expresión ***p**. Esto es, **x** es equivalente a ***p**.

NOTA: El operador * es unario, por lo que el compilador puede distinguir su aplicación respecto del operador * que representa la operación producto.

Ejemplo

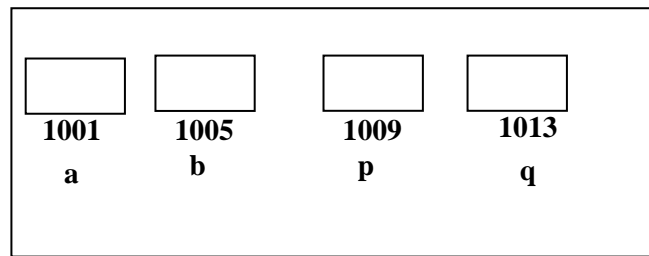
El siguiente esquema de memoria permitirá visualizar los distintos pasos del algoritmo que se muestra:

```
#include <stdio.h>  
void main (void )  
{  
    int a,b ,*p,*q;           1  
    p=&a;                     2  
    *p=8;                     3  
    q=&b;                      4  
    *q=23;                    5  
    printf(“%4d %4d \n”,a,b); 6  
    *p=*q+2;                  7  
    printf(“%4d %4d %4d %4d \n”,*p,*q,a,b); 8  
    p=q;                      9  
    printf(“%4d %4d %4d %4d \n”,*p,*q,a,b); 10  
}
```

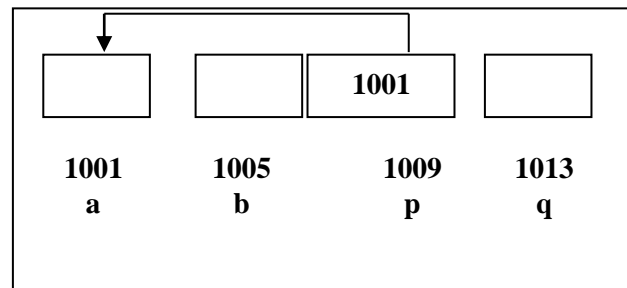
Paso 1

Se asigna espacio de memoria para las variables

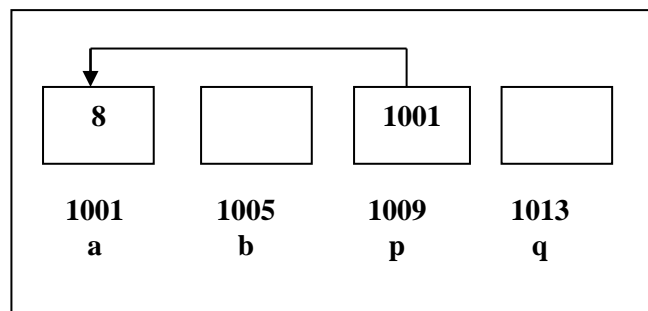
a, b, p y q.

**Paso 2**

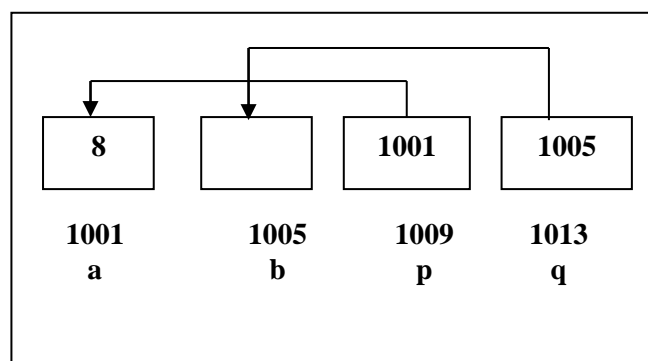
Se asigna al puntero **p**, la dirección de memoria de **a**, **p** apunta a la variable **a**.

**Paso 3**

Se asigna el valor 8 a la variable apuntada por **p**, ahora la variable **a** vale 8.

**Paso 4**

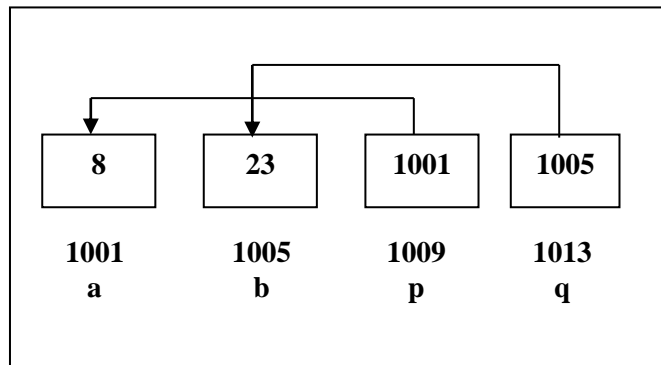
Se asigna al puntero **q** la dirección de memoria de la variable **b**.



Paso 5

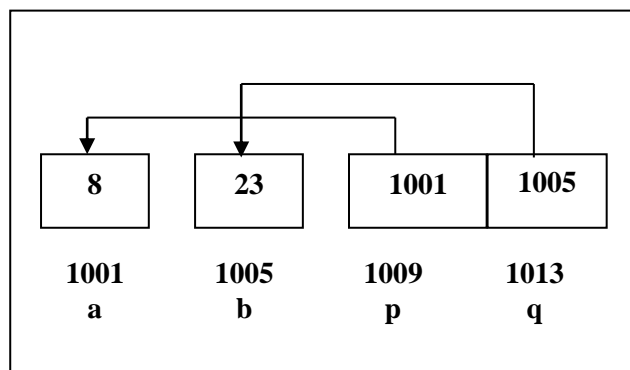
Se asigna el valor 23 a la variable apuntada por **q**,

b vale entonces **23**.

**Paso 6**

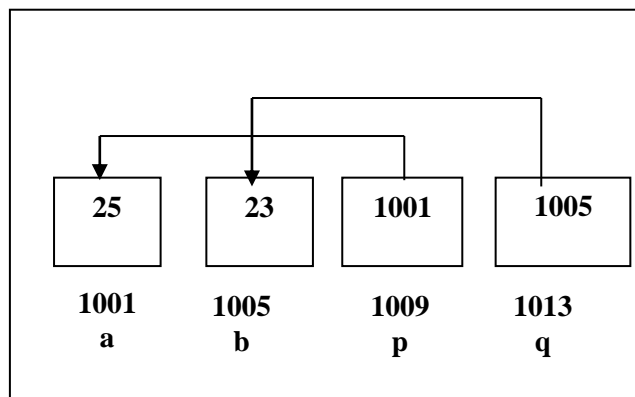
Muestra en pantalla el contenido de las variables **a** y **b**.

Salida: 8 23

**Paso 7**

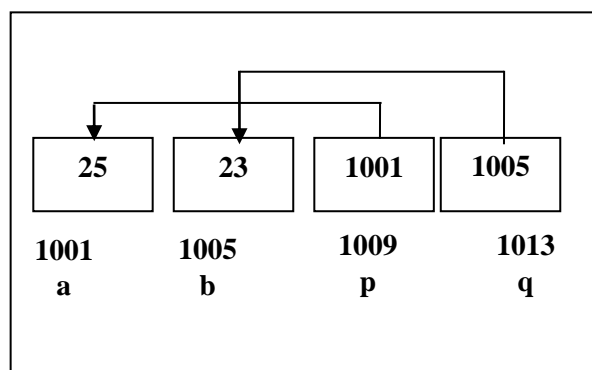
A la variable apuntada por **p** se le asigna el resultado de incrementar en 2 el valor de la variable a la que apunta **q**.

La variable **a** vale ahora 25.

**Paso 8**

Muestra en pantalla el contenido de lo apuntado por **p**, **q** y el de las variables **a** y **b**.

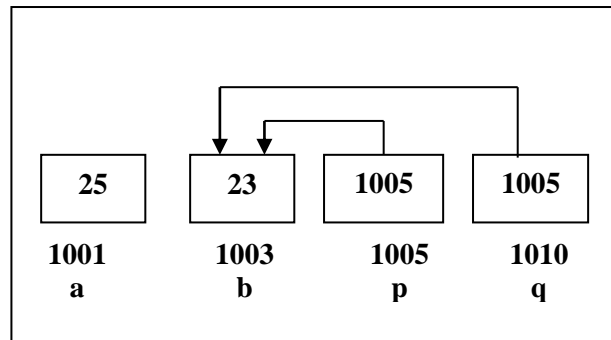
Salida: 25 23 25 23



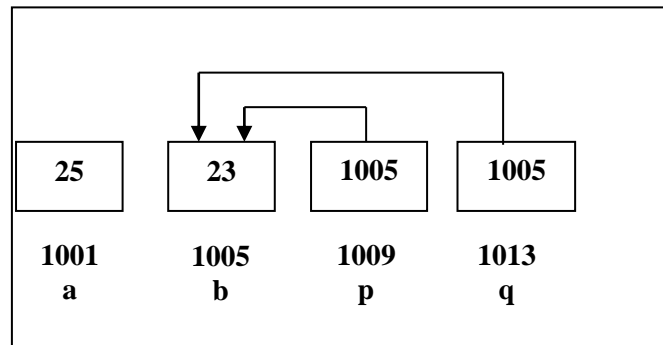
Paso 9

Al puntero **p** se le asigna el valor del puntero **q**.

Ambos apuntan ahora a la misma dirección de memoria, la variable **b**.

**Paso 10**

Se muestra por pantalla los contenidos de lo apuntado por los punteros **p** y **q** y el contenido de las variables **a** y **b** respectivamente.

**Salida:**

23 23 25 23

Asignación de valores a punteros

Existen tres formas para la asignación de valores a variables punteros: a través del operador **&**, por medio de otro puntero, o con una dirección de memoria constante.

- Por medio del operador &**

Como se ha visto, este operador permite asignar al puntero la dirección de memoria de una variable.

Ejemplo

```
char c , * punt;
```

```
punt =&c;
```

- Por medio de otro puntero**

A un puntero se puede asignar el valor de otro puntero que apunte al mismo tipo de dato.

Ejemplo

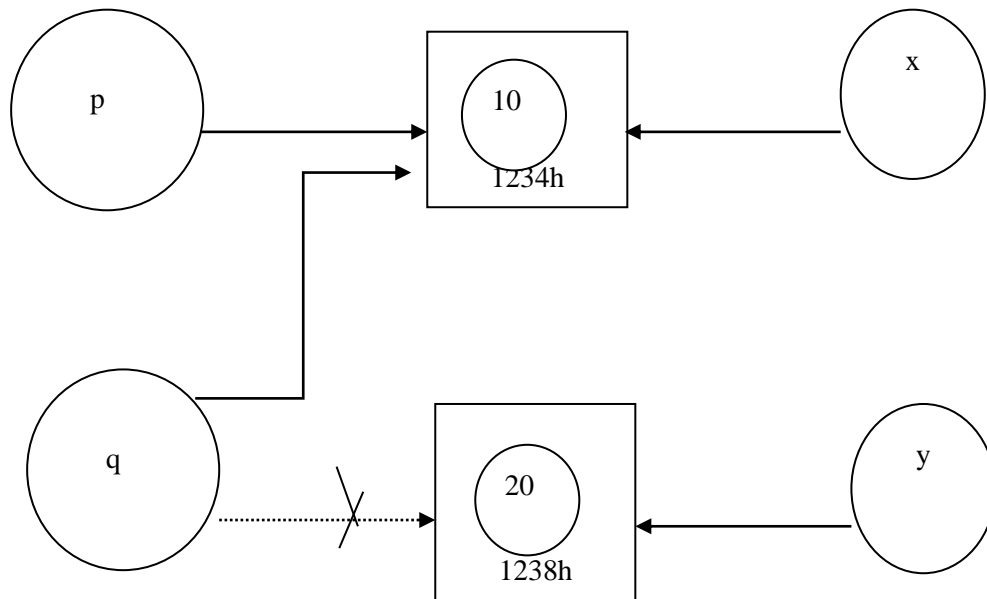
```
int x=10, y=20, *p , *q;
```

```
q=&y;
```

```
p=&x;
```

```
q=p;
```

En este caso, al puntero **q** se le asigna el valor del puntero **p**, por lo tanto ambos punteros apuntan a la variable **x**.



Como puede verse, la asignación de punteros en lenguaje C tiene una semántica distinta de la antes vista. La asignación entre punteros produce que se copien las direcciones en vez de los valores apuntados. Este tipo de asignación se llama asignación por compartición, y como se verá más adelante, provoca efectos colaterales no deseados en la asignación de objetos de datos estructurados del tipo struct, cuando estos incluyen campos o miembros de tipo apuntador.

Existen otros casos en los cuales a un puntero se puede asignar otro puntero que resulta de invocar a una función. Por ejemplo, la función malloc, devuelve como resultado un puntero a un bloque de memoria solicitado en forma dinámica. Este resultado puede ser asignado a otro puntero.

Esto se trata con más profundidad en la unidad 5.

Con una dirección de memoria constante

Cuando esto ocurre, al puntero se le asigna directamente la dirección de memoria a la cual va a apuntar. Por ejemplo si se asigna a un puntero la dirección de la memoria de vídeo o la parte baja de la memoria en donde se guardan datos referidos al sistema operativo como los muestra el siguiente ejemplo.

Ejemplo

```
int *pvar;
pvar=24A3;
```

Inicialización de variables punteros

Al igual que una variable de cualquier tipo, el lenguaje C no inicializa los punteros cuando se declaran, por lo tanto es necesario inicializarlos antes de usarlos. Los punteros pueden ser inicializados a NULL o a una dirección determinada.

NULL es una constante simbólica definida en el archivo de cabecera stdio.h . Se dice que un apuntador con el valor NULL apunta a ninguna dirección, esto es a ningún dato válido.

Un puntero puede ser inicializado con la dirección de una variable al momento de ser declarado, siempre que la variable esté previamente declarada.

En caso que no haya sido inicializado, un puntero quedará apuntando a una dirección de memoria desconocida en donde pueden existir datos. La alteración de estos datos puede producir resultados inesperados.

Comparación de punteros

Los punteros, al igual que las demás variables, se pueden comparar. La comparación es posible siempre que los punteros apunten al mismo tipo de datos.

Los operadores usados en la comparación, son los operadores relacionales conocidos:

== != < <= > >=

La comparación de punteros es válida siempre y cuando respete una lógica. Existen tres formas para comparar punteros:

- Dos punteros entre sí.
- Un puntero con la dirección de memoria de una variable, expresada por medio del operador &.
- Un puntero con una dirección de memoria dada directamente.

Ejemplo

```
#include <stdio.h>
void main (void )
{
    int *pa, *pb , x;
    :
    if ( pa ==&x )                /* comparación de un puntero con una dirección de variable */
        printf("El puntero está apuntando a la variable x");

    if ( pb != pa )               /* comparación entre dos punteros */
        printf("Los punteros apuntan a posiciones distintas");

    if ( pa >f041 )               /* comparación con una dirección determinada */
        printf("El puntero esta direccionado después de la posición f041");
}
```

Punteros NULL y void

Los punteros nulo (NULL) y genérico (void), son dos tipos de punteros muy utilizados.

El puntero NULL no direcciona ningún dato válido en memoria, mientras que el puntero void o puntero genérico es un puntero que apunta a cualquier tipo de datos. Dicho de otro modo, contiene la dirección de un dato de un tipo no especificado.

Este tipo de datos apuntador, será estudiado con más detalle cuando se trabaje con asignación dinámica de memoria.

Tipos de datos estructurados

Un objeto de datos que está construido por un agregado de otros objetos de datos, llamados componentes, se conoce como un objeto de datos estructurado o una estructura de datos. Algunos objetos de datos estructurados los define el programador y otros los define el sistema durante la ejecución del programa. Son ejemplos de estructuras de datos: arreglos, registros, cadenas de caracteres, listas, conjuntos y archivos.

Dado un grupo de tipos básicos como int , double y char, todo lenguaje ofrece diversas maneras para construir tipos más complejos, basándose en los tipos básicos; estos mecanismos se conocen como **constructores de tipo** y los tipos creados se conocen como **tipos definidos por el usuario**. Uno de los constructores de tipos más comunes es el arreglo.

Especificación de tipo de datos estructurados

Los atributos principales que se contemplan para especificar estructuras de datos son:

Número de componentes: Las estructura pueden ser de tamaño **fijo** o tamaño **variable**. Cuando es de **tamaño fijo** el número de componentes es invariable durante su tiempo de vida (arreglos y registros). Si el número de componentes cambia durante su tiempo de vida la estructura es de tamaño variable. Estas estructuras usan generalmente un puntero o apuntador para vincular las componentes. (Ej. Pilas, Listas, Archivos, etc.)

Tipo de componente: Si la estructura tiene componentes del mismo tipo de datos, se dice que es **homogénea** (Ejemplo Arreglos y Archivos). Caso contrario se dice que son estructuras **heterogéneas** (Ejemplo Registros).

Nombres que se debe usar para seleccionar una componente: Las estructuras de datos necesitan un mecanismo de selección para identificar componentes individuales de la estructura. En el caso de los arreglos, el nombre que lo identifica puede ser definido por el programador, mientras que una componente individual se identifica través de un subíndice. Para el caso de registros, generalmente el nombre que lo identifica lo define el programador. Para archivos por ejemplo, sólo se puede acceder a un componente particular, por ejemplo el componente apuntado en un momento determinado.

Organización de las componentes

1. Organización Unidimensional: serie lineal de componentes. (Ejemplo: arreglo unidimensional, registro, cadena de caracteres, listas, etc.)
2. Organización Multidimensional: Esta organización es similar a una serie lineal de componentes, donde cada componente es una estructura de datos de un tipo similar. (Ejemplo: arreglo multidimensional, registro cuyos componentes son registros, listas cuyos componentes son listas, etc.)

Operaciones sobre estructuras de datos

Hay operaciones que son propias de las estructuras, algunas de las cuales son importantes de destacar. En muchas ocasiones, el procesamiento de una estructura implica la recuperación de cada una de las componentes de la estructura. El acceso a una componente puede hacerse de dos maneras distintas:

- **Operación de Selección Directa:** permite el acceso arbitrario a una componente de una estructura.
- **Operación de Selección Secuencial:** en este caso, el acceso a una componente se realiza en un orden predeterminado.

Por ejemplo, al trabajar con un arreglo en lenguaje C,
`int a[10];` la selección directa de la tercera componente se hace a través de la operación de subindización `a[2]`,

y a través de la subindización y el uso del `for`, se puede seleccionar una serie de componentes
`for(i=0; i<10, i++)`
`.....a[i];`

Operaciones sobre una estructura de datos completa

Forman un conjunto limitado de operaciones, dependiendo del lenguaje. Por ejemplo, lenguaje Pascal provee un tipo de datos estructurado Set (conjunto) y dispone para este tipo de datos operaciones de union (+), intersección (*) y diferencia de conjuntos.

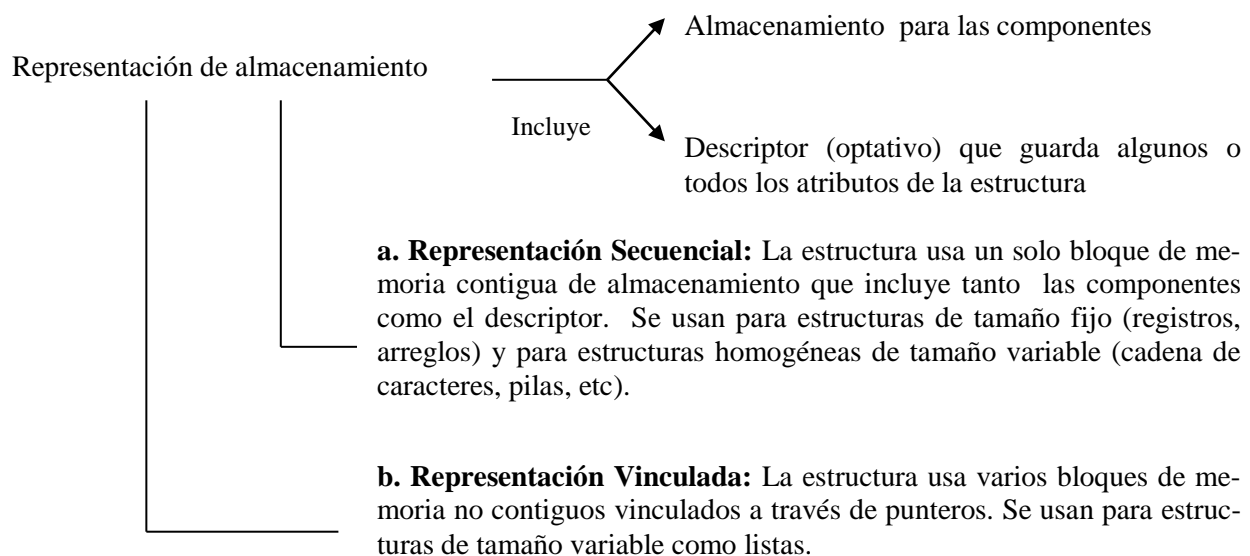
En este sentido, lenguaje C, provee la operación de asignación de registros, como operación sobre una estructura de datos completa. Otras operaciones, como la asignación de arreglos de igual dimensión, la suma de arreglos de igual dimensión, etc. deben ser definidas a través de funciones.

En lenguaje C, como se analizará más adelante, la asignación entre registros es válida siempre y cuando las componentes no sean punteros.

- *Inserción/eliminación de componentes:* Estas operaciones tienen un gran impacto sobre la representación de almacenamiento y la gestión de almacenamiento.
- *Creación/destrucción de estructuras de datos:* también tiene impacto sobre la gestión de almacenamiento.

Representación de tipos de datos estructurados

Se deben atender cuestiones que afectan fuertemente la elección de representaciones de almacenamiento: la selección eficiente de componentes de una estructura de datos y la gestión global eficiente del almacenamiento de memoria.



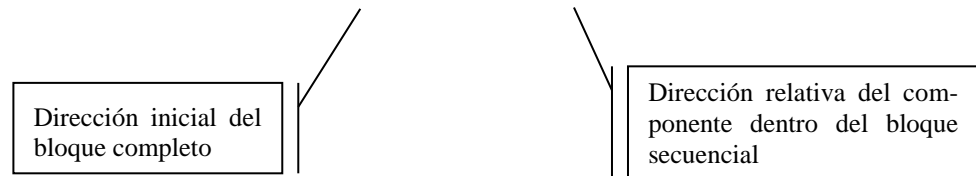
Importante: Casi todo el hardware carece de recursos para descriptores de estructuras de datos, manipulación de representaciones vinculadas, gestión de almacenamiento de estructura de datos y facilidades para el manejo de archivos externos. Por lo tanto las estructuras de datos y las operaciones que se realizan con ellas, deben simularse por software en la implementación del lenguaje de programación. Acá se observa una gran diferencia con los tipos de datos elementales en los cuales generalmente la representación de almacenamiento y las operaciones son manejadas por hardware.

Implementación de operaciones sobre estructuras de datos

Es importante la eficiencia tanto en la selección directa como en la selección secuencial

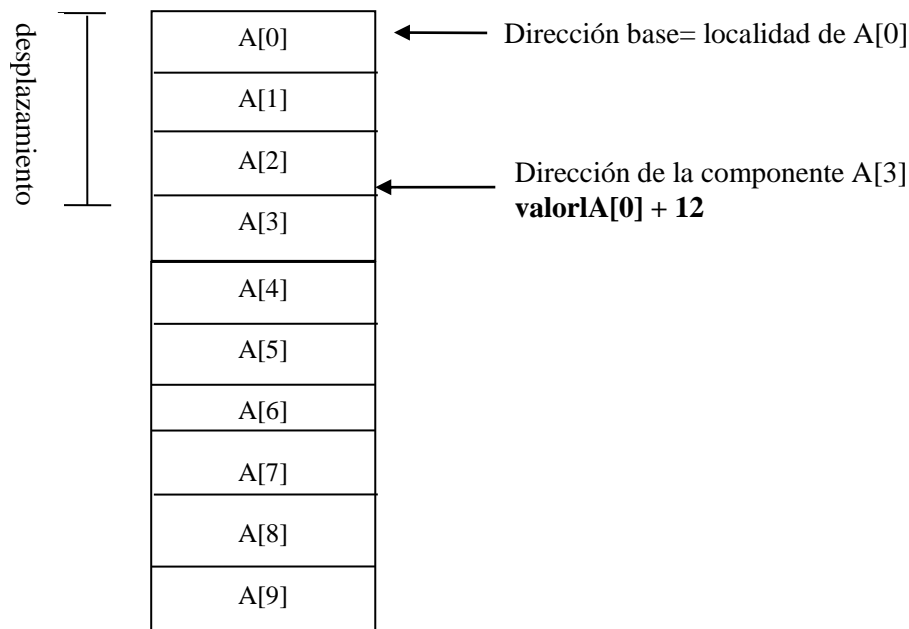
a) Representación secuencial: Para estructuras con representación secuencial la selección directa de un componente se calcula de la siguiente fórmula:

$$\text{Dirección de la componente} = \text{Dirección base} + \text{Desplazamiento}$$



Ejemplo

Sea en el lenguaje C, el arreglo `int A[10]`; se almacena de la siguiente forma:



Acceso a la componente `A[3]` en un arreglo de enteros en lenguaje C

b) Representación Vinculada: En este caso la selección directa implica seguir una cadena de apuntadores desde el primer bloque de almacenamiento de la estructura hasta la componente deseada. Este tipo de representación se analiza cuando se estudian listas enlazadas por punteros.

Declaraciones y verificaciones de tipo

Los conceptos y consideraciones a tener en cuenta sobre declaraciones y verificación de tipo para estructuras de datos, son similares a las expuestas para objetos de datos de tipo simple, no obstante las verificaciones son más complejas, pues son más los atributos a verificar.

Las declaraciones aportan varios atributos de la estructura. Por ejemplo, en la siguiente estructura,

int A[10][15]; los atributos son:

- tipo de datos: es un arreglo
- cantidad de componentes: 150
- dimensiones: 2
- tipo de datos de cada componente: entero

Cuando se hace la verificación de tipos la ruta a seguir para la verificación de una componente suele ser compleja. Además, aunque la verificación de tipo sea correcta en tiempo de compilación, nada nos asegura que un determinado desplazamiento sea válido en la estructura. En tiempo de ejecución, en alguna operación puede querer accederse a una componente inexistente y si por razones de eficiencia esto no se válida, entonces se produce un error de verificación de tipos.

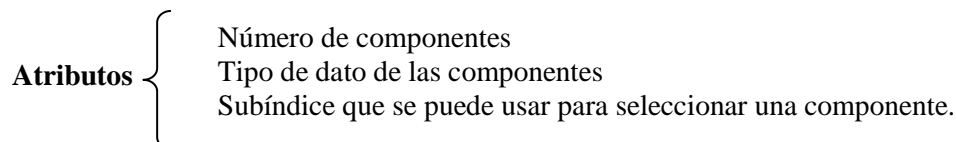
Clasificación de los tipos de datos estructurados

Los tipos de datos estructurados se clasifican en dos grandes grupos:

- **Tipos estáticos:** aquellos en los que la cantidad de componentes de la estructura esta predeterminada. Esto significa que al declararse un variable de este tipo, se reserva un espacio de memoria fijo que no podrá modificarse durante la ejecución del programa. Entre estos se encuentran: Arreglo (uni, bi y multidimensional), Registro y Archivo¹⁶.
- **Tipos dinámicos:** estos tipos tienen una cantidad de componentes que no está predeterminada al momento de su declaración. Esto significa que al declararse una variable de este tipo, se está reservando un espacio de memoria que se podrá modificar (aumentar o disminuir) durante la ejecución del programa. Entre estos se encuentran Listas, Árbol, Grafo, etc.¹⁷

Arreglos Unidimensionales

Un arreglo unidimensional (también llamados vectores) es una estructura de datos formada por un número fijo de componentes del mismo tipo de datos, organizadas como una serie lineal simple.



El arreglo es uno de los constructores de tipo más comunes.

La definición `int a[10]` crea una variable cuyo tipo es un “arreglo de enteros”.

El constructor arreglo toma el tipo base `int`, así como un tamaño (rango) y construye un tipo de datos nuevo. Esta construcción, que no tiene asociada un nombre (tipo anónimo), puede considerarse como una construcción implícita de un nuevo conjunto de valores, que se describe indicando la forma en que los valores pueden ser manipulados y representados.

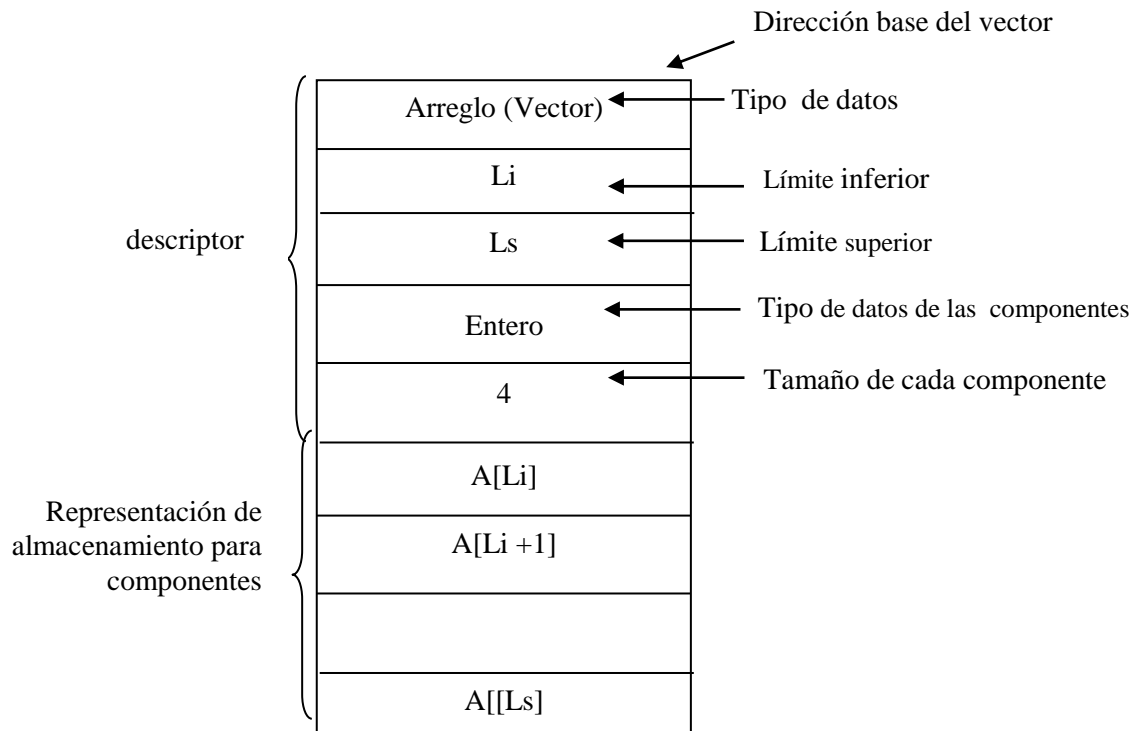
¹⁶ Tipo archivo se verá en la unidad 6.

¹⁷ En este curso solo se verá el tipo arreglo dinámico y lista, que se desarrollará en la unidad 5

Dado que el nombre de un tipo es de gran importancia para la verificación de tipos, para darle un nombre se utilizará typedef, como se verá más adelante.

Para los arreglos unidimensionales (vectores), la **representación secuencial de almacenamiento es la apropiada**. Se puede incluir un descriptor con algunos o todos los atributos, si esta información no se conoce hasta el tiempo de ejecución. En este descriptor también se puede guardar los límites inferior y superior del intervalo de subíndices que pueden usarse para acceder a un componente.

Una representación de almacenamiento completa de un arreglo unidimensional (vector) es la siguiente:



En general, Li y Ls se guardan si se quiere realizar verificación de subíndices. Los otros atributos no se guardan en tiempo de ejecución, sólo se necesitan durante la traducción para la verificación de tipos y para establecer la representación de almacenamiento.

Operaciones sobre arreglos (vectores):

La operación de selección de una componente se llama subíndización y se escribe con el nombre del arreglo seguido del subíndice que se desea seleccionar.

Puesto que los vectores son de tamaño fijo no se incluyen operaciones de inserción ni de eliminación.

En lenguaje C los subíndices son enteros, lenguaje Pascal por lo contrario, admite una gran variedad de subíndices que no tienen porqué ser enteros o subrango de enteros.

Hay lenguajes que permiten ciertas operaciones entre vectores, pero éstas pueden requerir almacenamiento temporal, y la gestión de este almacenamiento puede aumentar la complejidad y el costo de ejecución del programa.

Importante: La operación de selección de un componente o valor de datos de un objeto se debe distinguir de la operación de *referenciamiento*.

Para `int V[10]`; la selección de `V[5]` incluye dos pasos, a saber:

1. operación de *referenciamiento* que devuelve la dirección de V, valor L o Lvalue de V.
2. operación de selección que devuelve un apuntador a la localidad de ese componente

Implementación

En general es sencilla, debido a la homogeneidad de las componentes y el tamaño fijo de la estructura.

La homogeneidad



Implica que la estructura y el tamaño de cada componente es el mismo

el tamaño fijo



Implica que el número de componentes y la posición de cada una de ellas es invariante a lo largo del tiempo de vida del vector

Fórmula de acceso para una componente

Un arreglo en memoria siempre se almacena en posiciones contiguas a partir de la dirección base de comienzo del arreglo, la cual se obtiene cuando se declara una variable de este tipo (valor L).

El lenguaje C no necesita descriptores para los arreglos puesto que los índices siempre comienzan en cero, además el resto de la información no se necesita en tiempo de ejecución, ya que la misma se enlaza durante la traducción.

Pascal, permite definir arreglos donde los índices pueden tomar, por ejemplo, un subconjunto de valores enteros. En este lenguaje, el ámbito de subíndices deberá fijarse en tiempo de compilación de modo que puedan llevarse a cabo todos los cálculos de dirección en esta etapa, y por lo tanto no necesite descriptores durante la ejecución del programa.

En lenguaje C, **int a[10]**, declara un arreglo a de 10 componentes de tipo entero, donde los índices varían de 0 a 9.

En lenguaje Pascal, **a: Array [-3..6] of integer;** declara un arreglo a de componentes de tipo entero, donde los índices varían de -3 a 6.

Cálculo de la dirección en la que comienza a almacenarse un determinado elemento de un arreglo unidimensional:

Sea α la dirección del primer componente del vector, entonces la dirección de la componente $A[I]$ se calcula:

$$\text{Dir}(A[I]) = \text{ValorL}(A[I]) = \alpha + (I - L_i) * \text{Tamaño de la componente}$$

En lenguaje C, $L_i=0$, si llamamos E al tamaño de la componente, la dirección de la componente i-ésima se calcula con la siguiente fórmula: $\text{ValorL}(A[I]) = \alpha + I * E$

Como puede inferirse, en lenguaje C, el Valor L ($A[I]$) es una constante que se puede calcular en tiempo de traducción y esto hace el acceso a vectores mucho más rápido.

Luego, para el ejemplo en lenguaje C **int a[10]**, si la dirección α es 1000h, entonces la dirección de la componente $A[3]$, se puede calcular como:

$$\text{ValorL}(A[3]) = 1000 + 3 * 4 = 1012h$$

Arreglos bidimensionales y multidimensionales

A partir del concepto de vector, como un arreglo de una dimensión, puede definirse vectores de dos dimensiones (filas y columnas), de tres dimensiones (planos, filas, y columnas), y así sucesivamente.

Especificación de un arreglo multidimensional: La especificación difiere respecto a la de un arreglo unidimensional, en que para cada dimensión debe especificarse un intervalo de subíndices.

`int A[10][20]` declara en lenguaje C, un arreglo bidimensional de 10 filas y 20 columnas. Los índices varían de 0..9 para las filas, y de 0..19 para las columnas. A este tipo de arreglos se les llama tabla o matriz.

`Var A: array[3..12, -5..14] of integer;` declara en lenguaje Pascal, un arreglo bidimensional de 10 filas y 20 columnas. Los índices varían de 3..12 para las filas, y de -5..14 para las columnas.

Implementación

En general, un arreglo de cualquier dimensión está organizado o tiene una representación en *orden por filas*. Esta representación implica que el arreglo se divide en subvectores para cada elemento del intervalo del primer subíndice, luego cada uno de estos subvectores se divide en subvectores para cada elemento del intervalo del segundo subíndice, y así sucesivamente.

Para el caso de un arreglo bidimensional `int A[10][20]`, el arreglo A es un vector que tiene 10 subvectores de 20 componentes cada uno de ellos. A es un vector de vectores.

Una representación menos usada es la de orden por columnas, en la cual la matriz es considerada una sola fila de columnas.

El **tamaño** que ocupa el arreglo en memoria es el producto del número de sus elementos por el tamaño de cada uno de ellos.

En `int A[10][20]`; el tamaño es 4 byte * 200 elementos = 800 bytes

En el caso de un arreglo bidimensional, las componentes también se seguirán almacenando en posiciones contiguas de memoria y la forma en la que están almacenados esos elementos va a depender de que el arreglo se almacene fila a fila (orden por filas) o columna a columna, que también se conoce como almacenamiento de orden por columnas.

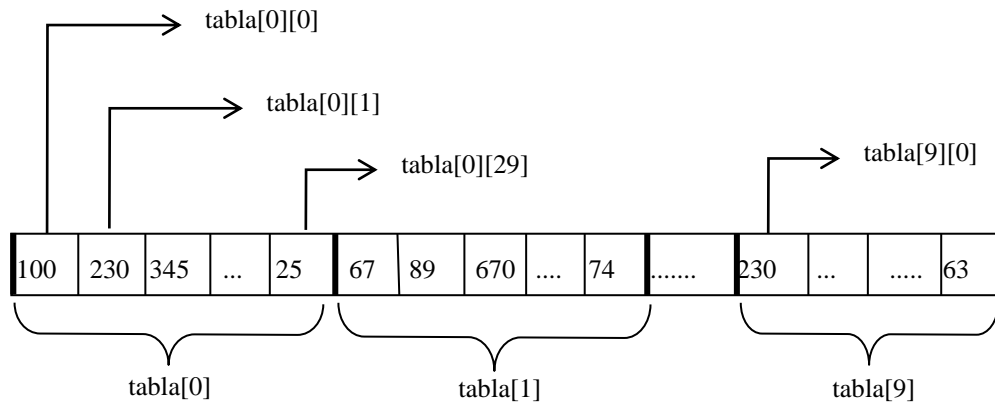
Por ejemplo, la siguiente tabla muestra el total de votos escrutados para 10 partidos políticos en 30 distritos.

		Distritos					
		0	1	2	29
Partido	0	100	230	345			25
	1	67	89	670			74
	2						
	3						
	8	811					125
	9	230					63

`int Tabla[10][30]` permite declarar un arreglo bidimensional en lenguaje C, para almacenar la información de la tabla.



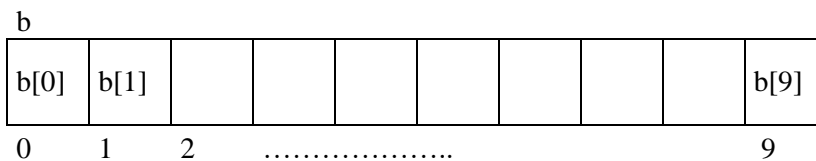
La información se almacenará en posiciones contiguas de la siguiente forma:



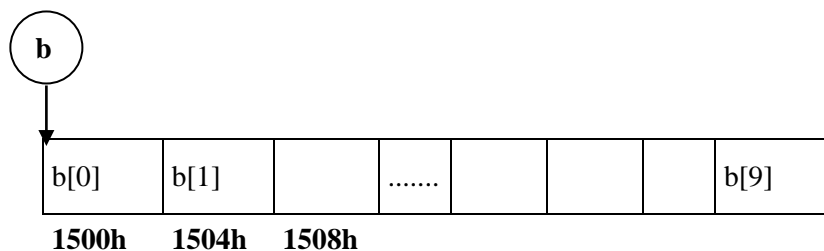
Arreglos y punteros en lenguaje c

En C existe una fuerte relación entre punteros y arreglos.

Hasta ahora para definir un arreglo **b** de 10 componentes enteras, hemos usado la expresión `int b[10]`, que permite reservar un bloque fijo de 40 bytes, cuya representación interna es la siguiente:



Teniendo en cuenta que el nombre de un arreglo es un puntero constante que contiene la dirección del primer elemento del mismo, **b almacena una constante, la dirección de `b[0]`**.



Por lo tanto la dirección del primer elemento del arreglo puede expresarse como **`&b[0]`** o simplemente **`b`**, la dirección del segundo elemento es **`&b[1]`** ó **`b+1`**

En general la dirección del elemento ubicado en la posición **`i`** del arreglo, puede expresarse como **`&b[i]`** ó **`b+i`**

Esta estrecha relación entre indexación en un arreglo y aritmética de punteros, permite acceder a la dirección de un elemento de un arreglo de dos maneras diferentes:

- escribiendo el elemento precedido por un **&**. Por ejemplo **&b[i]**.
- escribiendo el nombre del arreglo más una cantidad entera **i**. Por ejemplo **b+i**. Esta última expresión representa la dirección del elemento que está **i** posiciones después del primero. El valor de **i** se conoce como desplazamiento (offset).

Si **&b[i]** y **b+i** representan la dirección del elemento del arreglo ubicado en la *i*-ésima posición, entonces el acceso al contenido de esas direcciones se realiza con **b[i]** o ***(b+i)**

Por lo tanto, las siguientes expresiones son equivalentes:



Actividad 1

Si **b** un arreglo de 10 componentes enteras, completar con expresiones equivalentes.

***(b+6)** equivale a

&b[6] equivale a

***(b+2)+16** equivale a

Ejemplo: Carga y lista un arreglo

```
#include <stdio.h>
#include <conio.h>
const int N=5;

void carga (int x[], int lim)
{ int i;
  printf("\nCarga el arreglo con %d elementos\n", lim );
  for(i=0; i < lim; i++)
    scanf("%d",&x[i]); // equivalente scanf("%d",x+i);
}

void lista(int x[], int lim)
{ int i;
  printf("\n Componente Valor  Direccion \n" );
  for(i=0; i<N;i++)
    printf("\n x[%2d] %10d %10x ",i,x[i], x+i);
}

int main(void)
{ int a[N],i;
  carga(a, N);
  lista (a, N);
  for(i=0; i<N;i++)
    *(a+i)=*(a+i)*3;
  lista (a, N);
  printf("\n Valor del puntero %x ",a );
  printf("\n La direccion del primer elemento del arreglo es % x", &a[0] );
  getch();
}
```

Cadenas de caracteres en C

Una cadena de caracteres es un objeto de datos compuesto de una serie de caracteres.

Especificación y sintaxis: Existen distintos tratamientos de las cadenas, dependiendo del lenguaje¹⁸.

Cadena de longitud fija declarada en el programa.

Cadena de longitud variable hasta un límite máximo declarado en el programa.

Cadena de longitud no determinada.

En el caso del lenguaje C, el uso de cadenas de caracteres es muy particular. Este lenguaje no provee un tipo de datos cadena de caracteres. Al igual que en Pascal, utiliza arreglos de caracteres para almacenar cadenas, estableciendo por convención que el carácter nulo “\0” sigue al último carácter de la cadena. Es decir, cuando se guarda una cadena de caracteres en un arreglo, el traductor anexa el carácter nulo a la cadena.

El siguiente ejemplo muestra la forma en que se almacena una cadena de caracteres en un arreglo.

```
void main(void)
{
    char cad[13];
    gets(cad); /* suponer que por teclado se ingresa la cadena LUNES 31 */
    puts(cad); /* muestra LUNES 31 */
}
```

En memoria, el almacenamiento de la cadena es el siguiente:

0	1	2	3	4	5	6	7	8	9	11	12
L	U	N	E	S		3	1	\0			

Para cadenas que responden a los casos 1 y 2 citados, la asignación de almacenamiento para cada objeto cadena de caracteres se realiza en tiempo de traducción. Para cadenas de longitud no determinada, donde las cadenas pueden tener cualquier longitud, y la misma puede variar durante la ejecución del programa, la asignación de almacenamiento es dinámica, por lo que debe hacerse en tiempo de ejecución.

El lenguaje C también provee la posibilidad de crear cadenas dinámicas, como se analizará más adelante.

Uso de funciones de cadena de la biblioteca estándar – Lenguaje C

Para manipular cadenas, se utilizan funciones que se encuentran declaradas en el archivo de cabecera `<string.h>`, de la biblioteca estándar de C.

Las funciones cadena tratan a las cadenas como apuntadores a tipos de datos `char`, por lo tanto tienen argumentos declarados de la siguiente forma: `char *` o bien `const char *` lo cual significa que la función espera una cadena que puede o no modificarse.

Cuando se invoca a una función de este tipo, se puede especificar como argumento, el identificador de la variable declarada como arreglo de caracteres o como puntero a un carácter. En el caso que el argumento sea un arreglo de caracteres, en realidad C pasa la dirección del primer elemento del arreglo de caracteres.

¹⁸ Pratt, Terence y Marvin Zelkowitz. Lenguajes de Programación: Diseño e implementación.

La siguiente tabla muestra algunas de las funciones más usadas para manipular, comparar y buscar elementos en cadenas, con su correspondiente formato.

Función	Formato y Aplicación
strlen	Size_t strlen(const char * origen) Devuelve la longitud de la cadena
Asignación y concatenación de Cadenas	
strcpy	char * strcpy(char * destino, const char * origen) Copia la cadena origen en la cadena destino, el resultado es la cadena destino a la que agrega el carácter nulo.
strncpy	char * strncpy(char * destino, const char * origen, size_t n) Copia n caracteres de la cadena origen en la cadena destino, con el objeto de realizar truncamiento o relleno de caracteres.
strcat	char * strcat(char * destino, const char * origen) Agrega la cadena origen al final de la cadena destino, devuelve la cadena destino.
strncat	char * strncat(char * S1, const char * S2, size_t n) Agrega los n primeros caracteres de S2 a la cadena S1, devuelve S1.
Comparación de Cadenas	
strcmp	int strcmp(const char * S1, const char * S2) Compara alfabéticamente S1 y S2. El resultado es un número Positivo si S1 > S2 Negativo si S1 < S2 Cero si S1 = S2
stricmp	int stricmp(const char * S1, const char * S2) Es igual a strcmp, pero sin distinguir entre letras mayúsculas y minúsculas.
strncmp	int strncmp(const char * S1, const char * S2, size_t n) Compara S1 con la subcadena formada por los n primeros caracteres de S2. Al igual que strcmp devuelve un valor que puede ser positivo, negativo o cero.
Búsqueda de Caracteres y cadenas	
strrchr	char * strrchr(const char * S, int c) Devuelve un puntero a la última ocurrencia del carácter c en S , devuelve NULL si c no está en S . La búsqueda se hace en sentido inverso, desde el último al primer carácter.
strstr	char * strstr(const char * S1, const char * S2) Busca la cadena S2 en S1 y devuelve un puntero a los caracteres donde se encuentra S2 o NULL si no lo encuentra
Conversión de Cadenas	
strlwr	char * strlwr(char * S) Convierte las letras mayúsculas de la cadena S a minúsculas.
strupr	char * strupr(char * S) Convierte las letras minúsculas de la cadena S a mayúsculas.
Conversión de Cadenas a Números	
atoi	int atoi(const char * S) Convierte una cadena en un valor entero, devuelve cero en caso que no pueda realizar la conversión.
atol	long atol(const char * S) Convierte una cadena en un valor entero largo (long).
atof	double atof(const char * S) Convierte una cadena en un valor double en coma flotante.

NOTA

- Las funciones que permiten convertir cadenas en números se encuentran en la biblioteca `stdlib.h`
- El tipo `size_t` es equivalente, dependiendo del sistema, al tipo `unsigned long` o `unsigned int`.
- Como se observa en la tabla, algunos argumentos de las funciones aparecen precedidos de la palabra **const**, esto permite identificar rápidamente cuales parámetros son de entrada y cuáles de salida.

Así por ejemplo, en la función `strcpy`:

```
char * strcpy( char * destino, const char * origen)
```

La cadena origen sólo se utiliza para obtener los caracteres a copiar, ella no cambia, es la cadena destino la que se modifica. Aquellos datos que se utilizan como parámetros de entrada llevan la palabra `const`, no así los de salida.

- En general, las funciones manipulan los primeros caracteres de una cadena. Sin embargo, a veces es necesario trabajar con los últimos caracteres, en estos casos se recurre al uso de punteros, como lo muestra el ejemplo siguiente:

El siguiente algoritmo muestra como a partir de una cadena que contiene el nombre y apellido de una persona, se extrae la cadena que contiene sólo su apellido.

```
char c1 [30]="Darío Pérez" ;
char c2[30];
char *p= c1;
p+= 6;          /* apunta a Pérez*/
strcpy(c2,p);
puts(c2);
```

Registros

Especificación: Un registro es una estructura de datos compuesta por un número fijo de componentes de distinto o igual tipo. Por lo tanto, es una estructura de datos de *longitud fija*.

Diferencias entre un arreglo(vector) y un registro:

Los componentes de un registro pueden ser heterogéneos u homogéneos

Los componentes se designan con nombres simbólicos en lugar de indizarse con subíndices.

Los componentes de un registro se llaman campos o miembros.

Operación básica: La operación básica es la operación de selección.

Ejemplo en lenguaje C:

Declaración de tipo

```
struct empleados
{
    char nom[10];
    int edad;
}
```



Declaración de variable

empleados e;

En este caso se ha declarado o definido **el tipo de datos empleados** y es posible definir variables de ese tipo, en este caso **e**

e.nom indica la operación de selección del campo nom

Atributos de un registro

Número de componentes (en el ejemplo anterior, la struct empleados tiene 2 componentes)

Tipo de datos de cada componente (arreglo de caracteres y entero)

Selector que se usa para nombrar cada componente (en el ejemplo anterior nom y edad)

Implementación: Para la implementación de un registro se usa una representación secuencial de almacenamiento.

Las componentes individuales pueden requerir descriptores para indicar su tipo de datos u otros atributos, pero ordinariamente no se necesita un descriptor del registro en tiempo de ejecución.

La selección de una componente se implementa con facilidad pues los nombres de los campos se conocen en tiempo de traducción, en lugar de que se realicen los cálculos que en tiempo de ejecución, requieren los arreglos.

Para el ejemplo anterior, la representación en memoria es la siguiente:

**Variables del tipo struct en lenguaje C**

Una vez definido el tipo de dato struct (registro), se deben declarar las variables de la misma manera que se declaran las variables de tipos predefinidos.

El formato general de la **definición de un tipo struct** es:

struct <identificador><var 1>,...,<var n>;

Si por ejemplo definimos la siguiente estructura:

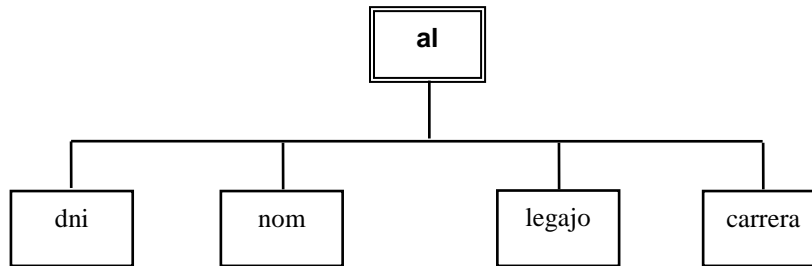
```
struct alumno
{
    int dni;
    char nom[20];
    int legajo;
    char carrera;
};
```

la declaración es: **struct alumno al;**

esto se lee la variable **al** es del tipo **struct alumno**



La variable **al** identifica a toda la estructura y se puede representar gráficamente en función de sus miembros, como sigue:



También es posible declarar la variable así :

```

struct alumno
{
    int dni;
    char nom[20];
    int legajo;
    char carrera;
} al;
  
```

Ejemplo

La siguiente estructura permite almacenar información de un determinado partido político:

struct partido_politico

```

{
    int numero;
    int votos [19];
};
  
```

Como se puede observar los miembros de una estructura pueden ser variables simples o estructuradas.

Acceso a los miembros de una estructura

Los miembros de una estructura se procesan generalmente en forma individual. El acceso a cada miembro de una estructura se realiza utilizando el operador de miembro de estructura (.) también conocido como *operador punto*, de la siguiente manera:

variable . miembro

En esta expresión, también conocida como *selector de miembro de una struct*, **variable** es el identificador de la variable de tipo estructura y **miembro** es el nombre de uno de los campos de la misma.

Para la estructura:

```

struct alumno
{
    int dni;
    char nom[20];
    int legajo;
    char carrera;
};
  
```

```

struct alumno al;
  
```

al. legajo y **al.carrera** hacen referencia al legajo y carrera de un alumno respectivamente.

Para la estructura:

```
struct partido_politico
```

```
{
```

```
    int numero;
```

```
    int votos [19];
```

```
};
```

```
struct partido_politico m;
```

m. numero refiere al número del partido político y **m.votos[5]** refiere a los votos obtenidos en el sexto distrito.

Anidamiento de struct

Si un miembro de una struct es otra struct, se está haciendo referencia a un anidamiento de structs.

Ejemplo

La siguiente estructura permite almacenar los datos de la cuenta corriente de un cliente de una determinada empresa.

struct fecha

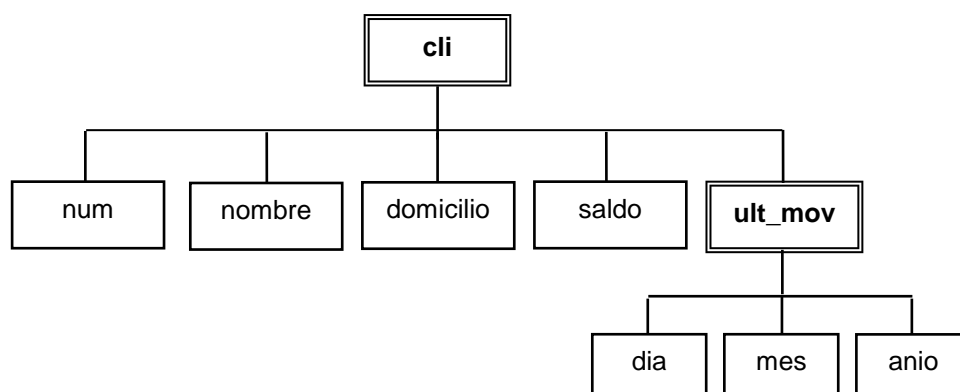
```
{
    int dia;
    int mes;
    int anio;
};
```

struct cliente

```
{
    int num;
    char nombre[20];
    char domicilio[30];
    float saldo;
    struct fecha ult_mov;
};
```

```
struct cliente cli;
```

Gráficamente



En este ejemplo, una estructura ha sido definida como miembro de otra estructura. En estas situaciones, la definición de la estructura interna debe preceder a la definición de la estructura externa que la contiene.

Actividad 2

Considerando la estructura del ejemplo, indicar como se accede al día, mes y año del último movimiento de un cliente.

```
struct fecha
{
    int dia;
    int mes;
    int anio; }

struct cliente
{
    int num;
    char nombre[20];
    char domicilio[30];
    float saldo;
    struct fecha ult_mov;
}
```

```
struct cliente cl;
```

Operaciones sobre estructuras

Las operaciones de **lectura y escritura** se deben efectuar individualmente para cada uno de sus miembros.

Para asignar valores a un miembro de una struct debe tenerse en cuenta que, una vez accedido mediante el selector, es tratado como cualquier otra variable del mismo tipo.

Sea **cl** una variable de tipo **struct cliente**, son correctas las siguientes expresiones:

```
scanf("%f",&cl.saldo);
scanf( "%d",& cl.ult_mov.dia);
gets(cl.nombre);
gets(cl.domicilio);
```

para ingresar respectivamente al saldo, día del último movimiento, nombre y domicilio de un cliente.

Sea **p** una variable de tipo **struct partido_político**, son correctas las siguientes expresiones:

```
printf("%d", p.numero);

for( i=0;i<19;i++)
    printf("%d", p.votos[i]);
```

para mostrar el número de un partido político y el total de votos obtenidos en cada uno de los 19 distritos.

Asignación de estructuras

La asignación entre estructuras está permitida si ambas son del mismo tipo y no contienen ningún miembro del tipo puntero. Entonces, dada la declaración:

```
struct alumno
{
    int dni;
    char nom[20];
    int legajo;
    char carrera;
};
```

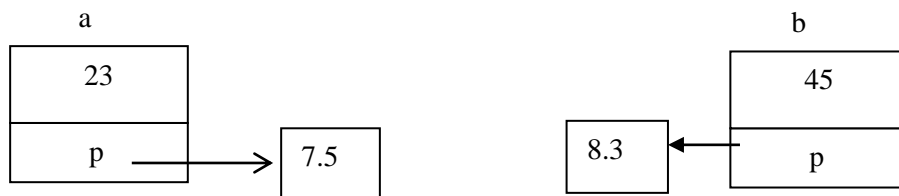
```
struct alumno a, b;
```

la asignación **a=b;** equivale a asignar el contenido de cada miembro de la variable b, a los miembros de la variable a.

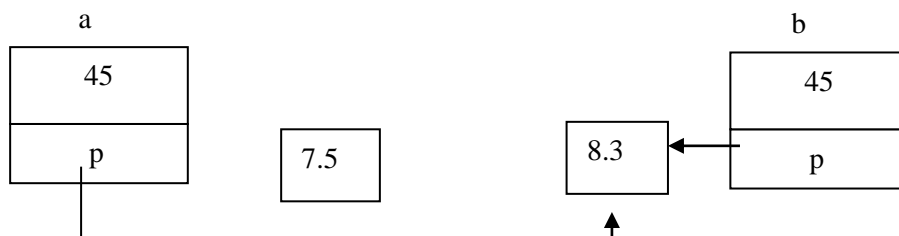
Para la declaración

```
struct datos
{
    int e ;
    float *p ;
};
struct datos a, b;
```

Supongamos que en memoria se almacenan los siguientes valores para cada uno de los componentes de a y b.



La asignación **a=b;** provoca la siguiente representación:



Esto provoca un efecto colateral no deseado, ya que en el caso que se modifique el valor apuntado por p desde el registro a, esa modificación afectaría al valor original apuntado por p desde el registro b.

De ahí que, cuando un tipo struct define una estructura donde uno de sus componentes es un puntero, la asignación tal cual se la interpreta en el capítulo 2, no es una operación válida.

Actividad 3: Realizar el seguimiento de los siguientes algoritmos y escribir conclusiones.

a)

```
typedef struct
{
    char nom[10];
    int edad;
} empleados;

main()
{
    empleados a,b;
    strcpy(a.nom, "carlos");
    a.edad=27;
    b=a;
    a.edad=30;
    printf("Registro a nombre %s edad %d", a.nom, a.edad);
    printf("\n Registro b nombre %s edad %d", b.nom, b.edad);
    getch();
}
```

b)

```
typedef struct
{
    char nom[10];
    int* edad;
} empleados;

main()
{
    empleados a,b;
    int e=27;
    strcpy(a.nom, "carlos");
    a.edad=&e;
    b=a;
    *(a.edad)= 30
    printf("\n Registro b nombre %s edad %d", b.nom, *(b.edad));
    getch();
}
```

Observaciones:

El operador punto (.), pertenece al grupo de mayor precedencia, por lo que tiene prioridad sobre los operadores monarios (++/--,etc).

Así la expresión, **++variable.miembro** es equivalente a **++(variable.miembro)**, siendo miembro una variable numérica.

Esto significa que el operador ++ se aplica al miembro de la estructura y no a la estructura completa.

La expresión **&variable.miembro** es equivalente a **&(variable.miembro)**.

Esto significa que la expresión accede a la dirección del miembro de la estructura.

La asociatividad del operador '.' es de izquierda a derecha. Así, la expresión del tipo **variable.miembro.submiembro** es equivalente a **((variable.miembro).submiembro)**

Punteros a estructuras

Así como un puntero puede apuntar a un entero, a un char, también puede hacerlo a variables estructuradas, en particular a una variable struct.

Un puntero a una variable struct contiene la dirección de comienzo de la estructura o valor L de la estructura.

Así, para la definición de tipo,

struct alumno

```
{ char nombre[20];
  int nota;
};
```

se puede declarar las variables,

struct alumno a, *pstrc ;

donde a es una estructura del tipo alumno y pstrc es un puntero a una variable del tipo alumno.

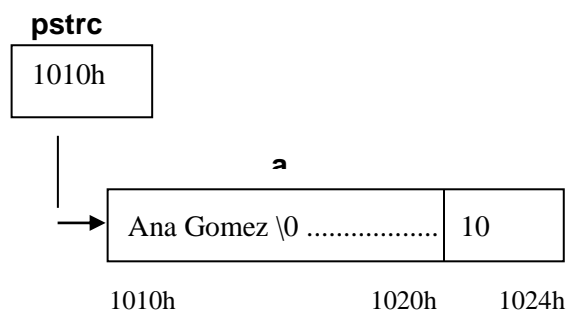
Las sentencias,

```
strcpy(a.nombre, "Ana Gomez");
a.nota=10;
```

asignan valores a cada miembro de la estructura.

Si el bloque de almacenamiento asignado en memoria para la estructura a, comienza en la dirección 1010h, entonces la asignación **pstrc=&a;** guarda en pstrc la dirección 1010h.

Gráficamente se pueden representar las variables:



Acceso a los componentes de una variable struct

Cuando se usan punteros a struct, el acceso a sus campos también se puede realizar utilizando el operador flecha -> (se forma con el signo menos seguido del símbolo mayor).

Por lo tanto, para acceder a través de un puntero a los miembros de una variable struct se pueden usar las expresiones:

(* puntero) . miembro

ó

puntero -> miembro

Por lo tanto, para la estructura definida:

pstrc->nombre	o	(*pstrc).nombre,	acceden al miembro <i>nombre</i> de a.
pstrc->nota	o	(*pstrc).nota,	acceden al miembro <i>nota</i> de a.

Ejemplo

El siguiente algoritmo muestra la forma de cargar y mostrar una estructura a través del uso de punteros.

```
# include <stdio.h>
struct alumno
{   char nombre[20];
    int nota;
};

void main(void)
{ struct alumno pers, *pstrc ;
  pstrc=&pers;
  printf("\n ingrese nombre ");
  gets( pstrc->nombre);           // equivale a  gets((*pstrc).nombre)
  printf("\n Ingrese nota");
  scanf("%d",&(pstrc->nota));      // equivale a scanf("%d",  & (*pstrc).nota)
  puts( pstrc->nombre);
  printf(" obtuvo %d puntos", pstrc->nota);
  getchar();}
```

Para acceder a submiembros de una estructura, se deben usar los operadores -> y punto.

Dadas las siguientes definiciones y declaraciones:

```
struct fecha
{ int dia, mes, anio;
} ;
```

```
struct egresado
{ char nombre[20];
  float promedio;
  struct fecha egreso; } ;
```

```
struct egresado egre, *p;
```

Realizada la asignación p=&egre;

Una de las formas de acceder al día, mes y año de egreso de un alumno egresado es:

```
p->egreso.dia
p->egreso.mes
p->egreso.anio
```

Actividad 4

Dada la siguiente estructura, que contiene los datos correspondientes a un cliente de una entidad bancaria:

```
struct domicilio;
{   char calle[30];
    int nro;
    cadena localidad;
} ;
```

```
struct cuenta
{
    int nrocli;
    char tipo;
    char nombre[30];
    float saldo;
    struct domicilio domi;
    float movi[10];
};
```

```
struct cuenta *pcli
```

Se pide:

Escribir la sentencia que permita mostrar el nombre de la calle correspondiente al cliente.

Si en el arreglo se han almacenado los 10 últimos movimientos realizados por el cliente, escribir la sentencia que permita mostrar la última operación realizada.

Registros variantes

En un registro ordinario, cada componente existe a lo largo del tiempo de vida del registro. En un registro variante existen algunos componentes, que dependiendo del valor marca, pueden estar o no presentes, si la marca cambia a su valor original.

Lenguaje C

```
typedef struct19
{
    float sueldo_mensual;
    int anio_inicio;
} asalariados;
```

```
typedef struct
{
    -----
    -----
} porHoras;
```

```
typedef union
{
    asalariados asalariado;
    porHoras porHora;
} sueldos;
```

Este tipo se llama **union libre** pues no admite marca.

Implementación: La implementación es más sencilla que su uso correcto. Durante la traducción se calcula el almacenamiento de las componentes variantes y se asigna almacenamiento para el registro en función de la variante más grande posible.

Durante la ejecución no se necesita un descriptor especial, pues la marca, si la hay, se considera un campo más del registro.

Selección de una componente: Se realiza de la misma forma que en un registro ordinario. Esto es, durante la traducción se calcula el desplazamiento del componente dentro del bloque de almacenamiento y durante la ejecución se suma a ese desplazamiento la dirección base.

Problemas de selección de componentes: Este problema ocurre cuando se desea seleccionar una componente inexistente, y su tratamiento es similar al error que se produce cuando al trabajar con arreglo hacemos referencia a un índice cuyo valor no está en el intervalo permitido.

Las posibles soluciones son:

Verificación dinámica. Validar la marca antes de tener acceso a las componentes

Ninguna verificación: Lenguaje Pascal provee formas de registros variantes sin campos marcas y C solo permite declarar uniones sin marcas, por lo tanto, estas implementaciones no permiten verificación, de ahí que puedan producirse errores tales como recuperar datos erróneos o sobrescribir datos en una componente que si existe.

¹⁹**typedef** permite definir la estructura una única vez y declarar varias variables con el mismo formato.

El lenguaje Pascal admite registros variantes con marca, el lenguaje C admite registros variantes sin marcas (union libre).

Struct y Union en lenguaje C

Las uniones son similares a las estructura ya que agrupan variables que pueden ser de distinto tipo, sin embargo difieren en su almacenamiento. Mientras en una estructura sus campos se almacenan en posiciones contiguas de memoria, en el caso de las uniones (que se declaran con la palabra reservada ***union***), los miembros se solapan entre sí en la misma posición.

En la *union* cada uno de los miembros comparten memoria con los otros miembros, por ello esta estructura de datos se utiliza cuando se trabaja con variables que no necesitan ser utilizadas al mismo tiempo. Si bien sintácticamente la *union* es similar a una struct, semánticamente existen diferencias.

A continuación la variable U del tipo *union* y su representación en memoria

```
union U
{
    int A;
    double B;
    char C;
}
```

```
struct T
{
    int A;
    double B;
    char C;
}
```

Para su almacenamiento la union U necesita sólo 8 bytes mientras que la struct T necesita de 13 bytes.

U

1000h		A / B / C
1001h		
1002h		
1003h		
1004h		
1005		
:		
1007		
Dirección de memoria	Contenido	Identificador

T

1000h		A
1001h		
:		
1003h		
1004h		B
:		
1011		
1012		C
Dirección de memoria	Contenido	Identificador

La union U tiene tres componentes, A, B y C, pero las tres ocupan la misma localidad de almacenamiento en memoria. T tiene también tres componentes, pero cada una de ellas tiene asignado un espacio de almacenamiento distinto. Por lo tanto, las tres componentes de U tienen el mismo valor L, el cual coincide con el valor L de U.

valorL[A]= valorL[B]= valorL[C]=valorL[U]=1000h

En la estructura, cada componente tiene un valor L distinto, y el valor L de T coincide con el valor L de A.

valorL[A]= valorL[T]= 1000h

valorL[B]= 1004h

valorL[C]=1012h

El siguiente ejemplo ilustra el comportamiento de la union:

```
#include <stdio.h>
union U
{
    int a;
    int b;
    int c;
};

main()
{ union U u;
  u.a=23;
  printf("u.a %d", u.a);
  printf("u.b %d", u.b);
  printf("u.c %d", u.c);
  getchar();
  u.b=10;
  u.c=45;
  printf("\u.a %d", u.a);
  printf("u.b %d", u.b);
  printf("u.c %d", u.c);
}
```

La salida de la ejecución de este programa es:

u.a 23 u.b 23 u.c 23

u.a 45 u.b 45 u.c 45

Como puede verse en este ejemplo, una componente de una union guarda el último valor asignado a cualquiera de sus componentes.

Actividad 5: Agregue al código anterior las sentencias que le permitan mostrar la dirección de los campos de la union.

Bibliografía

- Lenguajes de Programación Diseño e Implementación. Pratt Terrence y Marvin Zelkowitz. Editorial Prentice Hall. Hispano americana, S.A. Segunda Edición. 1987.
- Programación Orientada a Objetos. Técnicas Avanzadas de Programación. Fontela, Carlos. Buenos Aires. Editorial Nueva Librería. 2003..
- Lenguajes de Programación Principios y Práctica. Loudon, Kenneth C. Mexico, D.F. Editorial Thomson. 2004.

Unidad 3: Funciones

Introducción

Así como un lenguaje base suministra un tipo de dato primitivo entero y operaciones sobre enteros que hacen innecesario que el programador se preocupe de los detalles de la representación subyacente de enteros como serie de bits, del mismo modo se puede definir un tipo nuevo de datos de más alto nivel y un conjunto de operaciones sobre dicho tipo de dato, sin que el programador necesite ocuparse de los detalles de la implementación del mismo.

El objetivo actual en los diseños de lenguajes de programación es que los programadores puedan crear tipos nuevos de datos y operaciones sobre ese tipo, de modo que pueda ser usado como un tipo de dato provisto por el lenguaje.

Existen distintos mecanismos básicos para crear tipos nuevos de datos y operaciones sobre esos tipos. Es propósito de este apartado, entender que construcciones proveen los lenguajes procedurales, en particular C, para definir nuevos tipos de datos y describir la funcionalidad de los mismos.

Definición de tipos

Una *definición de tipos* proporciona un *nombre de tipo* junto con una declaración que describe la estructura de una clase de objetos de datos. Así, el nombre del tipo se convierte en el nombre de esa clase de objetos de datos, y cuando se necesita trabajar con un objeto de datos particular basta con proporcionar el nombre del tipo en lugar de repetir la descripción completa de la estructura de datos²⁰.

Dado un grupo de tipos básicos como `int`, `char` y `float`, todo lenguaje ofrece una variedad de formas para construir tipos más complejos, basándose en los tipos básicos. Estos mecanismos se conocen como *constructores de tipos*. Las *struct*, *union* y los *arreglos* son ejemplos de constructores de tipo en lenguaje C.

Los tipos creados por los constructores de tipos se llaman *tipos de datos definidos por el usuario*.

Ejemplo:

```
struct alumno
{   char nombre[20];
    int nota;
};
struct alumno arre[20];
int Tabla[10][30];
```

Los nuevos tipos creados con constructores no tienen automáticamente un nombre. Los nombres son importantes no sólo para documentar el uso de tipos nuevos sino para la verificación de tipos, de la cual ya se ha hablado antes.

Para asignar nombres a los nuevos tipos de datos se utiliza una *declaración de tipos*, en algunos lenguajes se denomina *definición de tipos*.

En lenguaje Pascal por ejemplo, si se necesita trabajar con varios registros que tienen una misma estructura, entonces el programador puede dar la definición de un nuevo tipo de datos a través de la palabra clave *Type*:

²⁰ El concepto de definición de tipos fue evolucionando y las primeras definiciones de tipos surgen con Pascal en 1970.

Type partido_politico = record

```
Numero: integer;  
Votos: array[1..19] of integer;  
End;
```

seguida de la declaración **Var P1, P2: partido_politico;**

De este modo la definición de la estructura de un objeto de datos del tipo *partido_politico* se da una sola vez, en lugar de repetirla tres veces para P1, P2 y P3.

En lenguaje C, la situación es algo distinta, los nuevos tipos de datos sólo se proveen a través de la construcción **struct**, luego:

```
struct partido_politico {  
    int Numero;  
    int Votos[19];  
};
```

también con:

```
typedef struct {  
    int Numero;  
    int Votos[19];  
} partido_politico;
```

define un nuevo tipo de datos de nombre *partido_politico*.

La declaración de tres objetos de datos de esa estructura se realiza de la siguiente manera:

struct partido_politico P1, P2, P3;

Algunos compiladores aceptan directamente esta declaración:

partido_politico P1, P2, P3;

Una definición de tipo permite separar la *definición* de la estructura de un objeto de datos, de los puntos en los cuales se va a *declarar* el objeto.

Por ejemplo, en lenguaje C:

```
struct complejo                    // definición de la estructura  
{  
    float real;  
    float imaginario;  
}  
void main(void)  
{  
    int a, b;  
    :  
    complejo c1, c2;               // definición del punto donde se van a crear los objetos c1, c2  
    :  
}
```


Lenguaje C y typedef

En varias ocasiones puede ser útil definir nuevos nombres para tipos de datos, estos nombres o 'alias' nos hace más fácil la declaración de variables y parámetros. Para esto C dispone de la palabra clave **typedef**, cuyo formato es:

typedef <tipo> <identificador>;

Por ejemplo:

typedef int entero,

significa que se ha dado un nuevo nombre al tipo de datos int, lo cual permite declarar variables **entero** a, b como si entero fuera un tipo de dato.

El siguiente cuadro muestra el uso de la construcción typedef en lenguaje C.

En Lenguaje C:

```
typedef struct
{
    float real;
    float imaginario;
}complejo;
```

Luego la declaración de variables es:

complejo c1, c2;

En Pascal *complejo* define un nuevo tipo de datos, no obstante, existen diferencias entre el **type** de Pascal y el **typedef** de C.

En lenguaje C la construcción **typedef** no crea un nuevo tipo de datos. El efecto producido por **typedef** es el de una sustitución por macro.

En nuestro ejemplo, cuando en el compilador encuentra la declaración de variables *complejo c1,c2*; se sustituye *complejo* por **struct complejo**.

Cuando se necesite trabajar con un objeto de datos con esa estructura, basta con proporcionar el nombre definido "complejo", en lugar de repetir su declaración.

Par el caso de arreglos, typedef permite que la declaración de una variable **arre** como un arreglo de 20 enteros, pueda ser usada como si fuera un tipo de dato. Lenguaje C utiliza el *typedef* de la siguiente forma.

typedef int arre[20];

Luego la declaración de variables es:

arre a,b ; //esta es la declaración de a y b como arreglos de 20 componentes enteras.
arre es usado como si fuera un tipo de datos.

Otros ejemplos de uso de typedef:

typedef char cadena[28];

```
main()
{
    cadena nombre="Blanca Flores";
    :
}
```

Sistema de tipos²¹

Los métodos utilizados para la construcción de tipos, el algoritmo de equivalencia de tipos, y las reglas de inferencia y corrección de tipos, se conocen de manera colectiva como *sistema de tipos*.

Si la definición de un lenguaje de programación especifica un sistema de tipos completo que pueda aplicarse estáticamente y que garantiza que todos los errores de corrupción se detectarán lo antes posible, entonces se dice que el lenguaje es *fuertemente tipificado*.

Pascal es un lenguaje fuertemente tipificado pese a que existen algunas fallas. No obstante, lenguaje C es un lenguaje que incluso tiene más fallas por eso a veces se lo conoce como un lenguaje débilmente tipificado. El lenguaje C++ ha eliminado muchas de las fallas de C, pero por razones de compatibilidad sigue siendo un lenguaje que no cumple con una tipificación fuerte.

Ventajas de la definición de tipos

1. Simplificación la estructura del programa.
2. Modificación más eficiente. Si se desea realizar cambios en la estructura definida, basta hacerlo una vez en la definición, en contraposición a lo que significaría la presencia de varias declaraciones del mismo tipo de datos.
3. En el uso de subprogramas, facilita el pasaje de argumentos, pues evita repetir la descripción del tipo de datos. En Pascal, esto es un requisito, pues los argumentos deben estar acompañados de un tipo de datos definido anteriormente. En lenguaje C, es una facilidad que se provee.

Subprogramas

La modularización, es una técnica usada en programación para hacer códigos más cortos y eficientes, consiste en reducir un problema complejo, en partes simples y sencillas (subproblemas), para luego buscar la solución por separado, de cada uno de ellos.

En C, se conocen como subprogramas o funciones aquellos trozos de códigos utilizados para dividir un programa con el objetivo de que cada uno realice una tarea determinada para resolver una parte del problema.

El uso de subprogramas definidos por el programador permite dividir un programa en un cierto número de componentes más pequeñas, cada una de éstas con un propósito específico y determinado.

Los subprogramas representan una operación abstracta definida por el programador.

Se sugiere el uso de subprogramas en distintas situaciones, a saber:

- Cuando un programa debe realizar reiteradamente una determinada tarea, el conjunto de sentencias que realiza dicha tarea se puede definir como un subprograma, al que se accederá las veces que sea necesario. Cada vez que se invoque ese subprograma, se podrá trabajar con un conjunto de datos distintos.
- Para lograr una mayor claridad y legibilidad en el programa principal, es óptimo definir subprogramas que realicen tareas específicas, aún cuando las mismas no se invoquen de manera repetida.

Por ejemplo:

```
#include <stdio.h>
int factorial (int a)
{
    int i, f=1;
    for(i=1; i <=a; i++)
        f*=i;
    return f;
}
```

²¹ Para entender mejor este concepto se sugiere leer Pág. 179 - Lenguajes de Programación. Louden. K.

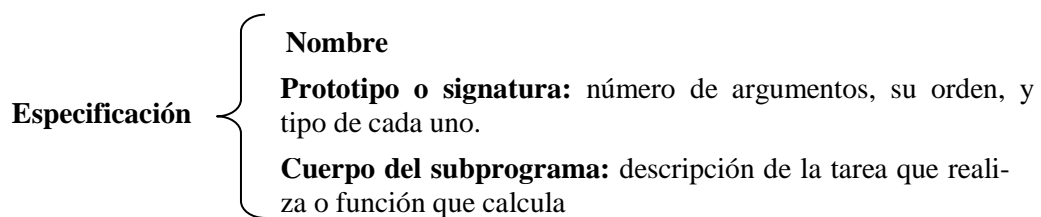
```

void main(void)
{
    int c;
    int m,n;
    printf(" ingrese m y n \n");
    scanf("%d",&m);
    scanf("%d", &n);
    c = factorial(m) / (factorial(n) * factorial(m-n)) ; // uso de la función factorial (invocación)
    printf("Combinaciones %d", c) ;
    getchar();
}

```

En este ejemplo, la función factorial es invocada tres veces.

Especificación de un subprograma



Ejemplo en lenguaje C:

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

typedef int vector[10]; → permite el uso del identificador vector como si fuera un tipo de datos

int escalar (vector a, vector b) → Operación producto escalar de vectores

```

{
    int e=0,i;
    for(i=0; i < 10; i++)
        e+=a[i]*b[i];
    return e ;
}

```

void carga(vector x) → Operación de carga del vector

```

{
    int i;
    printf(" \n ingrese las componentes del vector");
    for(i=0; i<10; i++)
        scanf("%d",&x[i]);
    return;
}

```

```

void main(void)
{
    vector v1, v2;
    carga(v1);
    carga(v2);
    printf(" el producto escalar es %d ", escalar(v1,v2));
    getchar();
}

```

En general, en los lenguajes procedurales se utilizan las siguientes denominaciones para los subprogramas:

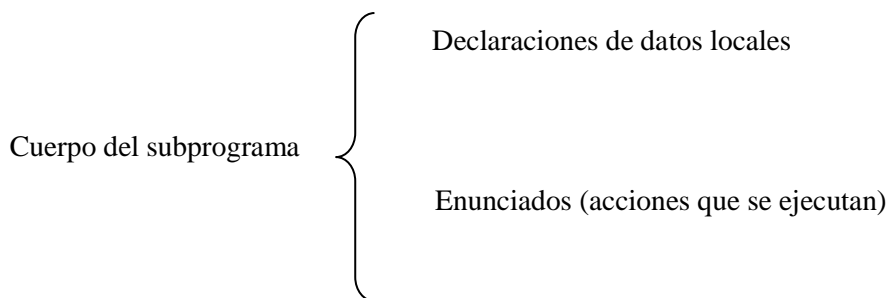
Si devuelve un resultado	→	función o subprograma
Si devuelve más de un resultado o modifica sus argumentos	→	procedimiento o subrutina

Implementación de un subprograma

Un subprograma es un conjunto de sentencias que realiza una tarea específica. Para su implementación se utilizan estructuras de datos y operaciones provistas por el lenguaje.

Un subprograma es una operación de la capa de una computadora virtual, construida por el programador.

La implementación está definida por el cuerpo del subprograma.



En **algunos lenguajes procedurales** el cuerpo puede incluir otras definiciones de subprogramas también encapsuladas, **en lenguaje C no es posible**.

En ambos casos, **la verificación de tipos es estática**, pues se provee el tipo de datos de los argumentos y del resultado.

En lenguaje C los subprogramas son funciones.

Es importante notar, que algunos lenguajes también proveen coerción de argumentos para transformarlos de manera automática al tipo de datos correspondiente.

Caso 1. Transforma un argumento int a float

```

float calculo( float a)
{
    int e=5;    /* Declaraciones de datos locales */
    a= a +e;    /* Enunciados (acciones que se ejecutan) */
    return a;   /*Resultado*/
}

main()
{
    int n;
    scanf ("%d", &n);
    printf ("\n %f", calculo(n)); /* Invocación*/
    getch ();
}

Salida: 11.000000 para un valor de n=6
  
```

Caso 2. Transforma automáticamente un float a int

```

void calculo(int a)
{
    /* No hay declaraciones de datos locales */
    printf("Valor del argumento %d", a); /* Enunciados (acciones que se
                                         ejecutan) */
    return (); /*no hay resultado, es opcional colocar la sentencia
               return*/
}

main()
{
    float n=23.40;
    calculo (n); /* Invocación*/
    getch ();
}

```

Salida: Valor del argumento 23

Definición, invocación y activación de subprogramas

Uno de los problemas principales en casi todos los lenguajes, es el diseño de recursos para definir e invocar subprogramas.

La **definición de un subprograma** es una propiedad estática de lenguaje; en tiempo de traducción es la única información que está disponible. Por ejemplo: tipo de variables de los argumentos, de las variables locales, etc.

Un subprograma, desde el punto de vista de la programación, se define como un proceso que recibe valores de entrada (llamados parámetros) y el cual retorna un valor resultado. Se pueden invocar (ejecutar) desde cualquier parte del programa, es decir, desde otra función, desde la misma función o desde el programa principal, cuantas veces sea necesario.

La **activación de un subprograma** se genera, en tiempo de ejecución cuando se lo llama o invoca. Al terminar la ejecución, la activación se destruye.

Por lo tanto, la definición es una plantilla, que permite generar activaciones en tiempo de ejecución.

Dada la definición de la función cálculo:

```

float calculo (float f, int i)
{
    const float por=10.7

    float g;
    int A[10];
    .....
    .....
}

```

La traducción de la definición da como resultado una **plantilla**, a partir de la cual se puede construir una activación particular de la función cada vez que se ejecuta el subprograma.

Por cada activación esta *plantilla completa* se podría crear en una nueva área de memoria, sin embargo la plantilla se divide dos partes con el fin de ahorrar memoria.

La plantilla se divide en una parte fija y otra dinámica:

El **segmento de código** es la *parte estática* compuesta por las constantes y el código ejecutable generado a partir de los enunciados del *cuerpo de la función*.

El **registro de activación** es la *parte dinámica*, compuesta por:

- Parámetros
- Resultados de la función
- Datos locales
- Punto de retorno
- Áreas temporales de almacenamiento necesarias para mantenimiento
- Vinculaciones para referencias de variables no locales

El registro de activación siempre tienen la misma estructura por cada activación, lo que cambia son los valores de los datos almacenados.

Los registros de activación se crean cada vez que se invoca un subprograma y destruyen cada vez que el mismo concluye con un retorno.

El tamaño y estructura del registro de activación se determina en tiempo de traducción, es decir, el compilador determina cuantos componentes tendrá el registro de activación y la posición de cada uno de ellos en la estructura.

Por lo tanto, para crear un nuevo registro de activación, solo hace falta conocer el tamaño del bloque, no la estructura interna en detalle, pues los desplazamientos se computan durante la traducción y en tiempo de ejecución solo se requiere la dirección base para acceder a un componente del registro.

Prólogo – Epílogo

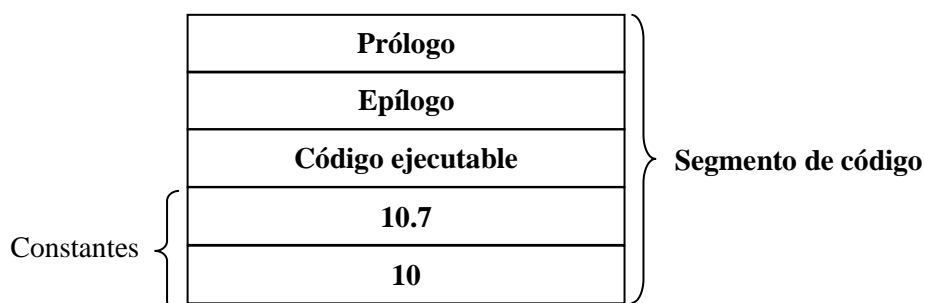
El **prólogo** es un bloque de código que el traductor introduce al comienzo del segmento de código, para permitir tareas de creación del registro de activación, transmisión de parámetros, creación de vínculos y actividades similares de mantenimiento.

El **epílogo**, al igual que el prólogo, es un conjunto de instrucciones que el traductor inserta al final del bloque de código ejecutable, para llevar a cabo acciones que permitan devolver resultados y liberar el almacenamiento destinado al registro de activación.

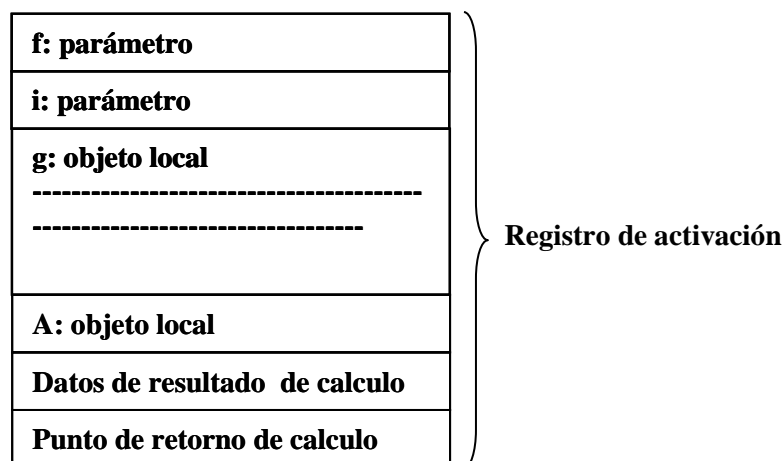
Por lo tanto, la estructura de una activación de la **función calculo**, antes definida es:

```
float calculo (float f, int i)
{
  const float por=10.7
  float g;
  int A[10];
  .....
  .....
}
```

En Memoria Estática (Área de Almacenamiento Dinámico):



En Memoria de Dinámica (Área Pila):



Funciones en lenguaje C

Así como el lenguaje C proporciona funciones de biblioteca para realizar distintas operaciones o cálculos frecuentes, también permite al programador definir y construir funciones que realicen tareas específicas.

El uso de funciones definidas por el programador permite dividir un programa en un cierto número de componentes más pequeñas, cada una de éstas con un propósito específico y determinado. Es por ello que se dice que un programa en C se puede *modularizar* a través del uso correcto de funciones. Al trabajar modularmente, descomponiendo un programa en varias funciones, se logra programas más fáciles de codificar y depurar.



Definición

Una función es un conjunto de sentencias que realiza una determinada tarea, que retorna como resultado cero o un valor.

Por ejemplo la función `getch()`, es una función predefinida que devuelve un único valor, un carácter.

```
char car;  
car=getch();
```

La función `clrscr()` es una función predefinida que no devuelve ningún valor.

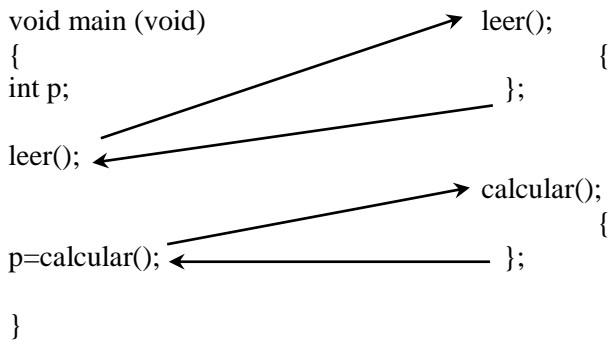
Un programa en C está formado por una o más funciones, siendo *main* la función principal por donde se inicia la ejecución del programa.

Al invocar una función desde algún punto del programa, se ejecutan las sentencias que forman parte de ella. Una vez que finaliza la ejecución de esa función, el control se devuelve al punto desde donde fue invocada, es decir, el control vuelve al `main` (programa principal) o a la función que la llamó.

Esquemáticamente:

```
void main (void)
{
  int p;
  leer();
  p=calcular();
}

leer();
calcular();
```



Si un programa contiene varias funciones, sus definiciones pueden aparecer en cualquier orden. Generalmente, una función usará o procesará la información que le es transferida desde el punto del programa donde es invocada o llamada. Esa información es comunicada a la función a través de identificadores llamados **argumentos** o **parámetros**. El resultado de la función se devuelve a través de la sentencia **return**.

La definición de una función consta de dos partes:

- **Cabecera** (la primera línea): incluye las declaraciones de los argumentos
- **Cuerpo** de la función.

La *primera línea* de la definición de una función contiene la especificación del tipo del valor devuelto, seguido del nombre o identificador de la función, y opcionalmente un conjunto de argumentos o parámetros, separados por comas y encerrados entre paréntesis. Cada argumento debe ir precedido por su declaración de tipo.

Si la función no incluye parámetros, el identificador de la función puede ir seguido de un par de paréntesis vacíos o con la palabra reservada *void*.

La primera línea de la definición o cabecera de una función, puede indicarse con el siguiente formato:

Formato:

<tipo de dato> <identificador>(< tipo1 arg1, tipo2 arg2, . . . , tipon argn>)

arg1, arg2, ..., argn, se denominan **argumentos formales** o **parámetros formales** e identifican a los datos que recibe la función en el momento de su invocación o llamada.

tipo1, tipo2, ..., tipon, representan los tipos de datos asociados a cada parámetro formal.

Al momento de invocar la función, los datos que se comunican a la función reciben el nombre de **argumentos reales**, **parámetros actuales** o **parámetros reales**, ya que se refieren a la información que realmente se transfiere. Cabe aclarar que los nombres de los parámetros formales y los nombres de los parámetros reales no necesariamente deben coincidir.

El cuerpo de la función se encierra entre llaves { } y contiene el conjunto de sentencias que se deben ejecutar cada vez que se la invoca.

Los parámetros formales y las variables declaradas dentro de una función, son locales a la función, pues no son reconocidos fuera de ella.

Cuando se usan funciones es importante tener en cuenta que:

- los parámetros actuales deben coincidir en tipo, orden y cantidad con los parámetros formales.
- el cuerpo de la función debe incluir al menos una sentencia *return* para devolver cero o un valor al punto de invocación o llamada. Es la sentencia *return* la que permite que se devuelva el control al punto de llamada o invocación sino se coloca, de modo implícito se realiza esta tarea en el punto donde se encuentra la llave de cierre de la función.

La instrucción *return* se especifica de la siguiente manera:

Formato:

return expresión;

expresión: se coloca si la función retorna un valor, si la función es tipo *void* no se coloca.

El valor de expresión, si existe, se devuelve a la función que llama. Si expresión se omite, el valor devuelto de la función es indefinido. El valor de expresión, si está presente, se evalúa y después se convierte al tipo devuelto por la función. Si la función se declaró con el tipo de valor devuelto *void*, una instrucción *return* que contiene una expresión genera una advertencia y la expresión no se evalúa.

Si no aparece ninguna instrucción *return* en la definición de función, el control vuelve automáticamente a la función de llamada después de que se ejecute la última instrucción de la función llamada. En este caso, el valor devuelto de la función llamada es indefinido. Si no se necesita un valor devuelto, se debe declarar la función para que tenga el tipo de valor devuelto *void*.

Si la función no retorna un resultado, la sentencia *return* puede no aparecer.

Ejemplo :

El siguiente código corresponde a la definición de la *función calcula*, que devuelve el perímetro de un terreno, cuyas dimensiones son recibidas en los parámetros formales.

```
float perimetro( float largo, float ancho) /*largo y ancho son parámetros formales*/
{
float perim;                               /* perim es una variable local a la función perimetro */
perim = 2 * ( largo + ancho) ;
return perim ;
}
```

Ejemplo

El siguiente código corresponde a la definición de la *función sumatoria*, que realiza el cálculo de:

$$\sum_{N=a}^{N=b} 2N + 1$$

para valores de *a* y *b* recibidos como parámetros formales. En este caso, la función no devuelve un resultado.

```
#include <stdio.h>
void sumatoria ( int a, int b )
{
int i, s=0;
for (i=a; i<=b; i++)
    s+=2 * i + 1;
printf(" la suma es %d ", s);
}
```

Ejemplo

La siguiente definición corresponde a la *función cabecera*, que imprime el membrete de una factura. En este caso, no recibe información y no devuelve un resultado.

```
#include <stdio.h>
void cabecera(void)
{
    printf("    EMPRESA UNION S.A  \n");
    printf("    Perú  123 Sur- Te: 02644378990  \n");
    printf("    San Juan  \n");
    return;
}
```

En casos como los dos últimos ejemplos, puede omitirse la sentencia `return` en la definición de una función, aunque esto no es una manera aconsejable de programar. En estos ejemplos, al llegar al final del cuerpo de la función, se devuelve el control al punto de llamada de la función sin retornar resultado alguno.

Como puede observarse, se coloca la palabra reservada *void* como especificador de tipo del resultado de la función y la sentencia `return` vacía.

También pueden definirse funciones que incluyan varias sentencias `return`, conteniendo cada una de ellas, una expresión distinta.

Ejemplo

La siguiente es la definición de una función que devuelve como resultado 'p', 'n' o 'c', según que el argumento recibido sea un número positivo, negativo o nulo.

```
char signo ( int num)
{
    if ( num > 0)
        return 'p ';
    else
        if ( num < 0 )
            return 'n ';
        else
            return 'c ';
}
```

Invocación o llamada a una función

La llamada o invocación a una función produce la ejecución de las sentencias del cuerpo de la misma, hasta que encuentra la sentencia `return` o la llave de final de cuerpo.

Se puede llamar o invocar a una función indicando su nombre seguido de la lista de argumentos encerrados entre paréntesis y separados por coma.

Formato:

<identificador de la función> (<arg1,arg2, . . . ,argn>)

`arg1,arg2, . . . , argn` se denominan **argumentos reales, parámetros reales o actuales**.

Estos argumentos deben corresponderse con los argumentos o parámetros formales que aparecen en la primera línea de la definición de la función. Si la llamada de la función no necesita argumentos, a continuación del identificador de la función se coloca un par de paréntesis vacíos.

La invocación a una función puede formar parte de una expresión cuando esta retorna un valor.

Ejemplo

Construir un programa en C que utilice una *función maximo* para determinar el mayor valor entre tres números enteros.

```
#include <stdio.h>
#include <conio.h>
int maximo ( int x, int y )           /* definición de la función maximo */
{
    int z;
    z = ( x >= y )? x:y ;
    return z;
}

void main (void )
{
    int a , b , c , d ;
    printf ( " \n a = " );           /* ingreso de las tres números enteros */
    scanf ( " % d " , &a );
    printf ( " \n b = " );
    scanf ( " % d " , &b );
    printf ( " \n c = " );
    scanf ( " % d " , &c );
    d = maximo (a, b );               /*primera invocación a la función maximo*/
    printf ( " \n el máximo de los tres números es: % d", maximo( c , d ) );
                                    /*segunda invocación a la función maximo*/
    getch();
}
```

En este ejemplo, se invoca a la función maximo desde dos lugares diferentes en el main:

- la primera invocación se efectúa en una asignación y los parámetros reales son las variables a y b.
- la segunda invocación se realiza en una escritura y los parámetros reales son las variables c y d.

Ambas invocaciones podrían reemplazarse por la siguiente línea:

```
printf ( " \n el máximo de los tres números es:% d", maximo ( c, maximo( a , b ) ) );
```

En este caso, una de las invocaciones de la *función maximo* es un argumento para la otra llamada, evitando así el uso de la variable d.

Ejemplo

Para dos números naturales distintos, el siguiente programa escribe un mensaje diciendo si uno de ellos es divisor del otro.

```
#include <stdio.h>
#include <conio.h>
int divisor (int x, int y)
{
    int res;
    ((x<y)&& ((y%x)==0))? res=-1:((x>y)&& ((x%y)==0))? res=1: res= 0;
    return res;
}
```

```
void main(void)
{
    int a,b,r;
    printf(" ingrese dos números naturales distintos\n");
    scanf("%d", &a);
    scanf("%d", &b);
    r=divisor(a,b);
    if (r== -1)
        printf("%d es divisor de %d", a,b);
    else
        if (r== 1)
            printf("%d es divisor de %d", b,a);
        else
            printf("no hay divisores");
    getch();
}
```

ACTIVIDAD 1

Construir una función que permita calcular la suma de todos los números menores o iguales a un número entero determinado. Codificar el programa principal con la invocación a la función.

Declaración de una función o prototipo de función

En los ejemplos realizados hasta ahora, se puede observar que la definición de una función siempre ha precedido a la función main. De esta forma, al compilar el programa, la función está definida antes de la primera invocación o llamada.

Sin embargo, el main puede aparecer antes de la definición de alguna función, es decir, la invocación de una función precede a su definición. Esta situación confundiría al compilador, salvo que se le alerte que la función a invocarse será definida más adelante en el programa. Con esta finalidad se utilizan los **prototipos de funciones**.

La forma general del **prototipo** de una función es:

<tipo de dato> <identificador>(< tipo1 arg1, tipo2 arg2, . . . , tipon argn>);

Puede observarse, que el prototipo de una función se corresponde con la primera línea de la definición de la misma, pero finaliza con punto y coma.

No es necesario declarar en ningún lugar del programa los parámetros o argumentos formales del prototipo de una función, pues éstos son argumentos ficticios que sólo se reconocen dentro del prototipo. Incluso, pueden omitirse los nombres de tales parámetros, ya que lo obligatorio es indicar el tipo de datos de cada parámetro formal.

Una función puede definirse en cualquier parte del programa, pero antes de su invocación debe estar presente al menos su declaración o prototipo, a veces llamada declaración por adelantado o declaración *forward*. Esta declaración informa al compilador que será accedida antes de que sea definida.

Ejemplo

En este ejemplo, la función divisor se define después de su invocación, por lo tanto su prototipo se incluye en la función main.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
    int divisor (int x, int y);    /* prototipo de la función divisor */
    int a,b,r;
    printf(" ingrese dos números naturales distintos\n");
    scanf("%d", &a);
    scanf("%d", &b);
    r=divisor(a,b);
    if (r== -1)
        printf("%d es divisor de %d", a,b);
    else
        if (r== 1)
            printf("%d es divisor de %d", b,a);
        else
            printf("no son divisbles" );
    getch();
}

int divisor (int x, int y)
{
    int res;
    ((x<y)&& ((y%x)==0)) ? res=-1:((x>y)&& ((x%y)==0)) ? res=1: res= 0;
    return res;
}
```

Organización de la memoria en C

Para la ejecución de un programa, el lenguaje C, al igual que otros lenguajes imperativos, usa una división tripartita de la memoria: estas tres áreas corresponden a:

- Un área de almacenamiento estático
- Un área asignada como pila (stack)
- Un área asignada como montículo (heap)

Almacenamiento Estático: La asignación de esta área de memoria se realiza en tiempo de traducción, y permanece fija durante la ejecución de un programa. En esa área se almacenan los segmentos de códigos de las funciones y otros datos del sistema.

Pila (stack): La pila es un bloque secuencial de memoria que se asigna en tiempo de ejecución. El lenguaje C utiliza la **pila** para gestionar las llamadas a las funciones. En este área se almacenan los registros de activación de las funciones,

La asignación de memoria en la pila se realiza secuencialmente a partir de un extremo, mientras que la liberación de la misma se realiza en orden inverso.

Generalmente, la pila crece hacia abajo, ocupando así, direcciones altas de memoria.

Cuando se invoca una función, el registro de activación correspondiente se almacena en la pila (se *apila*). Una vez que finaliza la ejecución de la misma, ese área se libera (el registro se *desapila*), retornando el control a la función llamante.

Montículo (heap): Un montículo es un bloque de almacenamiento dentro del cual se asignan o liberan segmentos de memoria. Sobre este bloque de almacenamiento se profundiza cuando se trabajan variables y estructuras dinámicas.

Los espacios destinados a la pila y al montículo se ubican en extremos opuestos de la memoria para que puedan crecer de manera conveniente.

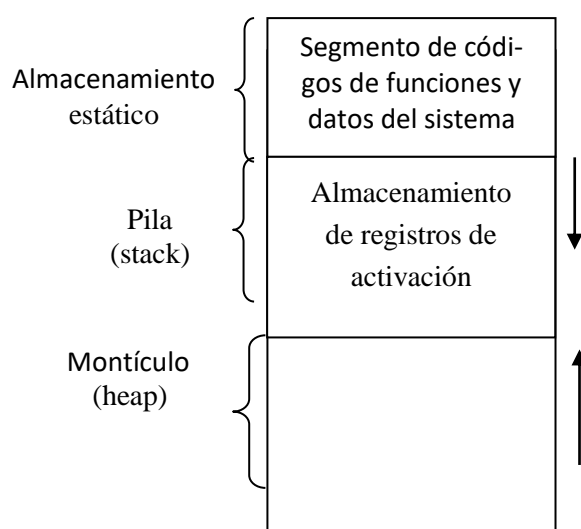


Fig. 1 Organización de la memoria durante la ejecución de un programa

Ejecución de un programa en C

Cuando se inicia la ejecución de un programa, en tiempo de compilación se generan los segmentos de códigos de las distintas funciones que el programa contiene.

En la pila, el primer registro de activación que se almacena corresponde al main, ya que la ejecución se inicia por esta función.

Cuando se invoca a una nueva función, se “apila” su correspondiente registro de activación, almacenándose la información necesaria en sus campos. Una vez finalizada la ejecución, ese registro de activación se “desapila” y queda sólo el registro de activación del main en la pila.

Ejemplo

El siguiente ejemplo, analiza la ejecución de un programa que calcula el perímetro de un terreno.

```
float perimetro ( float xl, float xa )
{
    float perim;
    perim = 2 * ( xl + xa );
    return perim;
}
```

```

void main(void)
{
    float largo, ancho;
    printf("ingrese el largo y el ancho");
    scanf("%f", &largo);
    printf("\n");
    scanf("%f", &ancho);
    printf("\n");
    printf("el perímetro del terreno es: %f", perimetro( largo,ancho ));
}

```

Al iniciarse la ejecución del programa, se genera el registro de activación del *main*. Este registro no reserva espacio para los parámetros ni para el resultado de la función *main*.

Al encontrar la invocación de la función *perímetro*, se genera otro registro de activación, en el cual se destina espacio para guardar: los parámetros formales *xa* y *xl*, la variable local *perim*, almacenamiento temporal, el resultado de la función y la dirección de retorno.

La figura 2 muestra el estado de la memoria en esta situación, para los valores *largo*=23.5 y *ancho*=13.6.

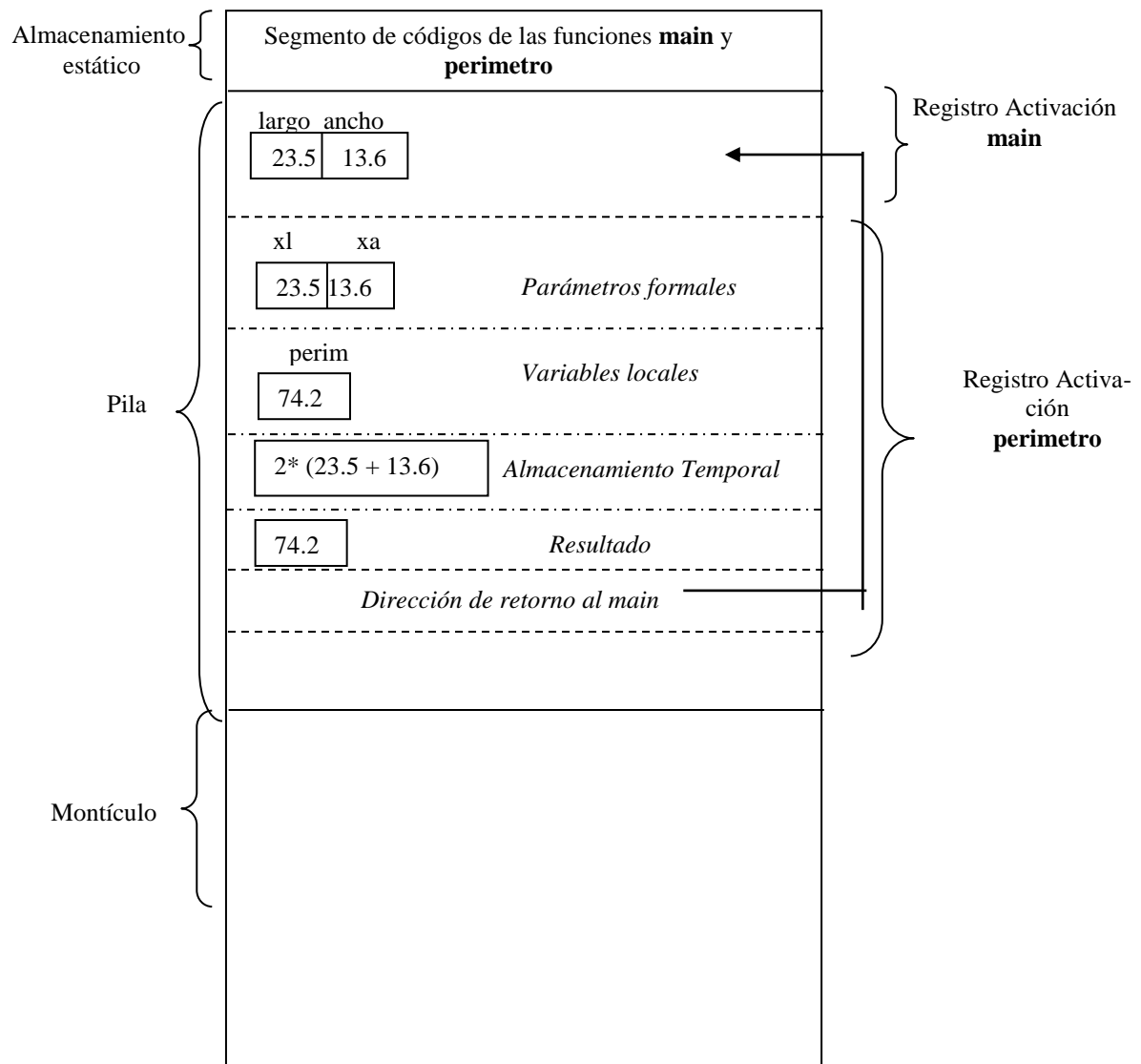


Fig 2. Organización de la memoria durante la ejecución de la función *perímetro*

Salida: el perímetro del terreno es: 74.2

Cuando termina la ejecución de la función *perímetro*, el sistema procede a desapilar el registro de activación asociado a ella en la pila y el control es devuelto al punto de llamada en el *main*. Una vez finalizado el programa, el sistema desapila el registro de activación del *main*.

Más adelante, cuando se trabaje el concepto de recursividad, se analizará el modo en que varios registros de activación de una misma función se pueden apilar consecutivamente.

Tipos de almacenamiento

Existen dos formas de clasificar variables: por su *tipo de datos* y por su *almacenamiento*.

El tipo de datos hace referencia al conjunto de posibles valores que puede tomar una variable. Por ejemplo una variable puede ser de un tipo de dato *int*, *float*, *char*, etc.

El tipo de almacenamiento hace referencia a la permanencia o tiempo de vida de la variable y a su alcance dentro del programa en el cual se reconoce.

Respecto a la permanencia, se puede observar que algunas variables tienen una existencia breve, otras son creadas y destruidas de manera repetida, y otras existen durante toda la ejecución del programa. El tiempo de vida de un objeto es la duración de asignación en memoria.

Hay tres tipos de almacenamiento: *estático* (para variables globales y estáticas), *automático* (para variables locales) y *dinámico* (para variables dinámicas).

La palabra reservada **auto** (automática) se utiliza para declarar variables de persistencia automática. Estas variables se crean al generarse el registro de activación de la función en la cual están declaradas, existen mientras dicho registro está activo, y se destruyen cuando se sale de ese bloque.

Variables Automáticas

Las variables locales de una función, aquellas declaradas en la lista de parámetros o en el cuerpo de la función, por lo regular tienen una persistencia automática. La palabra reservada **auto** declara en forma explícita a las variables de persistencia automática.

Respecto a su alcance, el mismo está restringido a la función en la cual se declaran. De ahí que todas las variables definidas dentro de una función no necesitan llevar el prefijo **auto** pues se asumen automáticas. Las variables automáticas se almacenan en la pila en el registro de activación asociado a la función.

Ejemplo

```
void fucion1( void)
{  doble d;
   int x ;
   ...
}
void fucion2 (void)
{  int y; float d ;
   ...
}
void main(void)
{  char z;
   ...
}
```

En este ejemplo, las variables automáticas de cada una de sus funciones son:

main: z

funcion1: d, x

funcion2: y, d

Como puede inferirse, las variables automáticas definidas en funciones diferentes serán independientes unas de otras, incluso si tienen el mismo nombre.

Se pueden asignar valores iniciales a las variables automáticas. Tales valores se reasignarán cada vez que se entre en la función, pues los valores de las variables automáticas se pierden cuando se sale de la función. Si una variable automática no es inicializada, su valor inicial será impredecible.

Variables Externas

Las variables externas difieren de las automáticas pues su alcance no se restringe a una función. Una vez definida una variable externa, toda función puede tener acceso a ella a través de su nombre.

Al trabajar con variables externas es necesario distinguir entre **definir y declarar** la variable.

Una variable externa debe definirse una sola vez, fuera de cualquier función. Esto reserva su espacio de almacenamiento en el área de almacenamiento estático. Además su definición puede incluir la asignación de un valor inicial.

Una variable externa también debe declararse en cada función que desee tener acceso a ella. Una **declaración** de variable externa tiene que comenzar con el especificador de tipo de almacenamiento **extern**. Su declaración no reserva espacio de memoria.

El nombre de la variable externa y su tipo de datos tienen que coincidir con su correspondiente definición de variable externa que aparece fuera de la función.

Si una función altera el valor de una variable externa, este valor está accesible por cualquier otra función que la utiliza.

Bajo ciertas circunstancias, la declaración de una variable externa puede omitirse. Tal es el caso de que la definición de la variable externa ocurra dentro del archivo fuente, antes de main. Se las llama también variables globales.

ACTIVIDAD 2

Realizar el seguimiento de los siguientes algoritmos para $j = 30$ y mostrar las salidas correspondientes.
a)

```
#include <stdio.h>
#include <conio.h>
int i=32;           /* definición de i como una variable externa y global */

void main(void)
{
    int j;           /* j es una variable automática de main */
    clrscr();
    printf("ingrese un entero ");
    scanf("%d",&j);
    int cambia(int j); /* declaración de la función */
    printf("\n Cambio realizado en j en el subprograma: %d",cambia(j));
    printf("\n Valor actual de j: %d",j);
    printf("\n Valor actual de i: %d",i);
    getch();
}

int cambia(int j)           /* j es una variable automática de cambia */
{
    j=j * 2;
    i=i * 2;
    return j;
}
```

En este ejemplo, la definición de la variable externa precede a la definición de la función, por lo que no es necesaria su declaración.

```

b)
#include <stdio.h>
#include <conio.h>

int cambia(int j)                                /* j es una variable automática de cambia */
{
extern i;                                        /* i es declarada como externa */
j=j*2;
i=i*2;
return j;
}

int i;                                           /* i es definida como externa */
void main(void)
{
    int j;                                       /* j es una variable automática de main */
    clrscr();
    printf("ingrese un entero ");
    scanf("%d",&i);                            /* ingreso 20 */
    printf("\n ingrese un entero ");
    scanf("%d",&j);                            /* ingreso 30 */
    printf("\n Cambio realizado en j en subprograma: %d",cambia(j));
    printf("\n Valor actual de j: %d",j);
    printf("\n Valor actual de i: %d",i);
    getch();
}

```

En este ejemplo la definición de la variable externa es posterior a la definición de la función **cambia**. Luego en la función **cambia**, la variable **i** debe declararse como externa.

ACTIVIDAD 3

¿Qué modificaciones debería realizar en el inciso b) de la actividad anterior, si la definición de la variable externa **i** se realiza después del **main()**?

Variables Estáticas

Las variables estáticas tienen el mismo alcance que las variables automáticas, son locales a la función en la que están declaradas. Sin embargo, respecto a su tiempo de vida, las variables estáticas retienen sus valores durante toda la vida del programa. Como consecuencia de esto, al invocar nuevamente a la función, se puede acceder al valor de la variable correspondiente a la última invocación. Esta característica permite a las funciones mantener información permanente a lo largo de toda la ejecución del programa. Las variables **static** se almacenan en memoria, en el área de almacenamiento estático, a continuación del segmento de código de la función.

La declaración de una variable estática deberá empezar con la especificación del tipo de almacenamiento, para lo cual se utiliza el prefijo **static**.

Las variables estáticas pueden usarse de la misma manera que las otras variables, sin embargo, no pueden ser accedidas fuera de la función en que están definidas.

En la declaración de variables estáticas se pueden incluir valores iniciales. Las reglas asociadas a la asignación de estos valores son esencialmente las mismas que las asociadas con la inicialización de variables externas, aunque las variables estáticas se definen localmente dentro de una función.

En general:

- Los valores iniciales deben ser expresados como constantes, no como expresiones.
- Los valores iniciales se asignan a sus respectivas variables al comienzo de la ejecución del programa. Las variables retienen estos valores a lo largo de toda la vida del programa, salvo que se le asignen valores diferentes en el curso de la ejecución.
- A todas las variables estáticas cuyas declaraciones no incluyan valores iniciales explícitos, se les asignará el valor cero. De esta manera, las variables estáticas tendrán siempre valores asignados.

Ejemplo

En el siguiente programa muestra el uso de variables estáticas

float promedio (int xa)

```
{  
    static int sum=0;  
    static int cont=0;  
    sum +=xa;  
    cont ++;  
    return (float) sum / cont ;  
}
```

void main (void)

```
{  
    float prom;  
    int a;  
    printf("\n ingrese un entero \n");  
    scanf("%d", &a);  
    while (a!=100)  
    {  
        if (a>0)  
            prom=promedio (a);  
        printf("\n ingrese un entero \n");  
        scanf("%d", &a);  
    }  
    printf("\n promedio positivos %f",prom);  
    getch();  
}
```

En este programa sum y cont son variables estáticas de la función promedio, por lo tanto retienen su valor después de cada ejecución de misma.

ACTIVIDAD 4

Realizar el seguimiento del programa anterior para el siguiente lote de prueba: 2, 8, -4, 5,-6,100.

Declaraciones, Bloques y Alcance²²

Declaraciones – Su importancia durante la traducción

Como se sabe, una declaración es un enunciado del programador que sirve para comunicar al traductor distinta información, primordial para establecer ligaduras.

Las declaraciones se utilizan para construir **en tiempo de traducción** la Tabla de Símbolos (TS), contribuyendo de ese modo a uno de los propósitos más importantes de las mismas: **permitir la verificación estática de tipos**.

Bloques en lenguaje C.

Un bloque es una secuencia de declaraciones seguidas por una secuencia de enunciados, y rodeados por marcadores sintácticos como son las llaves o pares inicio-terminación.

En lenguaje C, los bloques se conocen como enunciados compuestos y aparecen como el cuerpo de funciones en la definición de las mismas. Además, es posible también anidar bloques, como se muestra a continuación:

```
void p (void)
{  doble r;           /* el bloque de la función p*/
  .....
  {
    int x, y;          /* un bloque anidado */
    x=2 ;
    y=3 + x ;
  }
  .....
  .....
}
```

El lenguaje C también admite **declaraciones externas o globales** fuera de cualquier enunciado compuesto, como por ejemplo:

```
int x=10;             /* x es externa para todas las funciones y también es global*/

void main(void)
{
  float f;
  int g;              /* f y g  están asociadas al bloque del main*/
  .....
  .....
}
```

Alcance de un vínculo en lenguaje C:

Las declaraciones vinculan varios atributos a un nombre, el alcance de este vínculo es la región del programa donde este vínculo se mantiene.

En el lenguaje C, lenguaje estructurado en bloques, el alcance de un vínculo queda limitado al bloque donde aparece la **declaración asociada**, y a los bloques contenidos en el interior. Este alcance se llama **alcance léxico**.

²² Lenguajes de Programación Principios y Prácticas. Kenneth Loudon. Pág. 118.

El **alcance de una declaración** en lenguaje C, se extiende desde el punto siguiente a la declaración, hasta el final del bloque en el cual ésta se encuentra.

```
int x;
void p(void)
{  doble d;
  ...
}
void q(void)
{  int y;
  ...
}
void main(void)
{  char z;
  ...
}
```

En este ejemplo, las declaraciones de la variable **x**, y de las funciones **p**, **q**, **main** son globales, es decir tienen alcance global. Las declaraciones de **d**, **y**, **z** son **locales** respecto a las funciones p, q, y main respectivamente. Por lo tanto sus declaraciones son solamente válidas para dichas funciones.

Apertura en el alcance:

Una característica de la estructura de bloques es que las declaraciones en los bloques anidados toman precedencia sobre declaraciones anteriores.

```
int x;
void p(void)
{  doble d;
   int x ;
  ...
}
void q(void)
{  int y;
  ...
}

void main(void)
{  char z;
  ...
}
```

En este ejemplo, **la declaración de x** en la función **p** tiene precedencia sobre la **declaración global de x** del comienzo del bloque. Por lo tanto, dentro de la función p, la variable global x no puede ser accedida. Se dice que **la variable global x tiene una apertura en el alcance**. En este punto cabe agregar el concepto de visibilidad.

Visibilidad de una declaración:

La visibilidad incluye únicamente aquellas regiones de un programa donde las ligaduras de una declaración son aplicables.

En síntesis, en el ejemplo anterior la variable global x tiene alcance dentro de la función p, ya que la ligadura sigue existiendo, no obstante la variable global x está oculta.

ACTIVIDAD 5: Realice el seguimiento del siguiente algoritmo y analice el alcance de las distintas declaraciones.

```
int b=10;
void mostrar(int x, int y)
{
    printf ("\n en este ejercicio hemos sumado %d + %d y obtuvimos este resultado:%d",x,y,x + y);
    printf ("\n Valor de b :%d",b);
}

void main(void)
{
    int a=10,b=5;
    printf ("\n a + b antes del bloque %d", a+b);
    {
        int j=5,a=3;
        printf ("\n a + b dentro del bloque %d", a+b);
        j++;
        printf ("\n j dentro del bloque %d", j);
    }
    printf ("\n a + b al salir del bloque %d", a+b);
    mostrar(a,b);
    getchar();
}
```

Pasaje de parámetros a una función

Un programa se comunica con sus funciones a través de los parámetros. Existen distintas formas de pasar parámetros a una función.

Pasaje por valor

Este tipo de pasaje es uno de los más comunes. En este mecanismo los parámetros reales se evalúan al momento de la llamada a la función y se transforman en los valores que toman los parámetros formales durante la ejecución de la función.

El paso por valor es el mecanismo por omisión de lenguajes como C++ y Pascal, siendo el único mecanismo de pasaje de parámetros de C y Java. En todos estos lenguajes **los parámetros formales se consideran como variables locales** que toman como valor inicial el valor de los parámetros reales.

Los parámetros formales pueden cambiar sus valores a través de asignaciones sin que estos cambios afecten los valores de los parámetros reales.

La desventaja es que al pasar un parámetro por valor, se produce una duplicación del área de memoria.

Ejemplo

```
int factorial ( int x )
{
    int f=1;
    while (x)
    {
        f*=x;
        x--;
    }
    return f;
}
```

```

int main (void )
{ int a;
  int fac;
  printf (" \n Ingrese un valor para a: ");
  scanf("%d", &a);
  fac=factorial(a);
  printf (" \n El factorial de %d es %ld", a,fac);
  getch();
}

```

Salida:

Ingrese un valor para **a**: **9**

El factorial de **9** es **362880**

Estos resultados muestran que dentro de main, **a** no se ha modificado, aunque se haya modificado el valor correspondiente a **x** en la función factorial.

La ventaja del pasaje por valor es que se protege el valor del parámetro real de posibles alteraciones dentro de una función.

Como puede inferirse, el pasaje por valor ofrece la posibilidad de proporcionar como parámetro real una expresión o valor constante, en lugar de que éste sea necesariamente una variable.

La figura 3 muestra la organización de la memoria durante la ejecución de la función factorial anterior.

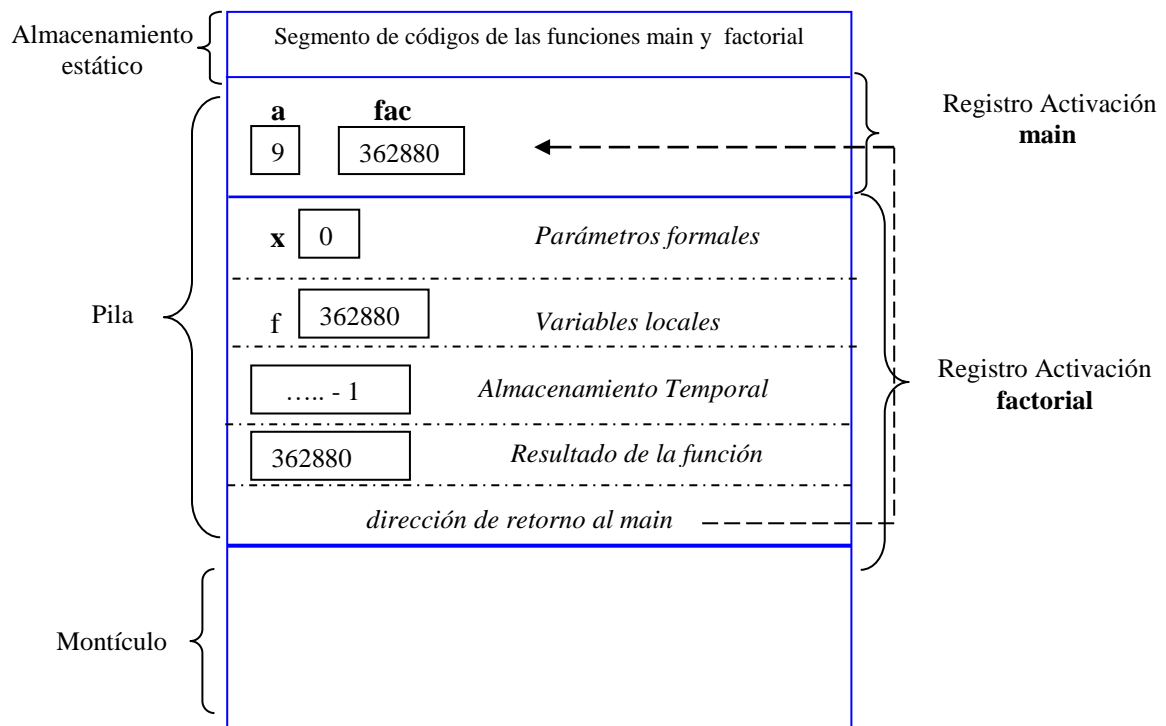


Fig. 3 Organización de la memoria antes de terminar la ejecución de la función factorial (pasaje por valor)

Nótese en este algoritmo que no hace falta definir una variable para la iteración, en su lugar se va decrementando el parámetro formal **x** hasta llegar a 1.

Pasaje de direcciones

El paso de pasajes de parámetros por valor no implica que no puedan ocurrir cambios fuera de la función. Si el parámetro es un puntero, esto es una dirección, puede usarse para cambiar la memoria apuntada por fuera de la función. Otra ventaja que ofrece el pasaje de parámetros por dirección es retornar más de un resultado desde la función.

En C, para pasar un parámetro por dirección, se utiliza el operador de dirección **&**, al momento de invocar a la función. Luego, el operador de indirección ***** debe utilizarse en la función para acceder al valor almacenado en esa dirección.

Retomemos la función factorial, pero de tal modo que la variable **a** se pasa por dirección: en los parámetros formales.

Ejemplo

```
int factorial ( int * x )
{
    int f=1;
    while (*x)
    {
        f*=*x;
        --(*x);
    }
    return f;
}
```

```
int main (void )
{
    int a;
    long int fac;
    printf (" \n Ingrese un valor para a: ");
    scanf("%d", &a);
    printf("Valor de a antes de invocar a la función factorial %d\n \n ", a);
    fac=factorial(&a);
    printf("Valor de a después de invocar a la función factorial %d\n \n ", a);
    printf (" \n El factorial es %ld", fac);
    getch();
}
```

Salida:

Ingrese un valor para a: **9**

Valor de a antes de invocar a la función factorial **9**

Valor de **a** después de invocar a la función factorial **0**

El factorial es **362880**

Como el operador **&** permite obtener la dirección de una variable, al definir la función factorial se debe indicar que el parámetro es un puntero y acceder indirectamente a través de éste a la variable **a**.

Más adelante, cuando se analiza el pasaje de arreglos como parámetros, se vuelve a trabajar con pasaje de direcciones.

La figura 4 muestra la organización de la memoria durante la ejecución de la función factorial anterior.

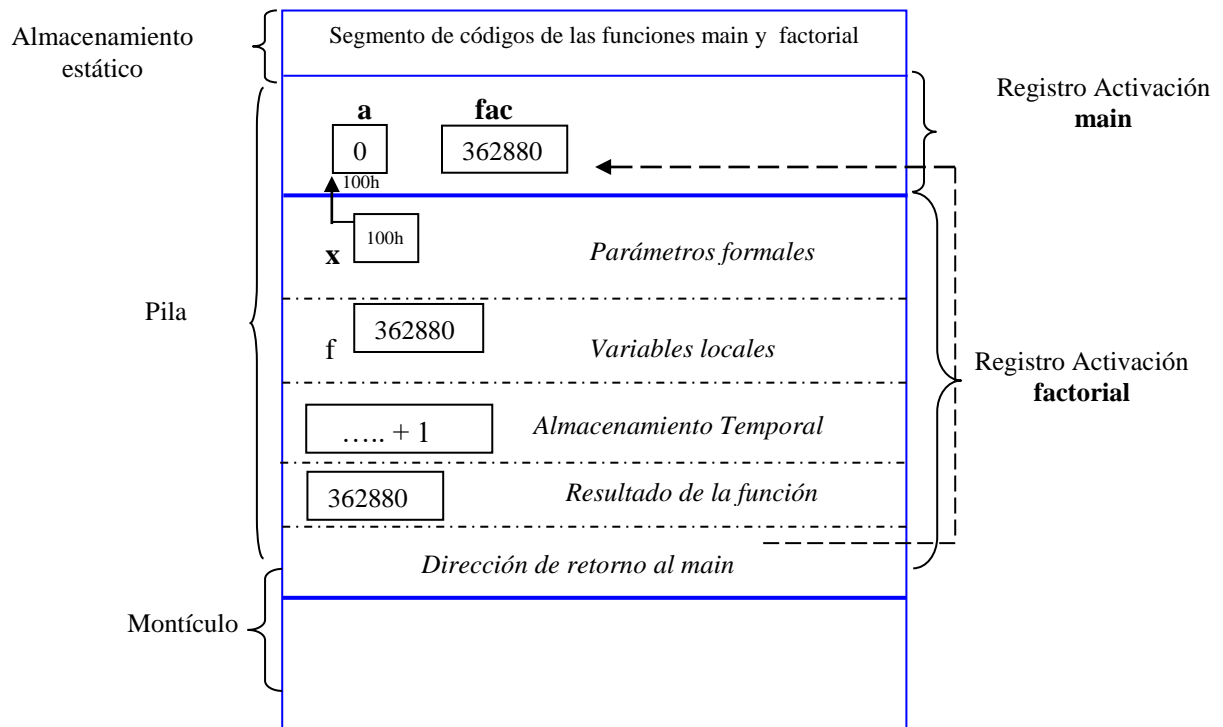


Fig. 4 Organización de la memoria antes de terminar la ejecución de la función factorial (pasaje de dirección)

Pasaje por referencias

Para realizar un pasaje por referencia el argumento a pasar debe ser una variable con una dirección asignada (si el paso fuese por valor el argumento puede ser una variable, una expresión o constante).

Esto es así por cuanto el pasaje por referencia no pasa la ubicación de la variable, por lo que el parámetro formal se transforma en un alias del parámetro actual de modo que cualquier cambio que se realiza en el parámetro formal se siente en el parámetro actual.

Esto puede interpretarse como que a una misma área de memoria se asignan dos nombres distintos.

Lenguajes como C++ y Pascal permiten el uso de pasaje por referencias.

En el caso de C++ , para indicar este tipo de pasajes se debe colocar el operador & a continuación del tipo de datos del parámetro formal.

Ejemplo : La función factorial antes definida, puede redefinirse usando pasaje por referencias:

```
int factorial ( int &x )
```

```
{ int f=1;
  while (x)
  { f*=x;
    x--;
  }
  return f;
}
```

```

void main (void)
{
    int a;
    int fac;
    printf (" \n Ingrese un valor para a: ");
    scanf("%d", &a);
    printf("Valor de a antes de invocar a la función factorial %d\n \n ", a);
    fac=factorial(a);
    printf("Valor de a despues de invocar a la función factorial %d\n \n ", a);
    printf (" \n El factorial es %ld", fac);
    getch();
}

```

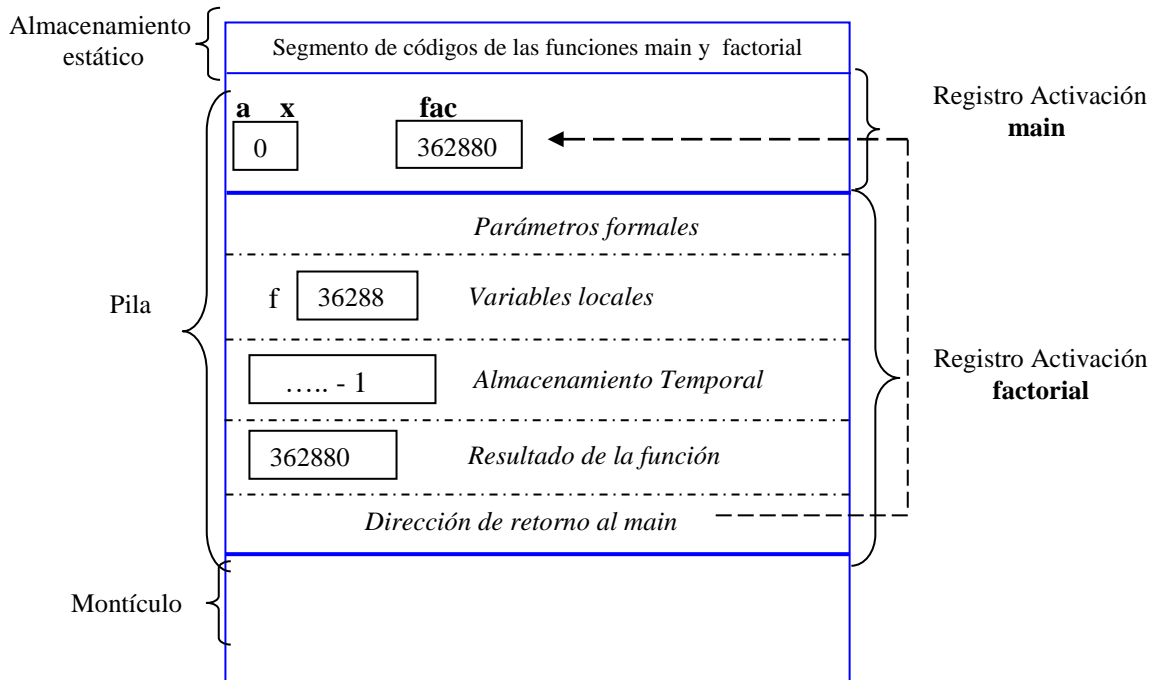


Fig. 5 Organización de la memoria antes de terminar la ejecución de la función factorial (pasaje por referencia)

Salida:

Ingrese un valor para a: **9**

Valor de a antes de invocar a la función factorial **9**

Valor de **a** después de invocar a la función factorial **0**

El factorial es **362880**

En este caso, **x** hace referencia a la variable **a**. Esto implica que no hay duplicación de memoria y cualquier modificación que se efectúe en el parámetro formal **x** afectará o se verá reflejado en el parámetro real **a**.

Observación: En lenguaje C estándar todas las llamadas son por valor, las llamadas por referencia se las puede simular haciendo pasaje por dirección.

No obstante en este curso se usará también los pasajes por referencias provistos por C++.

Funciones que devuelven más de un resultado

Recordemos que una función devuelve cero o un resultado. Hay situaciones en las cuales es necesario que la función devuelva más de un resultado.

Por ahora, la forma de lograr resolver este problema, es a través del pasaje de parámetros por direcciones o referencias.



Ejemplo

Supongamos que se necesita definir una función que retorna la suma de los números pares y la suma de los números impares, menores que un valor ingresado por teclado.

En este caso, uno de los resultados será devuelto por la función y el otro en un parámetro, a través de un pasaje de referencias.

```
int acumula_pares_impares (int &si, int xnum)
{
    int sp=0;
    while (xnum)
    {
        if ((xnum % 2)==0)
            sp+=xnum;
        else
            si+=xnum;
        xnum--;
    }
    return sp;
}

void main (void)
{
    int sumai=0, num;
    printf("ingrese un número \n"); scanf("%d", &num);
    printf(" suma de pares= %d \n", acumula_pares_impares (sumai, num));
    printf(" suma de impares= %d \n", sumai );
    getch();
}
```

En este ejemplo, sumai es una variable del main que almacenará la suma de los impares, para ello se pasa por referencia.

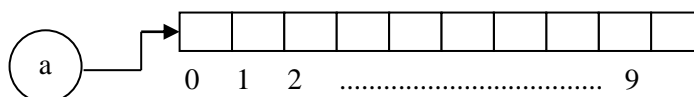
ACTIVIDAD 6

- Modificar el programa anterior de manera que el pasaje de la variable sumai se realice por dirección. ¿Qué conclusiones se pueden inferir si compara ambos tipos de pasajes?
- Modificar el programa anterior de manera que los dos resultados sean devueltos por la función a través de los parámetros.

Arreglos como parámetros de funciones

Como se sabe, el nombre de un arreglo es una *dirección constante* que apunta a la primera componente del mismo.

Para el arreglo **int a[10]**; la siguiente gráfica permite entender lo expuesto.



Por lo tanto, al pasar un arreglo como parámetro de una función, no se pasan los valores de las componentes del mismo, sino la dirección de la primera componente del arreglo. El parámetro formal se transforma entonces en un puntero al primer elemento.

De ahí, si se realiza alguna modificación a cualquier componente del arreglo, esa modificación será reconocida en todo el ámbito del arreglo.

Los prototipos de funciones que incluyen arreglos como argumentos, se pueden especificar de distintas formas:

- Colocando el nombre del arreglo en cuestión seguido de un par de corchetes vacíos o con el tamaño respectivo; todo esto precedido por el tipo de datos de las componentes del arreglo.
- Prescindiendo del nombre del arreglo, pero colocando un par de corchetes vacíos o indicando el tamaño del arreglo, precedidos por el tipo de datos de las componentes del arreglo.

Ejemplo

El siguiente programa muestra la forma de trabajar los arreglos como parámetros de funciones y las distintas formas de especificar el prototipo de las funciones.

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

```
#define n 20
```

```
void carga( float arre[n] )           /*carga del arreglo*/
```

```
{ int i;
  for (i=0; i < n ; i++)
  { printf("ingrese valor \n");
    scanf(" %f", &arre [i]);
  }
  return ;
}
```

```
void main (void )
```

```
{
float datos [n];
carga(datos);      //pasaje del nombre de un arreglo como parámetro
:
}
```

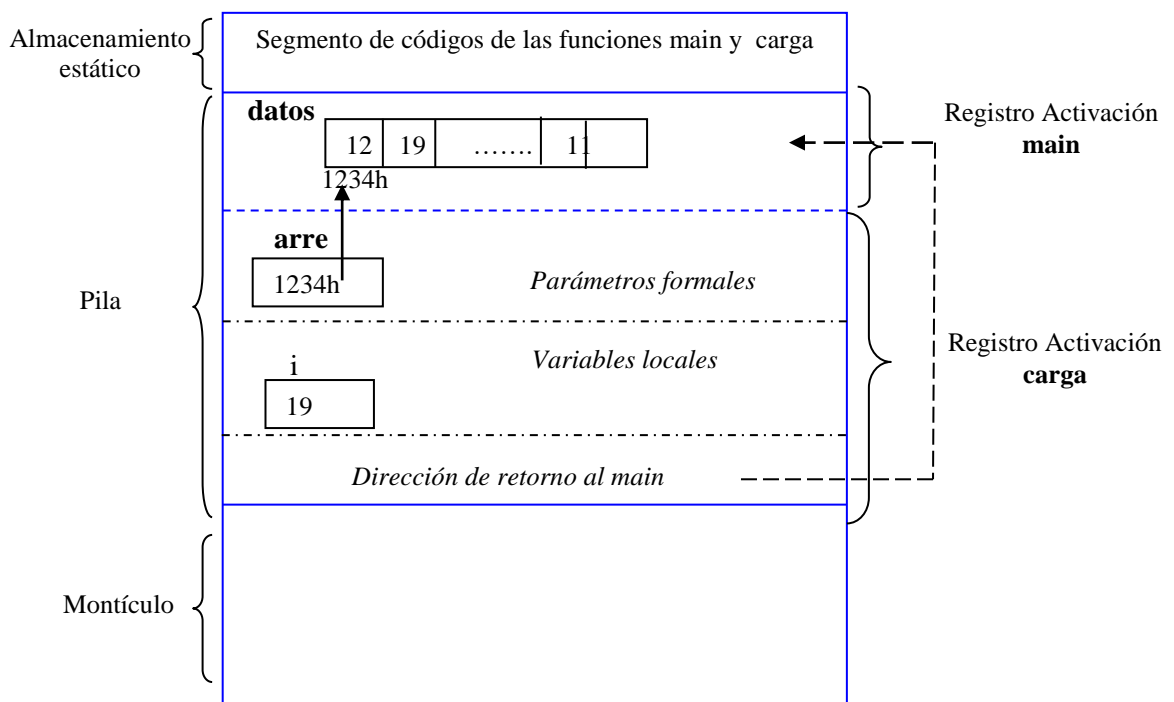


Fig. 6 Organización de la memoria antes de terminar la ejecución de la función carga

Observación: Como puede observarse, cuando se pasa un arreglo a una función se realiza un pasaje por valor de una dirección, la dirección de la primer componente del arreglo.

Otras formas de definir la función carga

a) Una buena práctica al momento de usar arreglos es enviar como parámetro no solo el nombre, sino también el tamaño.

```
void carga( float arre[], int n )           /*carga del arreglo*/
{ int i;
  for (i=0; i < n ; i++ )
  {   printf("ingrese valor \n");
      scanf(" %f", &arre [i]);
  }
  return ;
}
```

```
void mostrar( float arre[], int lim )
{ int i;
  for (i=0; i < lim ; i++ )
      printf("\n %f", arre[i]);
}
```

```
void main (void )
{
  float datos[5];
  float precios[30];
  carga(datos,5);
  mostrar(datos, 5);
  carga(precios, 30);
  .....
  getch();
}
```

En este ejemplo puede verse claramente la ventaja de enviar el tamaño del arreglo como parámetro.

b) Además, como el nombre del arreglo es un puntero, puede recibirse como un puntero y el tamaño del arreglo.

```
void carga( float * arre, int n )          /*carga del arreglo*/
{ int i;
  printf("\n Ingrese las %d componentes del arreglo \n", n );
  for (i=0; i < n ; i++ )
  {   printf("ingrese valor \n");
      scanf(" %f", &arre [i]);
  }
  return ;
}
```

```
void mostrar( float * arre, int lim )
{ int i;
  for (i=0; i < lim ; i++ )
      printf("\n %f", arre[i]);
}
```

```
void main (void )
{
    float datos[5];           // define el arreglo datos
    float arreglo[3];        // define el arreglo de nombre arreglo
    carga(datos,5);
    carga(arreglo,3);
    printf("\n datos del primer arreglo\n ");
    mostrar(datos, 5);
    printf("\n datos del segundo arreglo\n ");
    mostrar(arreglo, 3);
    getch();
}
```

ACTIVIDAD 7

Realizar un programa en C que, a través del uso de funciones, permita:

- Cargar un arreglo de N componentes enteras.
- Dado un valor entero, indicar si ese valor está o no en el arreglo. Si el valor está en el arreglo devolver la posición, de lo contrario devolver -1.
- Mostrar en el programa principal la cantidad de componentes pares del arreglo y la suma de las mismas.

Pasaje constante en arreglos

Dado que los arreglos son siempre pasados a las funciones a través del pasaje de una dirección, resulta difícil controlar las modificaciones de sus componentes. Para evitar estas modificaciones, C proporciona un calificador especial de tipo **const**.

En lenguaje C, los parámetros usados por una función pueden declararse como constantes (**const**) al momento de la declaración de la función. Un parámetro que ha sido declarado como constante significa que la función no podrá cambiar el valor del mismo (sin importar si dicho parámetro se recibe por valor o por referencia).

Ejemplo :

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

```
void intenta_modificar ( const int [ ] );
```

```
void main (void )
{
    int a [ ] = { 10 , 20 , 30 };
    intenta_modificar ( a );
    printf ( " %d %d %d \n", a [0], a [1], a [2 ] );
}
```

```
void intenta_modificar ( const int b [ ] )
{
    b [0] = 2;  /*error*/
    b [1] = 2;  /*error*/
    b [2] = 2;  /*error*/
}
```

Cada uno de los tres intentos por modificar las componentes del arreglo, en la función da como resultado el siguiente error de compilación: **cannot modify a const object**

Traducción y Tabla de Símbolos en Lenguaje C²³

La tabla de símbolos es una estructura de datos que se crea en tiempo de traducción del programa fuente. Es como un diccionario de variables, debe darle apoyo a la inserción, búsqueda y cancelación de nombres (identificadores) con sus atributos asociados, representando las vinculaciones con las declaraciones.

Aunque su nombre parece indicar una estructuración en una tabla no es necesariamente ésta la única estructura de datos utilizada, también se emplean árboles, pilas, etc.

Los símbolos se guardan en la tabla con su nombre y una serie de atributos opcionales que dependerán del lenguaje y de los objetivos del procesador. Este conjunto atributos almacenados en la TS para un identificador determinado se define como registro de la tabla de símbolos (symbol-table record).

La lista siguiente de atributos no es necesaria para todos los compiladores, sin embargo cada uno de ellos se puede utilizar en la implementación de un compilador particular:

- Nombre de identificador.
- Dirección en tiempo de ejecución a partir de la cual se almacenará el identificador si es una variable. En el caso de funciones puede ser la dirección a partir de la cual se colocará el código de la función.
- Tipo del identificador. Si es una función, es el tipo que devuelve la función.
- Número de dimensiones del array, o número de miembros de una estructura o clase, o número de parámetros si se trata de una función.
- Tamaño máximo o rango de cada una de las dimensiones de los arrays, si tienen dimensión estática.
- Tipo y forma de acceso de cada uno de los miembros de las estructuras, uniones o clases. Tipo de cada uno de los parámetros de las funciones o procedimientos, etc.

Operaciones con la TS²⁴

Las dos operaciones que se llevan a cabo generalmente en las TS son la inserción y la búsqueda.

Inserción y búsqueda

En lenguaje C, la operación **de inserción** se realiza cuando se procesa una declaración, ya que una declaración es un descriptor inicial de los atributos de un identificador del programa fuente.

Si la TS está ordenada, es decir los nombres de las variables están por orden alfabético, entonces la operación de inserción llama a un procedimiento de búsqueda para encontrar el lugar donde colocar los atributos del identificador a insertar. En tales casos la inserción lleva tanto tiempo como la búsqueda.

Si la TS no está ordenada, la operación de inserción se simplifica mucho, aunque también la operación de búsqueda se complica ya que debe examinar toda la tabla.

En las operaciones de inserción se detectan los identificadores que ya han sido previamente declarados, emitiéndose, a su vez, el correspondiente mensaje de error.

Ejemplo en lenguaje C: *multiple declaration for 's'* si s ya estaba en la TS

En las operaciones de **búsqueda** se detectan los identificadores que no han sido declarados previamente emitiéndose el mensaje de error correspondiente.

Ejemplo en lenguaje C: *Undefined simbolo 'x,'* si x es una variable que desea usarse pero no se declaró.

Set y Reset

Otras operaciones realizadas sobre una tabla de símbolos SET (activar un bloque) y RESET (desactivar el bloque). La operación de set se utiliza cuando el compilador detecta el comienzo de un bloque o función en el cual se pueden declarar identificadores locales o automáticos. La operación complementaria reset, se utiliza cuando se detecta el final del bloque o función.

²³ Lenguajes de Programación Principios y Prácticas. Kenneth Loudon. Pág. 127 - 131

²⁴ Lenguajes y Sistemas Informáticos: Tabla de Símbolos. Universidad de Oviedo. Pág. 9 a 16; 37 – 41.

Como consecuencia del uso de bloques anidados, la organización más simple para soportar la Tabla de Símbolos en los lenguajes estructurados en bloques, es una **PILA** (LIFO Last Input First Output).

Existen tablas de símbolos con estructura de árbol implementadas en pilas, Tablas de símbolos con estructura hash implementadas en pilas, pero éstos temas son objeto de estudio en años superiores.

Para comprender el manejo de la TS en lenguaje C podemos considerar una tabla de símbolos como un conjunto de nombres, cada uno de los cuales tiene una pila de declaraciones asociados con ellos, de manera que la declaración en la parte superior de la pila es aquella cuyo alcance está actualmente activo.

A la entrada de un bloque todas las declaraciones se procesan y se agregan las vinculaciones correspondientes a las TS; a la salida de un bloque se eliminan todas las ligaduras proporcionadas por las declaraciones, restaurando cualquier vínculo anterior que pudiera haber existido.

Podemos ver que la tabla de símbolos cambiará conforme se avanza en la traducción, para reflejar las inserciones o eliminaciones de ligaduras dentro del programa que se está traduciendo.

Una manera clara para ver el funcionamiento de la tabla de símbolos en lenguaje C, es a través de los gráficos con que usualmente se representan las variables.

Ejemplo

```
1) int x;
2) char y;
3) float func1(int n)
4) { float x=10.50;
5)     x*=2;
6)     {
7)         .....
8)         int y[10];
9)     }
10) ...
11) }
12) void func2(void)
13) { int y;
14)     ...
15) }
16) void main(void)
17) { char x;
18)     ...
19) }
```

Los identificadores (nombres) involucrados en este programa son **x**, **y**, **n**, **func1**, **func2** y **main**, pero **x** e **y** están cada uno de ellos asociados con tres declaraciones diferentes y con alcances distintos.

Después que el procesamiento alcanzó la función **func1**, la tabla de símbolos en la **línea 4** es:

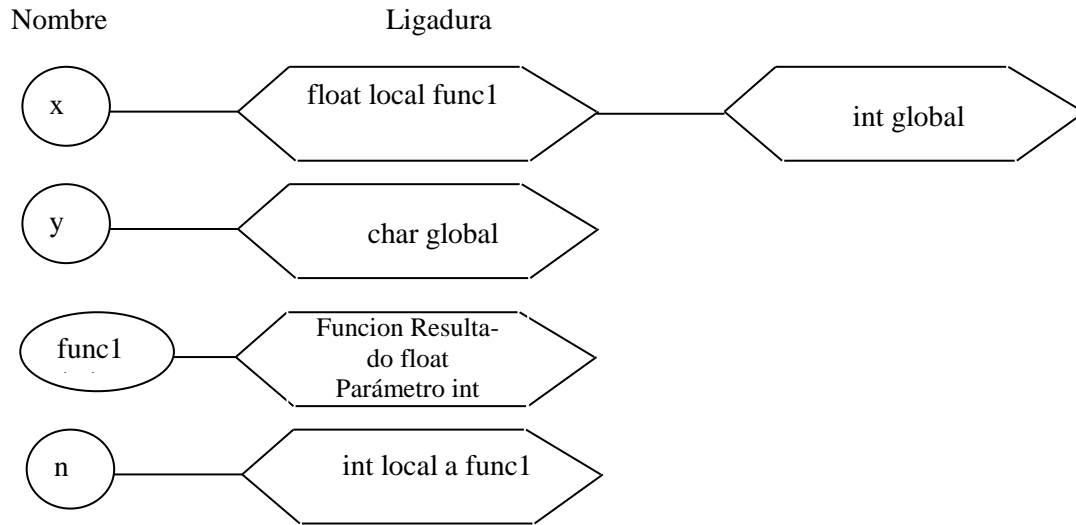


Tabla de Símbolos en la línea 4

Después del procesamiento de la declaración del bloque anidado en **func1**, con la declaración local de **y** en la línea 8, se tiene:

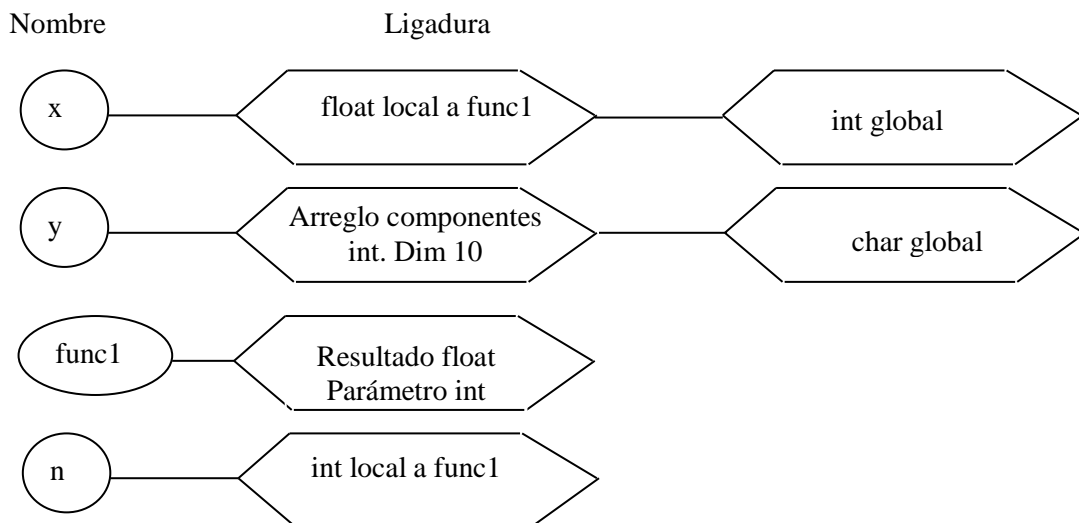


Tabla de símbolos en la línea 8

En la línea 10, la tabla de símbolos es la siguiente:

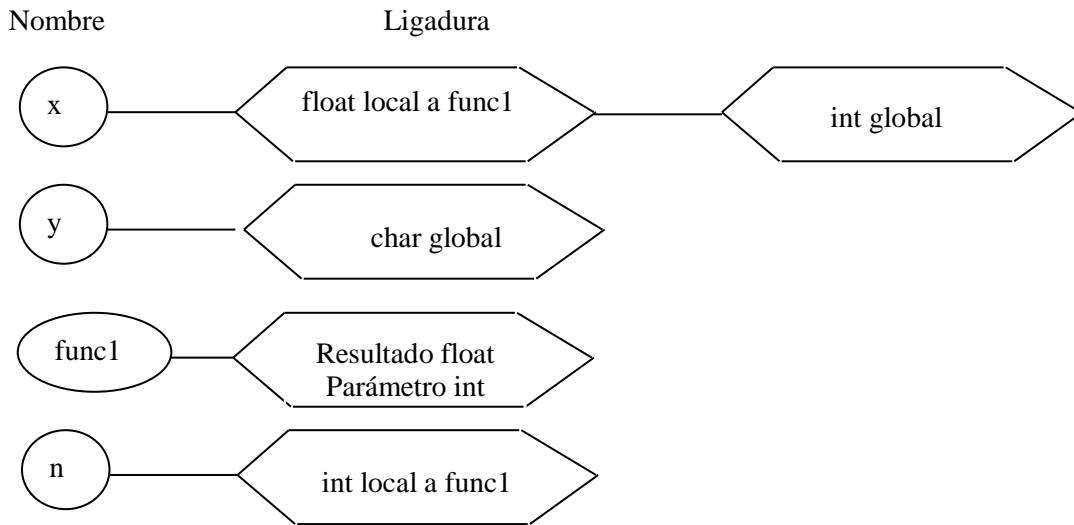


Tabla de símbolos en la línea 10

Una vez terminado el procesamiento de la función **func1** (línea 11), la tabla de símbolos cambia debido a las extracciones de las declaraciones locales de **n**, **x** e **y**.

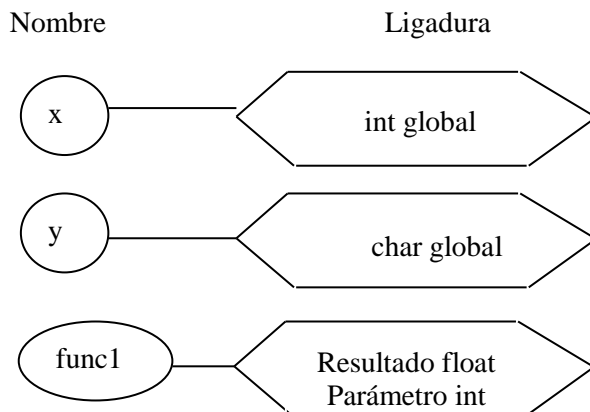


Tabla de símbolos en la línea 11

Una vez introducida la función **func2**, en la línea 13 la tabla de símbolos se convierte como en:

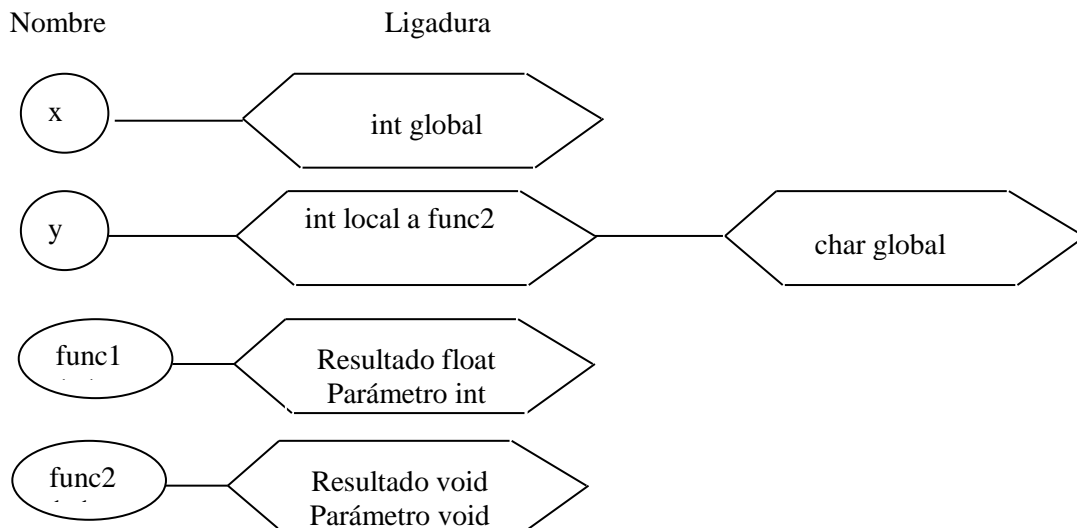


Tabla de símbolos en la línea 13

y cuando func2 termina (línea 15), la tabla de símbolos es:

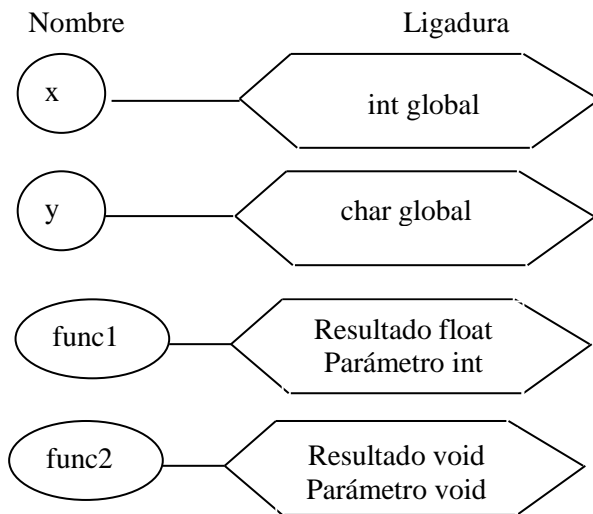


Tabla de símbolos cuando termina la línea 15

Finalmente, después de de iniciar la función **main** ,la tabla de símbolos en la línea 17 queda como se muestra:

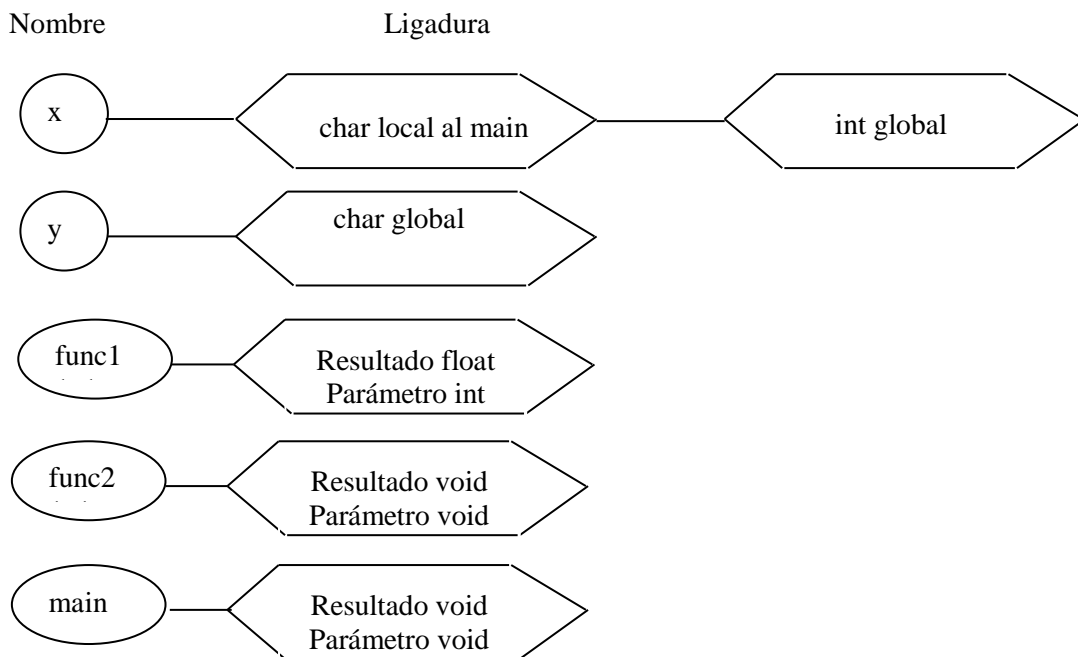


Tabla de símbolos en la línea 17

Observe que este proceso **conserva la información apropiada de alcance**, incluyendo las aperturas de alcance para la declaración global de **x** en el interior de **func1** y la declaración global de **y** en el interior de **func2**. Esta representación de la tabla de símbolos supone que la tabla procesa las declaraciones de manera estática, esto es, antes de su ejecución. Este es el caso, siempre que la tabla de símbolos sea manejada por un compilador y que las ligaduras de las declaraciones sean todas estáticas.

Si la tabla de símbolos está administrada de esta misma forma pero dinámicamente, esto es, durante su ejecución, entonces las declaraciones se procesan conforme se van encontrando a través del programa a lo largo de la trayectoria de ejecución. Esto da como resultado una regla de alcance diferente, lo que por lo general se conoce como alcance dinámico. La regla de alcance léxico anterior a veces se conoce como alcance estático.

Manejo de la TS en PILA

La organización más simple, desde el punto de vista conceptual, de una tabla de símbolos de un lenguaje estructurado en bloques es la pila. En este tipo de organización los registros que contienen los atributos de los identificadores se van colocando unos encima de otros según se van encontrando las declaraciones de las variables del texto fuente. Una vez que se lee el fin del bloque, todos los registros de los identificadores declarados en el bloque se sacan de la pila, pues dichos identificadores no tienen validez fuera del bloque.

Resulta importante entender el funcionamiento de la función SET y RESET. Para esto se usa una pila auxiliar, de tal modo que la operación SET guarda en esta pila un índice de bloque (BLOCK INDEX) que representa el lugar donde se encuentra almacenado el primer registro del bloque, y que al momento de la operación ocupa la parte superior (TOP) de la pila.

```
int x;
char y;
float func1(int n)
{ float x=10.50;
  x*=2;
  { .....
    int y[10]; /1*****/
    char z
  }
  .../2*****/
}
void func2(void)
{ int y;
  ...
}
void main(void)
{ char x; /3*****/
  ...
}
```

Volviendo al ejercicio anterior, en la línea marcada con /****/, la tabla de símbolo y la pila auxiliar tiene la siguiente información.

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1
4	n	int
5	x	float
6	y	Arreglo
7	Z	char

1
4
6

En la línea marcada con /2****/, la operación RESET saca de la pila todos los registros de las variables que se acaban de compilar en el bloque finalizado, para esto el BLOCK INDEX indica hasta que registro ha de sacar desde la parte superior de la tabla de símbolos.

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1
4	n	int
5	x	float

1
4

ACTIVIDAD 8: Muestre la tabla de símbolos y la pila auxiliar en la línea marcada con /3****/.

Bibliografía:

Louden, Kenneth (2004) Lenguajes de Programación Principios y Práctica. Editorial Thomson. Mexico.

Universidad de Oviedo (2006) Lenguajes y sistemas informáticos. Tablas de símbolos.

Deitel, Harvey M. y Deitel, Paul J. (2003) Como programar en C, C++ . Pearson Educación. Cuarta Edición. Buenos Aires.

Programación en C. Byron S. Gottfried. (1997): McGraw-Hill. Madrid - Buenos Aires.

Joyanes Aguilar, Luís (2001) Programación en C - Metodología, algoritmos y estructura de datos. McGraw-Hill. Madrid - Buenos Aires.

Documentos y Guías de trabajos Prácticos propuesto por el equipo de cátedra.

Practico 2

Lenguaje C - Funciones - Tabla de Símbolo

Ejercicio 1

A partir de un arreglo generado aleatoriamente con 50 números enteros, codificar un programa en C que permita:

- Indicar si alguno de los números generados es un cero.
- Escribir el contenido de las componentes que se encuentren en las posiciones pares.
- Indicar cantidad de números pares que contiene.
- Leer un número y si se encuentra en el arreglo indicar su posición (realizar búsqueda óptima).

Ejercicio 2

Un local comercial de ventas de repuestos de automotores desea obtener cierta información sobre todas las ventas registradas en un periodo de tiempo dado. El comercio cuenta con 250 artículos, almacenados en una estructura y de los cuales se conocen los siguientes datos: Código, Nombre, Precio Unitario y Stock.

Los datos ingresados de cada una de las ventas efectuadas son: Nombre del artículo, Cantidad de unidades vendidas. El ingreso de ventas termina con nombre "FIN".

Se pide realizar un programa en C, que utilizando subprogramas óptimos y estructuras adecuadas permita:

- Procesar las ventas registradas en ese periodo de tiempo.
- Mostrar los nombres de aquellos artículos que quedaron con stock nulo.
- Indicar el stock de un artículo cuyo código se ingresa previamente por teclado.
- Imprimir los nombres de los 20 artículos que quedaron con mayor stock.
- Indicar el monto total obtenido por las ventas de los productos.

Ejercicio 3

Un laboratorio abastece a 30 farmacias de la provincia. Dicho laboratorio comercializa 80 medicamentos (1..80) de los que se debe registrar: Código de medicamento, nombre y precio unitario.

Se ingresan las ventas realizadas ordenada por farmacia. Por cada venta a una farmacia se ingresa: código de medicamento y cantidad de unidades, finalizando con código de medicamento igual a 0 (cero), como lo muestra el siguiente ejemplo:

	Código Medicamento	Cant. Unidades
Farmacia 1	23	12
	32	20
	41	6
	0	
Farmacia 2	43	10
	25	24
	0	

Codificar un programa en C, que utilizando funciones permita:

- Calcular y mostrar total de unidades vendidas de cada uno de los medicamentos.
- Escribir el/los códigos/s del/los medicamento/s por el que se recaudó mayor importe.
- Indicar la cantidad de unidades vendidas para un código de medicamento ingresado por teclado.
- Dado el nombre de un medicamento indicar el importe total recaudado.
- Indicar la cantidad de unidades vendida a cada farmacia y el importe total que pagó cada una.

Ejercicio 4

Una industria comercializa 70 productos codificados entre 100 y 169. De cada producto se conoce el código de producto y precio unitario. Además se cuenta con la información de las ventas realizadas durante el fin de semana. Por cada venta se ingresa código de producto y cantidad de unidades, finalizando el ingreso con código de producto igual a cero.

Se pide realizar un programa en C, que utilizando funciones óptimas y estructuras adecuadas permita:

- Total de unidades vendidas de cada uno de los productos.
- Código del producto que recaudó mayor importe.
- Dado un código de producto de producto, indicar la cantidad de unidades vendidas.
- En función del total de unidades vendidas, decir de cuantos productos se vendieron 20, 21, 22.. 50 unidades.

Ejercicio 5

En la Facultad se realiza un congreso para el cual se destinan 6 salas de conferencias y cada una representa un área temática. En cada sala se dictaran 4 conferencias en distintos turnos. Para procesar la información, en un primer momento y por única vez se ingresa el nombre de cada una de las 6 áreas temáticas que se tratarán en el congreso y el cupo de personas para la sala donde se dictará la misma. Por cada interesado se ingresa su nombre, nombre del área temática, y número correspondiente a la conferencia a la que quiere asistir. La inscripción se realiza previa verificación del cupo de la sala. A partir de la información ingresada generar una tabla que permita responder los siguientes ítems:

- Decir para cada área temática qué conferencia tuvo menos asistentes y cuál la mayor cantidad (Suponer el mayor y el menor como valores únicos).
- Indicar el nombre del área temática con menos inscriptos.
- Dado un nombre de área temática decir cuál fue el promedio de inscriptos.
- Indicar la/s áreas temáticas que en algún turno tuvieron la sala completa, si las hubiera.


Ejercicio 6


Un supermercado ingresa las ventas de los últimos 6 meses, realizadas en los 8 departamentos de venta que posee. Por cada venta se ingresa mes, departamento e importe. Las ventas no traen ningún orden particular. Realizar un programa en C, que a través de funciones permita:

- Almacenar la información en una tabla que posea por cada mes, el importe total de ventas de cada departamento.
- Mostrar en el programa principal el departamento que tuvo menor importe de venta (suponer único).
- Mostrar importe promedio de venta del supermercado.
- Mostrar el/los departamento/s que supera/n la venta promedio, indicando el importe total vendido a lo largo del semestre.

Ejercicio 7

Para el siguiente programa se pide:

- Mostrar la tabla de símbolos en las líneas con la marca  utilizando la pila auxiliar.
- En cada subprograma indicar variables locales y globales.
- Hacer el seguimiento de la ejecución y mostrar el estado de la memoria cuando se ejecuta la función calculo en la última invocación. Lote de prueba: (2,3) (0,1), (3, 5), (0, 1), (8,7) (3,3).
- Indicar variables automáticas, estáticas y externas según corresponda.

```
int m=0;
int calculo(int v, int w)
{ static int z=0; 
  if ((w==1)&& (v==0)) z++;
  return z;
}
```


```

void main(void)
{
    int a,b, y=0;
    printf("\n ingrese par de valores a y b");
    scanf("%d %d", &a, &b);
    while (a<b)
    {
        int c;
        c=a*b;
        calculo (a,b);
        x+=a;
        y+=b;
        printf("\n ingrese valor a y b");
        scanf("%d %d", &a, &b);
        printf("\n .....%d", c);
    }
    printf("\n .....%d", calculo(a,b));
    printf("\n .....%d .....%d", x,y);
}

```

Ejercicio 8

Para el siguiente programa se pide:

- Mostrar la tabla de símbolos en las líneas con la marca  indicando variables locales y globales.
- Hacer el seguimiento de la ejecución y mostrar el estado de la memoria cuando se ejecuta la función calculo1.
- Indicar variables automáticas, estáticas y externas según corresponda.

```

int e=10;
int resuelve (int v,int w, int *z)
{
    *z=(v + w)*e;
    return *z;
}

```

```

void calculo (int &x, int y)
{
    char e='M';
    while (y)
    {
        int k=y + 1;
        printf("valor de k antes de entrar en la función: %d", k);
        resuelve(x, y, &k) ;
        printf("valor de k: %d", k);
        y--;
    }
    printf("valor de e: %c", e);
}

```

```

void main(void)
{
    int n=2,m=3;
    {
        float e=3.2 ;
        printf("valor de e: %c", e);
    }
    printf("valor de e: %d", e);
    calculo (n,m);
}

```


Ejercicio 9

Una empresa de seguros procesa la información de las ventas que han realizado sus 10 promotores. De cada uno de los 10 promotores se conoce el código de sector donde trabaja (número entre 30 y 37) codificado: 30:Moto - 31:Auto - 32:Camioneta - 33:Camión - 34:Ómnibus de Corta distancia - 35:Ómnibus de larga distancia - 36:Combis de pasajeros - 37:taxis. De cada seguro (son 3 tipos de seguros distintos) se conoce el tipo (una letra entre "A" y "C"), el nombre y su precio. Los tipos de seguro se codifican: "A": Seguro contra terceros, "B": Seguro de Incendio y "C": Seguro Total.

Nota: Leer la información que se pide, y de acuerdo a eso, ¿Qué estructura es la más adecuada para el almacenamiento de los datos?

Se pide realizar un programa que permita (utilizando Menú de opciones):

- Ingresar las ventas de seguros realizadas. Por cada venta se ingresa número de promotor (de 1...10) y tipo de seguro("A"... "C"). Las ventas no traen ningún orden específico y termina el ingreso con número de promotor igual a 0.
- Ingresar un tipo de seguro e indicar en qué sector se lo vende más y cuantos promotores tiene ese sector.
- Dado un número de sector, indicar cuál es el seguro que más se consume.
- Indicar para cada tipo de seguro, el nombre y el importe total de venta.

Ejercicio 10

Una fábrica de ropa comercializa 50 prendas que son vendidas a 35 comercios del país. Por cada venta realizada se cuenta con los siguientes datos: Código de comercio (60..94), Nombre de la prenda vendida y cantidad de unidades. Las ventas no traen ningún orden en particular.

En una estructura se registra por cada prenda que se comercializa su nombre y precio unitario, ordenado alfabéticamente.

Además por cada comercio se almacena su CUIL y Nombre.

Se pide realizar un programa en C, que utilizando funciones óptimas y estructuras adecuadas permita (utilizar Menú de opciones):

- Almacenar los datos de las ventas en una estructura que posea por cada comercio la cantidad de unidades vendidas de cada prenda.
- Indicar por cada comercio; CUIL, Nombre e importe total a pagar.
- Realizar un listado que contenga por cada producto, nombre y cantidad de unidades vendidas, este listado debe estar ordenado descendientemente por cantidad de unidades.
- Mostrar el nombre de los 5 productos que más se vendieron.

Ejercicios Propuestos**Ejercicio 1**

Codificar en C un programa que tenga:

- Un menú de opciones, una de las cuales debe ser secreta. Propuesta:
 - Opción 1
 - Opción 2
 - Opción 3
 99. Opción Secreta
- Crear una función que reciba tres valores enteros ingresados por el usuario, y que calcule el cuadrado de cada número. El pasaje de los parámetros debe ser por valor, por referencia y por dirección. Mostrar los valores de las variables antes del llamado a la función, dentro de la misma y al salir. Esta función debe ser la Opción 1 del menú.
- Mostrar mapa de memoria de la ejecución de la función del punto anterior.
- Crear una función que genere una tabla de NxM componentes enteras, y lo llene con números aleatorios entre 100 y 199. (Opción 2)

Ayuda: <https://blog.martincruz.me/2012/09/obtener-numeros-aleatorios-en-c-rand.html>

- e) Crear dos funciones que procese la tabla anterior. Una, que busque el máximo de la fila 0 (enviar sólo la fila a procesar); y la otra, que cuente todas las componentes de la columna M. (Opción 3)

1. ¿Se puede enviar como parámetro una sola columna de la tabla? Justifique
2. A nivel de memoria, ¿Qué diferencia hay entre mandar una fila o la tabla completa? ¿Cuál forma es la más óptima?

Nota: Responder usando mapa de memoria

Ejercicio 3

Realizar un programa en C que permita:

- a) Leer datos enteros hasta que se llegue a 1000 datos o hasta que el dato ingresado sea igual a -50.
- b) Imprimir cantidad de elementos ingresados.
- c) Imprimir porcentaje de datos pares leídos.
- d) Calcular e imprimir promedio de todos los datos ingresados.

Ejercicio 4

Se tienen los datos relacionados a un censo de pacientes de un hospital. Por cada paciente se ingresa número de paciente, edad y peso. El ingreso finaliza cuando se lee un peso negativo o cuando la cantidad de pacientes supere los 400.

Realizar un programa en C, que permita:

- a) Calcular la cantidad de pacientes cuya edad este comprendida entre 7 y 11 años inclusive.
- b) Determinar el porcentaje de pacientes mayores de 11 años cuyo peso no supera los 50kg.
- c) Imprimir el número de paciente y edad con menor peso.

Ejercicio 5

Una empresa desea realizar un control de 700 productos distintos que comercializa, de los mismos posee: Código de identificación del producto, cantidad de productos con ese código y precio unitario.

Realizar un programa en C que permita:

- a) Almacenar los datos de todos los productos.
- b) Calcular el monto total del stock por producto (precio por cantidad).
- c) Mostrar el código de identificación de los productos con mayor precio unitario y cuyo código de identificación esté comprendido entre 250 y 300.
- d) Ingresar un valor correspondiente a una cantidad de stock mínima y generar un arreglo que contenga los códigos de aquellos productos cuya cantidad sea menor que la ingresada.

Ejercicio 6

Se tienen las notas de 10 materias de 25 alumnos que cursan 3er año de secundaria. Además se cuenta con los nombres de los alumnos almacenados en un arreglo.

Realizar un programa en C, que permita:

- a) Cargar y mostrar por cada alumno la calificación obtenida en cada materia.
- b) Calcular la nota promedio por cada alumno.
- c) Calcular la nota máxima y mínima en cada materia.
- d) Imprimir el nombre del alumno con mejor promedio.

Ejercicio 7

Realizar un programa en C, que permita:

- a) Cargar una matriz de $n \times m$, con elementos enteros desde el 1 al 100.

Buscar los números primos que contenga la matriz e indicar la posición en la que se encuentra cada uno de ellos.

Unidad 4: Recursividad

Introducción

La recursividad es una alternativa a la iteración, es un proceso mediante el cual se puede definir una función en términos de sí misma.

Generalmente, la solución recursiva a un problema planteado suele resultar menos eficiente en cuanto al tiempo de ejecución y al almacenamiento en memoria. A pesar de esta dificultad, en muchos casos si el algoritmo está definido recursivamente; se elige la solución recursiva por ser más simple y natural que la solución iterativa correspondiente, permitiendo así, reducir su complejidad y ocultar detalles de programación.

Funciones Recursivas

El factorial de un número natural, está definido iterativamente de la siguiente manera:

$$y \begin{cases} n! = n * (n-1) * (n-2) * \dots * 2 * 1, & \text{si } n > 0 \\ n! = 1, & \text{si } n = 0 \end{cases}$$

Ese mismo cálculo puede expresarse de manera recursiva, así:

$$y \begin{cases} n! = n * (n-1)!, & \text{si } n > 0 \\ n! = 1, & \text{si } n = 0 \end{cases}$$

A continuación, se presenta en lenguaje C la solución recursiva:

```
#include <stdio.h>
int factorial (int i); /* declaración de la función factorial */
int main ()
{
    int n;
    printf ("\n n = ");
    scanf ("%d", n);
    printf ("\n n! = %d\n", factorial ( n ));
}

int factorial (int i )
{
    if ( i == 0 )
        return ( 1 )
    else
        return ( i * factorial ( i - 1 ) );
}
```

Puede observarse que la función factorial se llama así misma de manera recursiva, con un argumento real (i - 1), que decrece en cada llamada recursiva. Estas llamadas recursivas finalizan cuando el parámetro real toma el valor 0.

Cuando se ejecute este programa, se invocará a la función factorial sucesivamente, una vez desde main y (i - 1) veces desde dentro de sí misma, aunque no sea advertido por el programador.



Al construir una función recursiva, se debe asegurar la existencia de:

Un caso base (puede haber más de uno), que permite detener la invocación sucesiva de la función; caso contrario se tendrían una serie infinita de invocaciones sucesivas.

Uno o más casos generales, que permiten que la función se invoque a sí misma con valores de parámetros que cambian en cada llamada; acercándose cada vez más al caso base.

Cuando se ejecuta una función recursiva, las sentencias que aparecen después de cada invocación no se resuelven inmediatamente, éstas quedan pendientes. Por cada invocación recursiva, se genera y se apila en el stack o pila un registro de activación. Luego, esos registros de activación se van desapilando en el orden inverso a como fueron apilados, ejecutándose además aquellas sentencias que quedaron pendientes.

Retomando el ejemplo de la función factorial recursiva, las llamadas sucesivas de la función serán:

```

n ! = n * ( n - 1 ) !
( n - 1 ) ! = ( n - 1 ) * ( n - 2 ) !
( n - 2 ) ! = ( n - 2 ) * ( n - 3 ) !
.....
.....
2 ! = 2 * 1 !
1 ! = 1 * 0 !
0 ! = 1

```

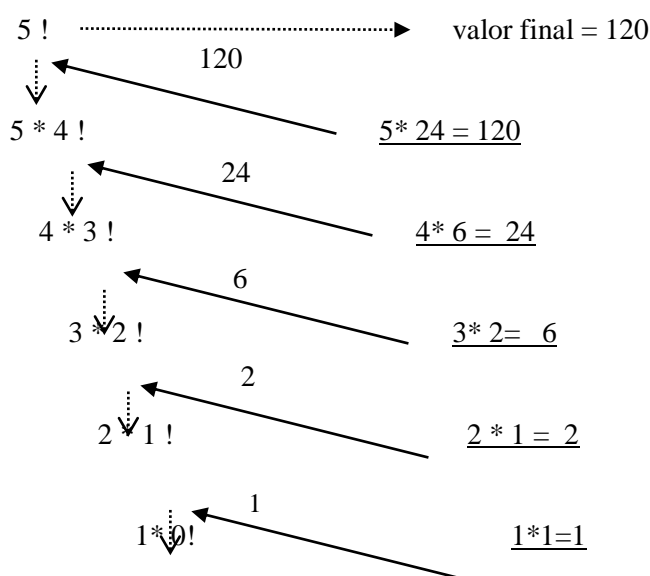
Luego, los valores de retorno se devolverán en el orden inverso:

```

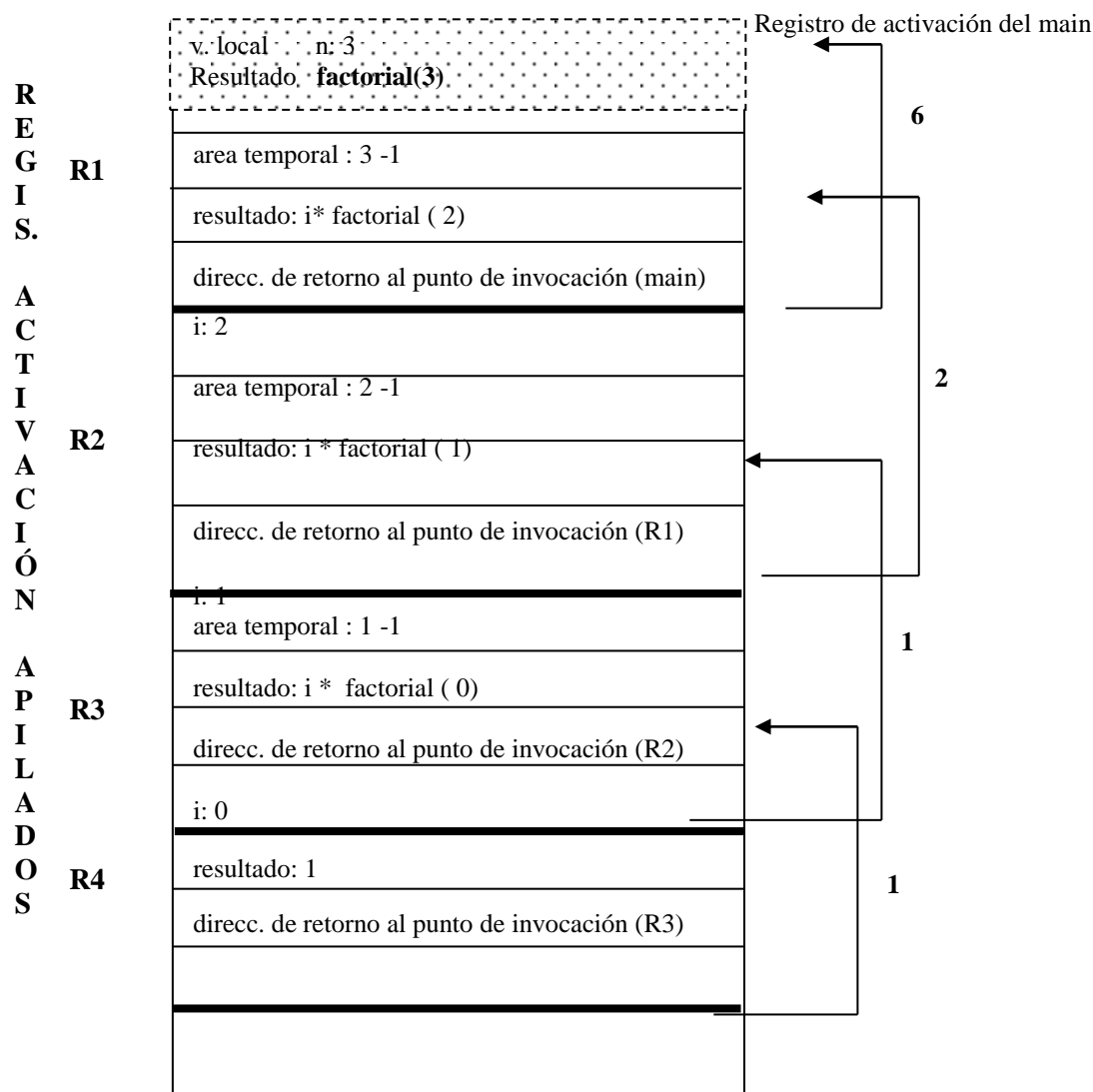
1 ! = 1
2 ! = 2 * 1 = 2
3 ! = 3 * 2 = 6
4 ! = 4 * 6 = 24
.....
.....
n ! = n * ( n - 1 ) !

```

Por ejemplo, al calcular $5!$, se realizan sucesivas llamadas recursivas y finalmente, al encontrar el caso base, se retornan los valores obtenidos en cada llamada:



El orden inverso de ejecución es una característica típica de todos los algoritmos recursivos. Gráficamente, lo que sucede internamente en la memoria al invocar a la función factorial, por ejemplo para $n = 3$, es:



Al invocar a **factorial (3)**, se apila el primer registro de activación **R1**; en el cual se guarda el *parámetro actual* $n = 3$, la *dirección de retorno*, el *resultado de la función* y también un *área temporal de almacenamiento* para los cálculos intermedios. Como en el cálculo de factorial (3) está involucrado el cálculo de **factorial(2)**, se apila el registro de activación **R2**, que almacena el parámetro actual $n = 2$, el punto de retorno, el resultado y un área temporal de almacenamiento para los cálculos intermedios.

Como en el cálculo de **factorial(2)** está involucrado el cálculo de **factorial(1)**, se apila el registro de activación **R3**, que almacena el parámetro actual $n = 1$, el punto de retorno, el resultado y un área temporal de almacenamiento para los cálculos intermedios.

Del mismo modo, como en el cálculo de factorial (1) está involucrado el cálculo de **factorial(0)**, se apila el registro de activación **R4**, que almacena el parámetro actual $n = 0$, el punto de retorno y el resultado (en este caso no hace cálculos intermedios). Finalmente, al llegar al caso base, este resultado se transfiere al registro de activación **R3**, retornando al punto de invocación; desapilándose **R4**.

El resultado obtenido en **R3**, se transfiere al punto de invocación en **R2**, desapilándose **R3**.

El resultado obtenido en **R2**, se transfiere al punto de invocación de **R1**, desapilándose **R2**.

El resultado obtenido en **R1**, retorna al punto del programa desde donde fue invocada factorial(3); desapilándose **R1**.

Al invocar una función recursiva, se apilan sucesivamente cada uno de los registros de activación, correspondientes a las distintas invocaciones. Al llegar al caso base, comienzan a desapilarse cada uno de esos registros de activación, resolviendo tareas pendientes, si es que existen.

Cabe aclarar que al ejecutar una función recursiva, ésta genera un número considerable de auto invocaciones; se corre el riesgo de ocupar gran cantidad de memoria en la pila.

Si una función recursiva posee variables locales, se creará un conjunto diferente de variables por cada llamada. Obviamente, los nombres de esas variables locales serán siempre los mismos; sin embargo, las variables representarán un conjunto diferente de valores cada vez que se invoque la función.

Tanto la recursión como la iteración son lógicamente equivalentes; esto es, todo algoritmo recursivo puede escribirse de manera iterativa y viceversa.

Generalmente, la recursividad presenta mayor grado de simplicidad y síntesis, al momento de codificar. La desventaja se presenta normalmente en tiempo de ejecución, debido a la cantidad de memoria utilizada.

ACTIVIDAD 1: Completar el programa con una función recursiva “divisor” que recibe dos números naturales distintos a y b, $a > b$. La función devuelve un valor 1 al programa principal para que se escriba un mensaje diciendo que b es divisor de a, caso contrario cuando b no es divisor de b devuelve cero(0).

Hacer el mapa de memoria para cada una de las soluciones planteadas con el lote de prueba $x=7$, $y=2$.

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

```
int divisor (int x, int y) //completar
{
    ----
    ----
    ----
}
```

```
main(void)
{
    int a,b,r;
    printf("Ingrese dos números naturales distintos el primero mayor que el segundo\n");
    scanf("%d %d",&a, &b);
    if (divisor(a,b)==1)
        printf("%d es divisor de %d", a,b);
    else
        printf("no hay divisores");
    getch();
}
```

ACTIVIDAD 2: Para el siguiente programa:

- Defina una función “Binario” que transforme un número natural en el número binario correspondiente.
- Graficar el mapa de memoria al invocar a Binario (9).
- ¿Puede generalizar el algoritmo para sistemas de base 3, 6, etc.? ¿Qué modificaciones deberían hacerse?

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

void Binario(int n) //completar
{
    -----
    -----
}

void main()
{
    int n;
    printf("\n Introduzca un entero positivo: ");
    scanf("%d", &n);
    printf("\n El Numero Decimal %d en binario es ", n);
    Binario(n);
    getch();
}
```

ACTIVIDAD 3

Teniendo en cuenta la siguiente definición de una función recursiva:

```
void listar ( int n)
{
    if ( n != 0 )
    { printf ( " %d", n);
      listar ( n - 1 );
      printf ( " %d", n);
    }
    else
        printf ( " el número es cero");
    return;
}
```

- 1) Construir en C, el programa que pueda invocarla.
- 2) Realizar el seguimiento
- 3) Graficar el mapa de memoria al invocar a **listar (4)**.

Ejemplo : La siguiente función calcula, recursivamente, la suma de los números pares menores o iguales que un valor ingresado por teclado.

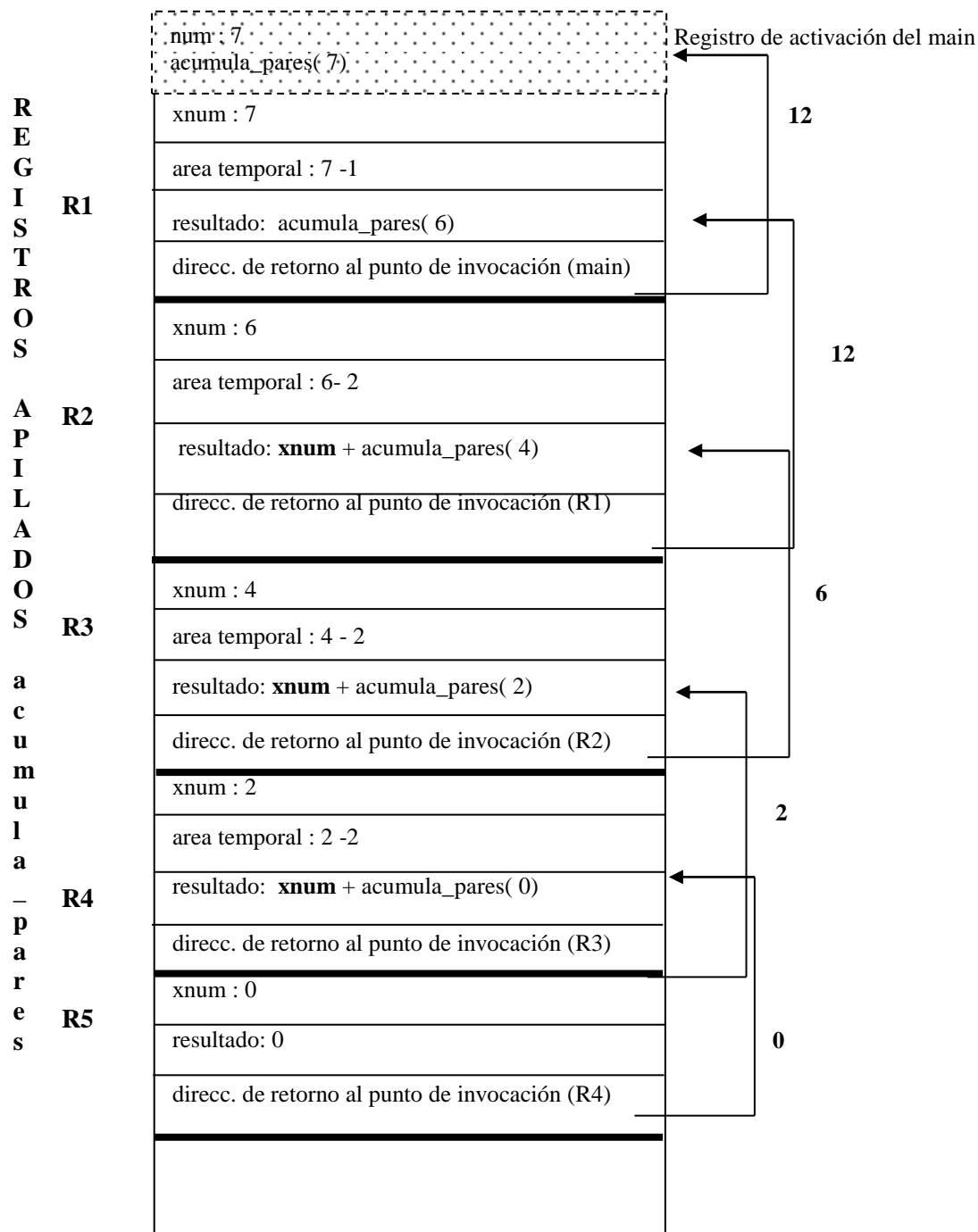
```
#include <stdio.h>
#include <conio.h>
int acumula_pares ( int xnum)
{
    if (xnum)
        if ((xnum % 2) == 0)
            return xnum + acumula_pares( xnum - 2);
        else return(acumula_pares( xnum - 1));
    else return (0);
}
```

En este caso, la función presenta tres sentencias return que se corresponden con el caso base y los dos casos generales.

La invocación puede verse en el siguiente algoritmo:

```
int main()
{ int num;
  scanf("%d", &num);
  printf("\n suma de pares menores que: %d es %d", num, acumula_pares(num-1));
  getch();
}
```

A continuación se muestra en forma gráfica cómo se apilan los distintos registros de activación en la llamada de la función para un valor de num igual a 7.



Puede observarse que el valor del parámetro actual **num** no se modifica, a la salida de la ejecución de la función `acumula_pares`.

Funciones recursivas que devuelven más de un resultado

Modifiquemos el programa anterior de modo que la función retorne al programa principal los resultados de la suma de los números pares y de la suma de los números impares menores o iguales que un número leído desde teclado.

Ejemplo : Realizar el seguimiento, y mostrar la organización de la memoria cuando se ejecuta `acumula_pares_impares` con `xnum = 5`.

```
int acumula_pares_impares ( int xnum, int &imp)
{
    if (xnum)
        if ((xnum % 2) == 0)
            return xnum + acumula_pares_impares ( xnum - 1,imp);
        else
            { imp+= xnum;
              return (acumula_pares_impares ( xnum - 1,imp));
            }
        else return (0);
}
int main()
{ int num, impares=0;
  scanf("%d", &num);
  printf("\n suma de pares menores que: %d es %d", num, acumula_pares_impares(num-1,impares));
  printf("\n suma de impares impares es : %d:", impares);
  getch();
}
```

ACTIVIDAD 4: Realizar el seguimiento de la siguiente función recursiva.

- Analice qué realiza para los siguientes lotes de prueba [a:18, b:4]; [a:23, b:5]; [a:21, b:3]
- Realice el mapa de memoria para el primer lote de prueba.

```
int funcion (int x, int y, int &r)
{
    if ((x-y)<0)
        return 0;
    else
        {
            r= x-y;
            return 1 + funcion(x - y, y, r);
        }
}

int main (void )
{ int a,b,c=0;
  printf("\n INGRESE DOS VALORES\n ");
  scanf("%d %d", &a, &b);
  printf("resultados %d --- %d ", funcion(a,b,c), c);
  getch();
}
```

Recursividad usando Arreglos

Dentro de las estructuras que se pueden procesar recursivamente se encuentran los arreglos. A continuación, se analizan algunos ejemplos:

Ejemplo

Calcular recursivamente la suma de las componentes de un arreglo.

Se declara y se inicializa el arreglo arre:

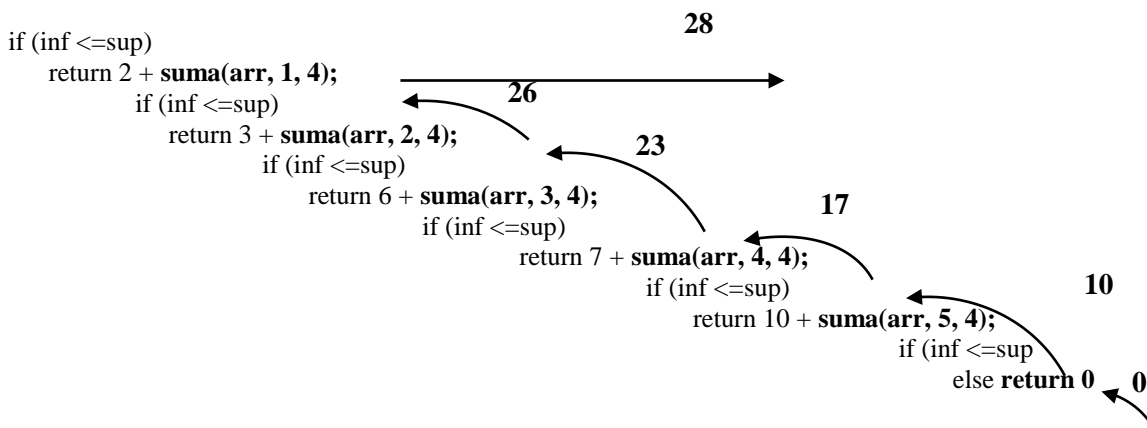
```
int arre[5]={2, 3, 6, 7, 10}
```

se define una función **suma** que reciba el arreglo arre, el límite inferior (0) y el límite superior (4).

La invocación desde el programa principal será **suma(arre, 0,4)** y el prototipo de la función es:

```
int suma (int arr[], int inf, int sup);
```

Esquema de la solución:



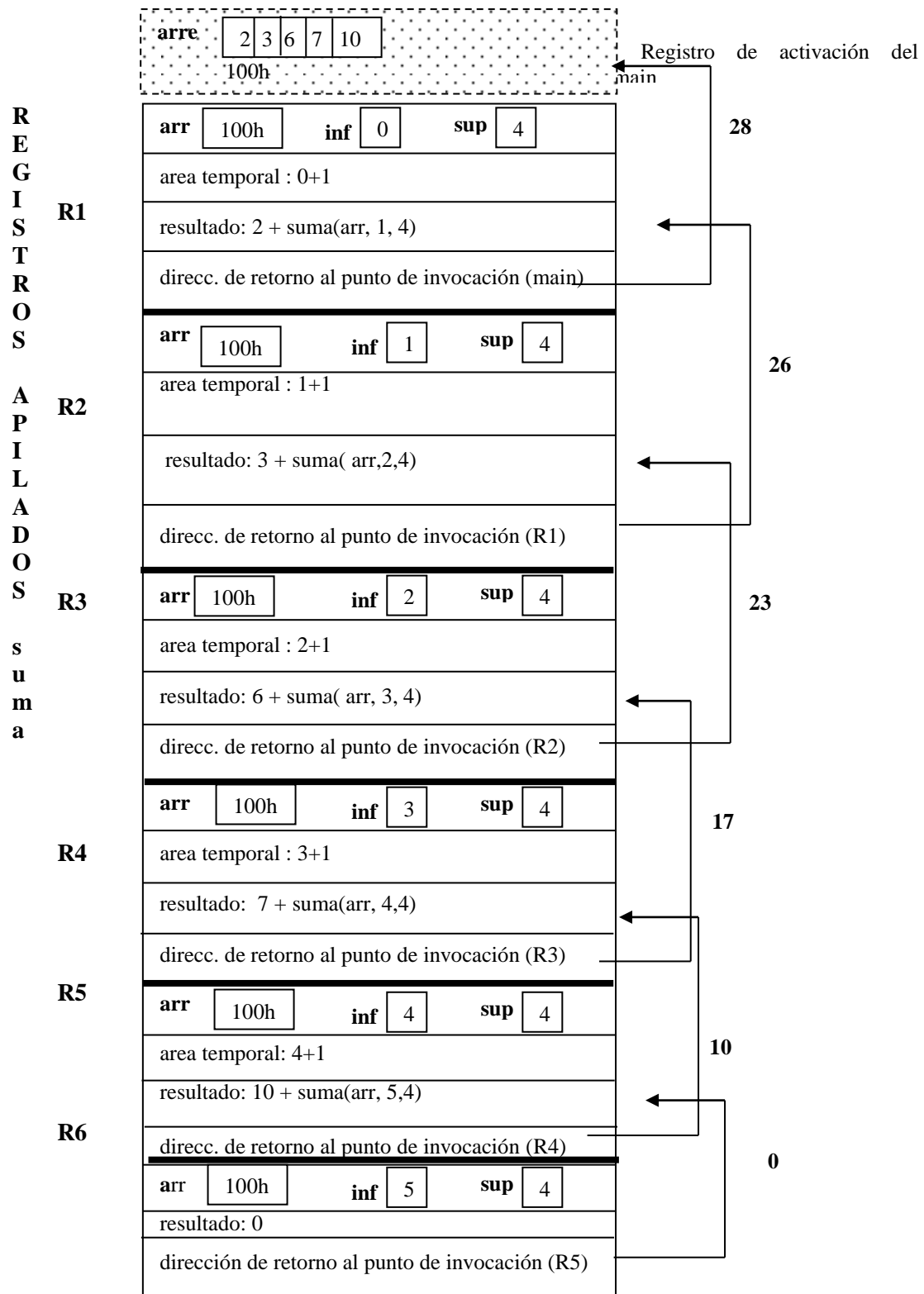
Código en lenguaje C

```
#include<stdio.h>
#include<conio.h>
int suma(int arr[], int inf, int sup)
{
    if (inf <=sup)
        return arr[inf] + suma(arr, inf+1, sup);
    else return 0;
}

int main()
{ int arre[5]={2, 3, 6, 7, 10} ;
  printf("\n Suma de las componentes %d", suma(arre, 0,4));
  getch();
}
```



Mapa de Memoria



ACTIVIDAD 5

Modifique el algoritmo anterior de modo que la función devuelva la suma de las componentes y el total de componentes mayores a 5.

Ejemplo : El siguiente programa usa las funciones recursivas **carga**, **busq_sec_rec** y **escalar**, para realizar las siguientes tareas:

- 1- Cargar dos vectores de 5 componentes enteras positivas
- 2- Dado un valor, indicar en cuál de los vectores se encuentra
- 3- Calcular el producto escalar de los dos vectores

```
void carga (int arr[], int i, int n)
{
    if (i!=n)
    {
        printf ("\n ingrese el valor de la posición %d:", i);
        scanf ("%d",arr+i);
        carga(arr, i+1,n);
    }
    else printf ("\n el arreglo esta cargado \n");
}

int busq_sec_rec ( int arr[], int xi, int n, int elem)
{
    if( xi==n )
        return -1;
    else if (arr[xi]==elem)
        return 1;
    else return busq_sec_rec ( arr, xi+1,n,elem);
}

// esta función recibe la última posición del arreglo
int escalar (int x[],int y[],int j)
{
    if (j >= 0)
        return x[j]*y[j]+ escalar(x,y,j-1);
    else return 0;
}

int main(void)
{
    int a[n], b[n], valor;
    printf ("\n Carga del primer vector \n"); carga(a,0);
    printf ("\n Carga del segundo vector \n"); carga(b,0);
    printf ("\n Ingrese valor a buscar en los vectores: ");
    scanf ("%d",&valor);
    if (busq_sec_rec(a,0,valor)==1)
        printf ("\n El valor está en el primer vector\n");
    else printf ("\n El valor NO está en el primer vector\n");
    if (busq_sec_rec(b,0,valor)==1)
        printf ("\n El valor está en el segundo vector\n");
    else printf ("\n El valor No está en el segundo vector \n");
    printf ("\n Producto escalar de los vectores = %d", escalar(a,b,n -1));
    getch();
}
```

NOTA

- a) La función **busq_sec_rec** recibe como parámetros formales el arreglo, la posición del primer elemento, el tamaño y el elemento a buscar. En esta función el recorrido del arreglo se inicia en la posición 0.
- b) La función **escalar** recibe dos arreglos de igual dimensión y la posición del último elemento de los arreglos. En este caso el recorrido se realiza desde la última componente del arreglo hasta la primera componente (posición 0), por eso no hace falta enviar como parámetro la posición del primer elemento.

ACTIVIDAD 6

1. Construya una función recursiva que invierta los elementos de un arreglo.
2. Construya la función recursiva búsqueda binaria

ACTIVIDAD 7

Realice el seguimiento del siguiente programa cuando se ejecuta la función **muestra**, y el mapa de memoria correspondiente.

```
void muestra(int arr[], int inf, int sup)
{
    if (inf <= sup)
    {
        muestra(arr, inf+1, sup);
        printf("\n %d", arr[inf]);
    }
    else return;
}

int main(void)
{
    int arre[5]={2, 3, 6, 7, 10};
    muestra(arre, 0,4);
    getch();
}
```

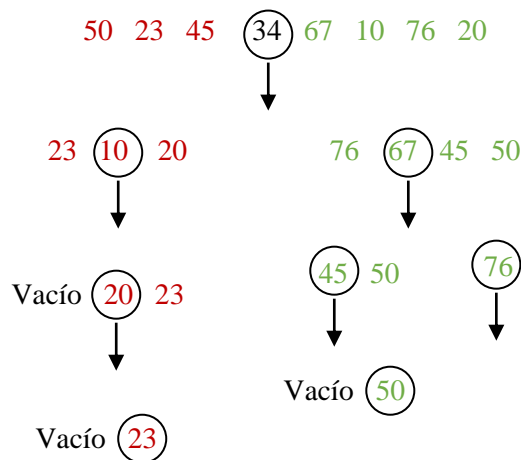
Ordenación Rápida (Quicksort)

El **Quicksort** es uno de los métodos más rápidos de ordenamiento. El método consiste en:

1. Elegir un elemento del arreglo denominado pivote
2. Dividir el arreglo en dos subarreglos, de modo que todos los elementos menores que el pivote estén en un subarreglo y los elementos mayores que el pivote estén en el otro.
3. Repetir el proceso para cada subarreglo hasta llegar a subarreglos con un único elemento.

Acá se nota la conveniencia del uso de un algoritmo recursivo.

Gráficamente:



Finalmente, el arreglo ordenado es (10 .20 .23 .34 .45 .50 .67 .76)

Ejemplo : El siguiente ejemplo corresponde al método de ordenamiento rápido **Quicksort**, versión recursiva, para un arreglo de 8 componentes.

```
#define long 8
void carga(int arr[long],int i)
{
    if (i<long)
    {
        printf("\n Ingrese %dº numero: ",i+1);
        scanf("%d",&arr[i]);
        carga(arr,i+1);
    }
} //fin de la funcion cargar
```

```
void quicksort(int arr[ ], int inf, int sup)
{ int t[long];
  int i,piv,m,n;
  if (inf < sup)
  {
      i=(inf+sup)/2;
      piv=arr[i];
      for( i= inf;i<=sup;i++)
          t[i]=arr[i];
      m=inf; n=sup;
      for(i=inf; i<= sup; i++)
          if (t[i] < piv)
          {
              arr[m]=t[i];
              m=m+1;
          }
      else if (t[i] > piv)
          {
              arr[n]=t[i];
              n=n-1;
          }
      for( i=m;i<=n;i++)
          arr[i]=piv;
      quicksort(arr,inf,m-1);
      quicksort(arr,n+1,sup);
  }
  return;
}
```

```
int main ()
{ int i;
  int a[long];
  carga(a,0);
  quicksort(a,0,long-1);
  for(i=0; i<long;i++)
      printf(" a[%d]= %d ",i,a[i]);
}
```

ACTIVIDAD 8

Realice el seguimiento del programa anterior para el arreglo que tiene los siguientes componentes:

50	23	45	34	67	10	76
----	----	----	----	----	----	----

Recursión versus Iteración

Tanto la recursión como la iteración se basan en una estructura de control:
 la iteración utiliza una estructura de repetición
 la recursión utiliza una estructura de selección.



Tanto la recursión como la iteración implican repetición:
 la iteración utiliza la repetición de manera explícita
 la recursión consigue la repetición mediante repetidas llamadas a una misma función.



La recursión y la iteración involucran una prueba de terminación:

- la iteración termina cuando falla la condición de continuación del ciclo
- la recursión termina cuando se reconoce un caso base.

Practico 3

Recursividad

Ejercicio 1

- Realizar el programa principal y la invocación de las siguientes funciones de tal manera que muestre en el main el resultado en caso de que corresponda.
- Realizar el Mapa de Memoria de las siguientes funciones.
- Indicar que hace cada una de ellas.

FUNCION	LOTE DE PRUEBA				
<pre>void Funcion A (int n) { if (n) { printf("%d", n%10); n=n/10; FuncionA (n); } }</pre>	n = 6745				
<pre>int FuncionB (int n) { if (n==0) return n; else return FuncionB (n/10)+(n%10); }</pre>	n = 5679				
<pre>int FuncionC (int x[], int n, int dato) { if(n==0) { if (dato > x[n]) return x [0]; else return dato; } else { if (dato > x[n]) Return FuncionC (x, n-1, x[n]); else Return FuncionC (x, n-1, dato); } }</pre>	X <table border="1"><tr><td>25</td><td>18</td><td>56</td><td>35</td></tr></table> n = 3 dato= 50	25	18	56	35
25	18	56	35		

- La definición recursiva de la multiplicación de dos números a y b (es decir el número a multiplicado b veces), se deriva de la definición de la multiplicación como una suma abreviada y la aplicación de la propiedad asociativa de la suma, es decir:

$$a*b = a + a + (b \text{ veces...}) + a$$

Se tiene que $a*b = a + a*(b-1)$

Realizar el seguimiento del siguiente algoritmo para **Lote de prueba** n=2 y b=3


```

void producto (int n, int b)
{
    if (b>1)
    {
        printf ("%d +", n);
        producto (n, b-1);
    }
    else printf ("%d ", n);
}

main()
{
    int n, b;
    scanf ("%d", &b);
    scanf ("%d", &n);
    printf ("\n %d * %d=", n, b);
    producto (n, b);
    getchar ();
}

```

Ejercicio 2

Si quieres conseguir un buen trabajo vas a necesitar buenas habilidades. Uno de los perfiles profesionales más demandados son los programadores, pero ¿qué lenguaje de programación merece la pena aprender?. Aprender a programar te abrirá puertas a otros empleos. Son muchas las empresas las que valoran esta habilidad, pese a que no sea necesario para el puesto, por la agilidad mental que denota. Por todo ello, la comunidad de desarrolladores Stack Overflow llevó a cabo encuestas sobre las tendencias del sector, sobre cuál de los siguientes lenguajes utilizan. Por cada encuestado se ingresa uno de los siguientes nombres:

1. **Javascript:** A pesar de tener nombres similares, Javascript no está relacionado con Java. Permite a los desarrolladores crear elementos interactivos en los sitios web, convirtiéndolo en uno de los lenguajes más omnipresentes de la web y el más popular del mundo.
2. **HTML:** Aunque técnicamente no es un lenguaje de programación - es un "lenguaje de marcas" - HTML es la base para la estructura de cada sitio web.
3. **Cascading Style Sheets, o CSS:** Es el lenguaje de programación más utilizado para diseñar sitios web y aplicaciones basadas en navegadores.
4. **Java:** Fue inventado originalmente por Sun Microsystems en 1991 como lenguaje de programación para sistemas de televisión interactiva. Desde la compra de Sun, Oracle ha convertido a Java en una potencia. El lenguaje de programación es la forma más común de construir aplicaciones en Android.
5. **Python:** Python data de 1989 y es amado por sus fans por su código altamente legible. Muchos programadores creen que es el lenguaje más fácil de usar.
6. **C:** Es uno de los lenguajes de programación más antiguos aún en uso común, fue creado a principios de la década de los 70. En 1978, el legendario manual del lenguaje, "The C Programming Language", fue publicado por primera vez.

Realizar un programa en C que , utilizando al menos una función recursiva, permita:

- a) Generar una estructura para almacenar los resultados de las encuestas realizadas, es decir que almacene la cantidad total de encuestados para cada lenguaje.
- b) Mostrar el/los lenguaje/s que tienen no más de 4000 programadores.
- c) Indicar el lenguaje más popular. (suponer único)
- d) Mostrar la cantidad de lenguajes que eligieron entre 5000 y 9000 programadores, como así también más de 9000.
- e) Mostrar el total de programadores registrados.
- f) Realizar el mapa de memoria cuando se invoca a la función construida en el punto e).

Ejercicio 3

Escribir la siguiente función iterativa en forma recursiva, para ello

int mcd (int a, int b)

```
{ int r;

if (a>b)
{   while(b>0)
    {       r=b;
            b=a%b;
            a=r;
    }//fin while
    return a;
} //fin if
else return 0;
} //fin mcd
```

Ejercicio 4

Construir un programa en lenguaje C que a través de funciones recursivas resuelva los siguientes ítems:

- Cargar un arreglo de enteros, de N componentes.
- Generar un subarreglo con las componentes del arreglo cargado, cuyo valor es mayor al Promedio.
- Indicar cuantas componentes del subarreglo mayores al promedio y cuantas menores a éste.
- Ingresar un número y decir si se encuentra en el subarreglo.
- Realice el ítem anterior si el arreglo original estuviera ordenado ascendentemente.

Ejercicio 5

Realice una función que busque el mayor valor de un arreglo, de modo tal que al llegar al caso base ya haya encontrado este valor; y en la etapa de volver al punto de invocación vaya mostrando los valores iguales al mayor.

Ejercicio 6

El Ministerio de Producción de la Nación ha lanzado un Plan de Promoción de Capacitación de Empleados (PPCE) para las PYMES (Pequeñas y Medianas Empresas). La siguiente tabla detalla los montos financiados para el año 2018 (expresados en millones de pesos) según las distintas categorías/sectores, lo que permite clasificar cada una de las empresas.

<i>Categoría</i>	<i>Sector</i>				
	Agropecuario	Industria y Minería	Comercio	Servicios	Construcción
Micro	\$2	\$7,5	\$9	\$2,5	\$3,5
Pequeña	\$13	\$45,5	\$55	\$15	\$22,5
Mediana	\$100	\$360	\$450	\$125	\$180

- Indicar el monto total financiado para una categoría ingresada por teclado.
- Indicar el monto total financiado para el sector de Servicios, sin importar la categoría de la empresa.
- Emitir un listado con el total financiado, sin importar la categoría/ sector.
- Emitir un listado con los montos superiores a uno ingresado por teclado, y a continuación los inferiores e iguales, indicando sector y categoría.

Ejercicio 7

Dadas dos matrices cuadradas A y B de componentes enteras positivas y de dimensión N, realizar un algoritmo en C que utilizando funciones recursivas permita:

- Cargar cada una de las matrices (función reusable).
- Calcular el producto escalar de una fila de A por una fila de B. Ingresar por teclado el número de cada fila.
- Calcular el producto escalar de una columna de A por una columna de B. Ingresar por teclado el número de las columnas.

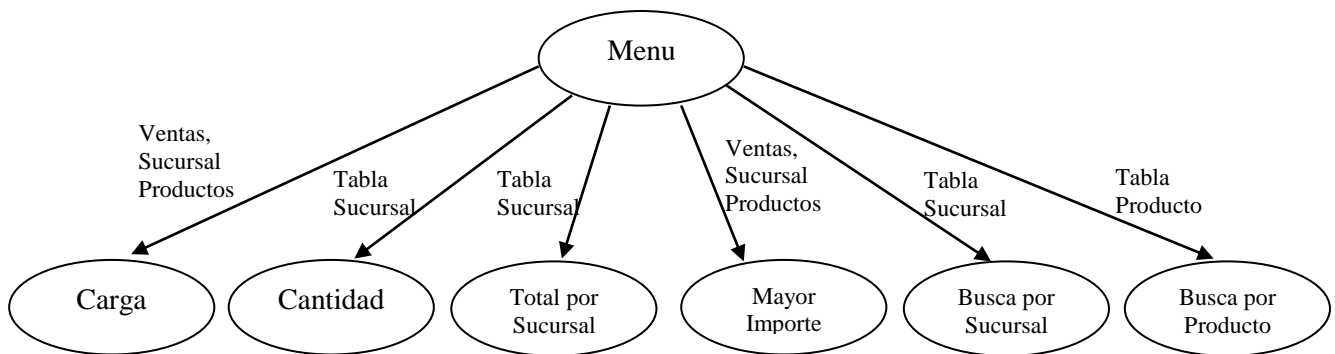
Nota: validar el ingreso de datos.

Ejercicio 8

La cadena de comidas rápidas Mostaza necesita información estratégica que permita apoyar la toma de decisiones en relación a las ventas realizadas en cada una de las 4 sucursales que dispone en la ciudad Autónoma de Buenos Aires. De cada sucursal se conoce: número, nombre, dirección y teléfono. En cuanto a los 10 productos distintos que comercializa, se conoce: código de producto, nombre, calorías y precio. Por cada venta realizada, se ingresa el número de sucursal, código de producto y cantidad. Esta información no tiene ningún orden en particular, y el ingreso termina con número de sucursal 0. Concretamente necesita conocer:

- Cantidad de productos vendidos por sucursal.
- Importe total de productos vendidos por sucursal.
- Obtener la sucursal (nombre) y el producto (nombre y precio), que registró el mayor importe de venta.
- Dado un número de sucursal, indicar el producto (todos los datos) que registró el mayor consumo de calorías (suponer único).
- Dado un número de producto, indicar la sucursal (nombre y teléfono) donde se registró el menor importe vendido.

NOTA: Es importante optimizar el código, por lo tanto cuando deba trabajar sobre una fila de la tabla, pasar sólo la fila. Realice un menú de opciones y validar los datos de entrada.



Unidad 5: Estructuras Dinámicas

Introducción

Hasta ahora los tipos de datos simples y estructurados con que se ha trabajado son de almacenamiento fijo o estático. Esto significa que el sistema se encarga de asignar a cada variable el área que ocupará en tiempo de compilación, la que permanecerá fija durante toda la ejecución de la función que la declara. No se podrá utilizar por tanto esas posiciones de memoria para otros fines, durante dicha ejecución. En el caso de variables globales, el almacenamiento se mantiene fijo durante toda la ejecución del programa y se usa para ellas la zona de memoria que llamamos de **Almacenamiento Estático**. Para las variables locales, el almacenamiento que se les asigna corresponde a la **Pila (Stack)**.

Existen situaciones en las que es más apropiado utilizar variables o estructuras que puedan crearse en tiempo de ejecución, de modo que se permita liberar su espacio de almacenamiento cuando no se las necesite así como modificar su tamaño durante la ejecución del programa. A este tipo de variables y estructuras se las denomina *dinámicas*.

El uso de punteros o apuntadores permite el manejo de objetos de datos dinámicos.

La diferencia entre objetos de datos creados en tiempo de compilación y los creados en tiempo de ejecución es que:

- El objeto creado en tiempo de ejecución no necesita tener nombre
- Estos objetos se pueden crear en cualquier punto durante la ejecución de un programa, no sólo al entrar al subprograma.

Se puede sintetizar diciendo que una variable dinámica se crea y se libera por petición, durante la ejecución del programa y no en forma automática como las de almacenamiento estático.

El Montículo o Montón(Heap)

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución, no es posible su ubicación en memoria estática, ni en la pila. Son ejemplos de este tipo de objetos los arreglos dinámicos, las listas enlazadas, las cadenas de caracteres de longitud variable, etc. Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable y que no se vea afectada por la activación o desactivación de subprogramas. Este trozo de memoria se llama **montículo o montón** (traducción de *heap*).

Las operaciones básicas que se realizan sobre el montículo son:

Alojamiento: Se demanda un bloque de memoria para almacenar un objeto de un cierto tamaño.

Desalojo: Se indica que ya no es necesario conservar un objeto de dato, la memoria que ocupa debe quedar libre para ser reutilizada en caso necesario por otros objetos.

Según sea el programador o el propio sistema el que las invoque, estas operaciones pueden ser explícitas o implícitas respectivamente. En caso de alojamiento explícito el programador incluye en el código fuente una instrucción que demanda una cierta cantidad de memoria para la ubicación de un dato (en C mediante *malloc*, en PASCAL mediante la instrucción *new*, etc.).

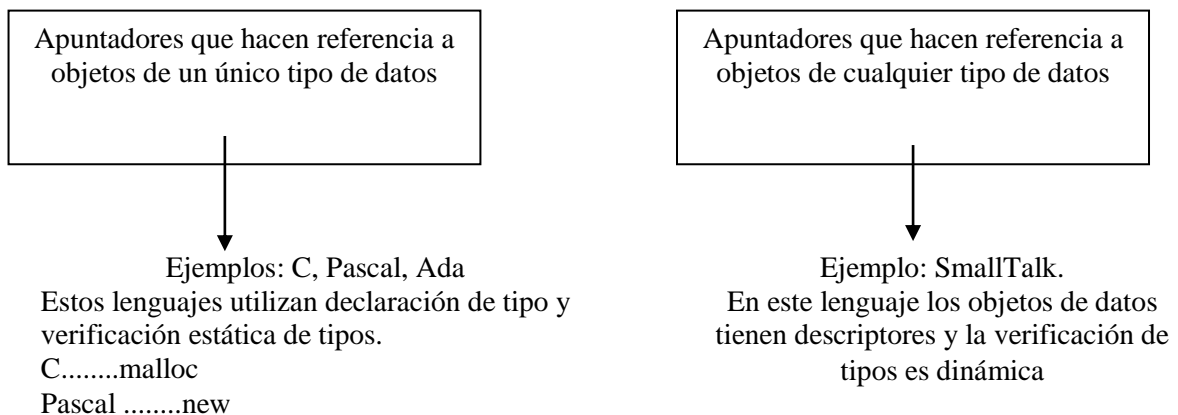
La cantidad de memoria requerida puede ser calculada por el compilador en función del tipo de dato correspondiente al objeto que se desea alojar, o puede ser especificado directamente por el programador. El resultado de la función de alojamiento es por lo general un puntero a un trozo contiguo de memoria dentro del montículo que puede usarse para almacenar el valor del objeto. Los lenguajes de programación imperativos utilizan por lo general alojamiento y desalojo explícitos.

La gestión del montículo requiere técnicas adecuadas que optimicen el espacio que se ocupa o el tiempo de acceso, o bien ambos factores.

Como los apuntadores permiten definir objetos de datos de tamaño variables, el lenguaje debe poseer las siguientes características:

1. Un tipo elemental de datos apuntador, también llamado tipo de referencia o de acceso (un tipo puntero).
2. Una operación de creación de objetos de tamaño fijo. La operación devuelve el valor L del bloque de almacenamiento creado y ese valor L se convierte en el valor R de una variable del tipo apuntador.
3. Una operación de desreferenciar para objetos del tipo apuntador, la cual permite acceder al valor del objeto al cual apunta.

El tipo apuntador se puede tratar de dos maneras:



IMPLEMENTACION: Existen dos formas de representación de almacenamiento para valores de una variable apuntador:

Direcciones Absolutas: El apuntador contiene la dirección de memoria real del bloque de almacenamiento del objeto de datos.

Direcciones Relativas: El apuntador contiene el desplazamiento del objeto respecto a la dirección base de un bloque determinado de almacenamiento más grande que contiene al objeto (montículo).

Manejo del Montículo en Lenguaje C

FUNCIONES MALLOC Y FREE

El lenguaje C provee funciones que permiten el manejo de memoria de forma dinámica. La función **malloc** (Memory **ALLO**Cation: asignación de memoria) permite asignar espacio en tiempo de ejecución y la función **free** permite liberarlo. Los prototipos de estas funciones se encuentran en el archivo de cabecera `<alloc.h>` para el caso de C++ y `<malloc.h>` para el caso de DEV C++.

La función **malloc** solicita al sistema operativo un bloque de memoria en el montículo, del tamaño especificado en el argumento. Su resultado es la dirección de comienzo del bloque asignado o la constante **NULL**, si no hay espacio disponible.



Formato: (void *) malloc (tamaño del bloque)

donde *tamaño*, hace referencia a la longitud en bytes del bloque.

La función malloc devuelve un puntero a void o puntero genérico, por esto se debe realizar la conversión forzada de tipo, utilizando el operador cast, que permite especificar el tipo de dato al que debe apuntar el puntero.

Por ejemplo si se debe solicitar espacio para almacenar 10 componentes enteras, la función malloc es:
int *p;

p= (int *) malloc(10* sizeof(int))

Op. Cast Pedido de Memoria Tamaño del Bloque

malloc devuelve la dirección de comienzo del bloque en un puntero a void, por ese motivo se debe hacer una conversión explícita de tipo (cast) para convertirlo en un puntero a entero.

El Sistema Operativo maneja una tabla de direcciones de memoria que indica el espacio de memoria ocupado (para que un programa no modifique direcciones de memoria que no le corresponden, corrompiendo la memoria de otros procesos / programas / información).

malloc indica que se va a cargar en memoria una estructura y por ello el sistema operativo registra en su tabla de direcciones de memoria que un bloque ha sido ocupado, y envía a malloc la dirección de inicio de ese bloque.

La función free, libera bloques asignados previamente por malloc. De esta manera existe más espacio de memoria disponible para posteriores asignaciones.

Formato: free (variable puntero);

Por ejemplo si se desea liberar el espacio antes solicitado, la función free es:

free(p);

free indica que el bloque de memoria asignado a través de malloc se ha dejado de usar, con esta información el Sistema Operativo registra en su tabla de direcciones de memoria que dicho bloque ha sido desocupado. La variable p conserva el valor que tenía antes de free, es decir no cambia su contenido, por este motivo es que para evitar errores de acceso, resulta ser una buena práctica inicializar la variable después de su liberación (p = NULL;).



Lenguaje C: Variables dinámicas simples

El siguiente ejemplo muestra las sentencias necesarias para generar una variable dinámica de tipo entero.

Ejemplo 1: Almacenar en una variable dinámica un número entero

```
#include <alloc.h>
void main(void)
{
    int * n;                1
    n = (int *) malloc (sizeof (int))  2
    *n=12;                  3
    :
    free(n);                4
}
```

En la sentencia 1 se declara un puntero a entero, luego en la sentencia 2, se solicita espacio para almacenar en el montículo una variable dinámica entera y la dirección obtenida como resultado, se guarda en el puntero n.

Mediante la sentencia 3, se asigna un valor a la variable apuntada por el puntero.

La sentencia 4 permite liberar el espacio reservado a la variable dinámica apuntada por n.

La figura 1 muestra la organización de la pila y el montículo, antes de invocar a la función free.

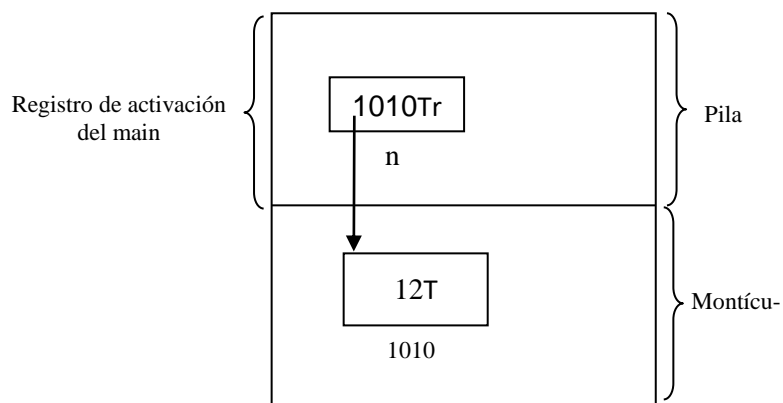


Fig. 1 Organización de la pila y el montículo cuando se crea una variable dinámica

La variable dinámica se almacena en el montículo y se libera cuando ya no se necesita, quedando ese espacio libre para posteriores asignaciones.

¿Cómo sería la gráfica de memoria después de ejecutar free(n);? Exactamente igual, solo cambiaría la Tabla de direcciones de memoria del Sistema Operativo.

Un problema a tener en cuenta

El problema principal de los apuntes y los objetos de datos construidos por el programador es la asignación de memoria asociada a la operación “crear”.

El **lenguaje C** utiliza para manejo de funciones y recursividad una gestión de almacenamiento basada en pilas. Cuando se agregan punteros con la operación malloc, es necesaria una ampliación considerable de la estructura global de la gestión de almacenamiento.

Para objetos de tamaño fijo, el tiempo de vida se *inicia* con la asignación de una localidad de almacenamiento para el objeto (enlace del objeto al bloque de almacenamiento) y *termina* cuando se elimina el enlace del objeto al bloque de almacenamiento. Cuando se crea el objeto, es decir al inicio de su tiempo

de vida, se crea una *ruta de acceso* al objeto, para que las operaciones del programa puedan tener acceso al objeto de datos. La ruta de acceso se logra asociando al objeto un identificador o nombre o bien a través de un puntero al objeto el cual se guarda en otro objeto conocido y de fácil acceso.

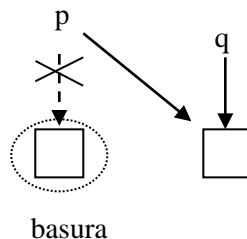
Puede haber varias rutas de acceso adicionales, por ejemplo cuando se pasa un objeto como argumento en un subprograma o cuando se crean varios punteros al objeto. Por lo tanto, existen varios problemas que se originan por la acción conjunta del tiempo de vida y las rutas de acceso a una estructura de datos, a saber:

Basura: En el contexto de la gestión de almacenamiento en montículos, un elemento basura es aquel que está disponible para nuevo uso pero no se halla en la lista de espacios libres de la Tabla de direcciones de memoria del Sistema Operativo, es decir no se ha ejecutado una sentencia `free`, por ello se ha vuelto inaccesible. Si se pierde la ruta de acceso a una estructura (por ejemplo a causa de una sobre escritura por asignación) pero la estructura misma no ha sido liberada para que se recupere el almacenamiento, entonces la estructura se convierte en basura.

Si la basura se acumula, el almacenamiento disponible se reduce de manera gradual hasta que el programa puede ser incapaz de continuar por falta de espacio libre.

Ejemplo en lenguaje C:

```
int *p, *q;
q=(int *) malloc(sizeof (int));
p=(int *) malloc(sizeof (int));
....
p=q;
....
free(p);
....
```



Referencias desactivadas: Si se destruye un objeto creado dinámicamente (se libera el almacenamiento usando `free`) antes de haber eliminado todas las rutas de acceso a él (se hicieron asignaciones de ese objeto a otras variables), toda ruta de acceso remanente (esas otras variables) se convierte en una referencia desactivada. Por lo tanto, si luego se usa esa ruta para hacer una asignación podría modificarse el almacenamiento de algún otro objeto o provocar errores en la integridad del sistema de gestión de almacenamiento.

Ejemplo 2

```
void modifica(int *&p)
{
    int *x;
    x=(int*)malloc(sizeof(int));
    *x=25;
    p=x;
    free(x);
}

void main(void)
{
    int *n;
    modifica (n);
    :
}
```

El uso de **p** después de ejecutar la función `modifica` puede acarrear problema al apuntar a una dirección de memoria que ha sido liberada con `x`.

Lenguaje C: Arreglos dinámicos

El arreglo, tal como se ha visto hasta el momento, es un ejemplo de una estructura de almacenamiento fijo. El espacio reservado cuando se declara la estructura, no puede modificarse durante su tiempo de vida.

Cuando la asignación de almacenamiento es estática, las direcciones de almacenamiento para todos los datos se generan en tiempo de compilación, esto hace eficiente a este mecanismo de gestión de memoria, por cuanto no se gasta tiempo durante la ejecución.

Sin embargo, en muchos casos, la utilización de estructuras de almacenamiento fijo no conduce a la solución más eficiente del problema desde el punto de vista del manejo de memoria. Por ejemplo, si se define un arreglo de 80 componentes y durante la ejecución del programa sólo se ocupan 30, se están desperdiciando 50 lugares. Por el contrario, podría ocurrir que este espacio reservado resultara insuficiente.

Como el nombre del arreglo es un puntero constante al primer elemento del mismo, también se puede definir un arreglo como una variable puntero, en lugar de hacerlo de la forma convencional.

Un arreglo de 10 componentes enteras se puede declarar entonces de dos maneras:

int b[10]; ó utilizando una variable puntero **int *b;**

Estas dos declaraciones difieren en cuanto al almacenamiento de memoria.

Con la definición convencional **int b[10]**, el sistema reserva un *bloque fijo y consecutivo* de diez celdas de memoria, que permanece durante toda la ejecución de la función en la que el arreglo está declarado. Por esto también se dice que el almacenamiento es estático.

Gráficamente:



Si se analiza la segunda declaración **int *b**, surge una primera pregunta ¿es posible, a partir de un puntero a entero, reservar espacio para almacenar diez números?

Como se ha visto, la función **malloc** solicita al sistema operativo, en tiempo de ejecución, la asignación de un bloque de memoria en el montículo del tamaño especificado en su argumento.

En el ejemplo considerado, se debe solicitar espacio para almacenar 10 componentes enteras, de la siguiente manera:

b = (int *) malloc(10 * sizeof(int));

El puntero **b** recibe la dirección de comienzo del bloque o la constante **NULL** si no hay espacio disponible.

Este modo de creación de arreglos tiene dos momentos:

Declaración de la variable de acceso (nombre del arreglo): **int *b;**

Asignación dinámica de memoria: **b = (int *) malloc (10 * sizeof(int));**

A diferencia la declaración estática que hace ambos pasos en un mismo tiempo.

La siguiente, es una síntesis de las diferencias entre un arreglo estático y un arreglo dinámico, en cuanto a la declaración, manipulación y almacenamiento.

Arreglo Estático

```
void carga(int x[], int n)
```

```
{ int i;
  for(i=0; i<n; i++)
    scanf("%d",&x[i]);
}
```

```
void main(void)void main(void)
```

```
{ int b[10];
```

```
carga(b, 10);
```

```
....
```

```
}
```

Arreglo Dinámico

```
void carga (int *x, int n)
```

```
{ int i;
  for (i=0; i<n; i++)
    scanf("%d",&x[i])
}
```

```
{ int *b;
```

```
  b=(*int) malloc(sizeof(int)*10);
```

```
  carga(b,10);
```

```
.....
```

```
free(b);
```

```
}
```

Como se observa en la figura 2, el arreglo estático se genera en la pila, y está disponible durante toda la ejecución de la función que lo declara. El arreglo dinámico se genera en el montículo, por lo tanto su almacenamiento puede liberarse cuando se considere adecuado.

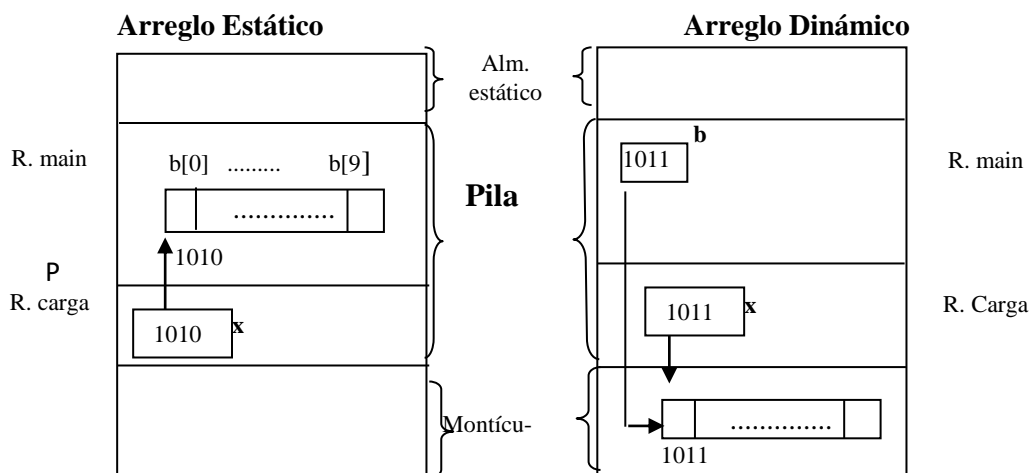


Fig. 2 Organización de la memoria cuando se genera un arreglo estático y un arreglo dinámico

Ejemplo 3

El Instituto Provincial de la Vivienda ha implementado un sistema que consta de 5 planes de pago distintos, con el fin de que los adjudicatarios de sus viviendas puedan cancelar sus deudas. Por cada uno de los 5 planes, se ingresa en forma ordenada la cantidad de adjudicatarios adheridos y por cada uno de ellos el monto adeudado.

Se necesita realizar un programa, que utilizando de manera óptima funciones, informe:

Para cada plan:

- Monto total adeudado por los adjudicatarios adheridos al mismo.
- Monto promedio adeudado.
- Cantidad de usuarios cuyo monto adeudado es superior al promedio calculado.
- El o los números de planes con mayor cantidad de adjudicatarios

Un análisis de la situación propuesta, plantea la necesidad de generar las siguientes estructuras: un arreglo manejado en forma dinámica para cada uno de los planes, que almacene la deuda de los usuarios adheridos al mismo y un arreglo estático de 5 componentes que contenga la cantidad de usuarios adheridos a cada plan.

El siguiente código resuelve el problema planteado.

```
#include <stdio.h>
#include <alloc.h>
#include <ctype.h>
#define N 5

void carga( float *t, int xcu)
{
    printf("\n Ingrese la deuda de los %3d Usuarios ",xcu);
    for( int i=0; i<xcu; i++)
        scanf("%f",&t[i] );
    return;
}

float total( float *t, int cu)
{
    float prom=0;
    for(int i=0; i<cu;i++)
        prom=prom+ t[i];
    printf("\n Total adeudado %5.2f ", prom );
    prom= prom/cu;
    return prom;
}

int cantidad( float *t, int cu, float prom)
{
    int c=0;
    for( int i=0; i<cu; i++)
        if (t[i]>prom)
            c++;
    return(c);
}

void maximo(int *tot, int N)
{
    int max=0, i;
    for(i=0; i < N; i++)
        if (tot[i] > max)
            max=tot[i];
    printf("\n Planes con mayor cantidad de adjudicatarios" );
    for( i=0; i < N; i++)
        if (tot[i] == max )
            printf("\n  %3d ",i+1 );
    return;
}
```

```

void main(void)
{ int i,cu,totu[5];
  float *monto, prom;
  for(i=0; i<N;i++)
  { printf("\n Ingrese cantidad de adjudicatarios del plan %3d ",i+1 );
    scanf("%d",&cu);
    totu[i]=cu;
    monto = (float *) malloc(cu * sizeof(float)); /* solicita espacio para almacenar la deuda
    carga(monto, cu);                             de los adjudicat. de un plan*/
    prom=total(monto, cu);
    printf("\n Promedio adeudado por Adjudicatarios del plan %3d es %5.2f ", i+1, prom );
    printf("\n Total adjudic. con monto superior a promedio %4d",cantidad(monto, cu,prom));
    free(monto); // libera el espacio asignado a un plan, cuando no se necesita
  }
  maximo(totu,5);
}

```

La figura 3 muestra el manejo de memoria cuando se invoca la función carga. Suponiendo que el programa principal ejecuta sólo el primer plan y que dicho plan tiene 8 adjudicatarios.

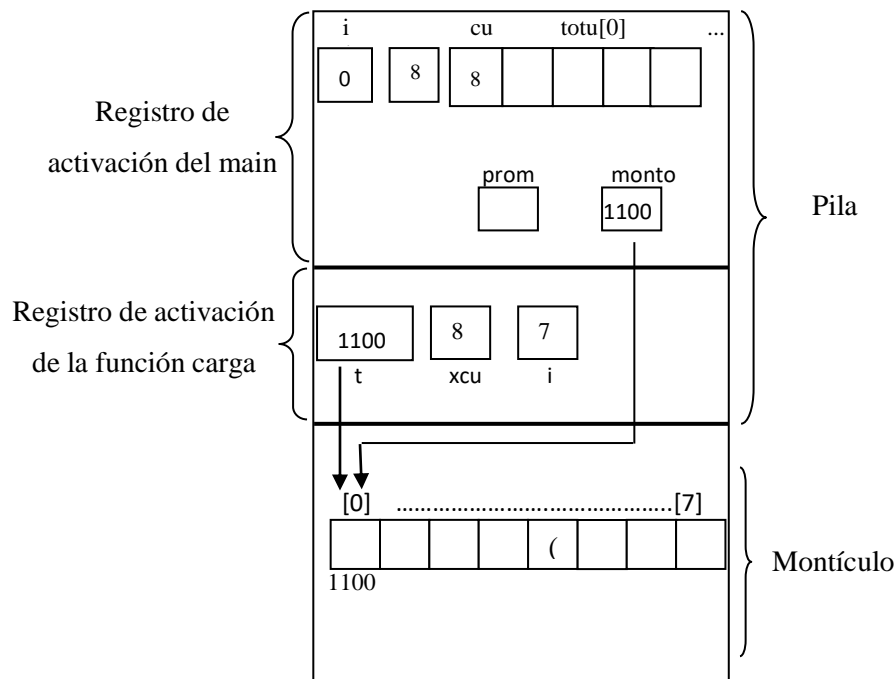


Fig. 3 Organización de la memoria cuando se ejecuta la función carga

En el montículo se reserva espacio para un único arreglo unidimensional por cada plan, cuyo tamaño varía según un valor ingresado desde teclado. El espacio reservado es liberado cada vez que se termina de procesar la información del plan correspondiente.

ACTIVIDAD 1

Codificar nuevamente la función carga, suponiendo que en ella se solicita el espacio de memoria para almacenar el arreglo monto.

1. ¿Qué cambios realizó en la función?
2. Muestre la organización de la memoria cuando se ejecuta la función carga

ACTIVIDAD 2

Defina la estructura de datos adecuada que le permita resolver la siguiente situación problemática:

El Instituto Provincial de la Vivienda ha implementado un sistema que consta de 5 planes de pago distintos, con el fin de que los adjudicatarios de sus viviendas puedan cancelar sus deudas. Por cada uno de los 5 planes, se ingresa en forma ordenada la cantidad de adjudicatarios adheridos y por cada uno de ellos el DNI y monto adeudado. Se pide:

1. Realice la carga de la información
2. Para un adjudicatario cuyo DNI se ingresa por teclado, indicar el número de plan al cual se adhirió y el monto adeudado.
3. Muestre el esquema de memoria, después de ejecutar la función que carga los datos

ACTIVIDAD 3

Se ingresa el registro y la nota obtenida (valor entero entre 1 y 10) por los alumnos que rindieron en la última mesa de examen de la materia Programación Procedural. Escribir un programa en C que permita:

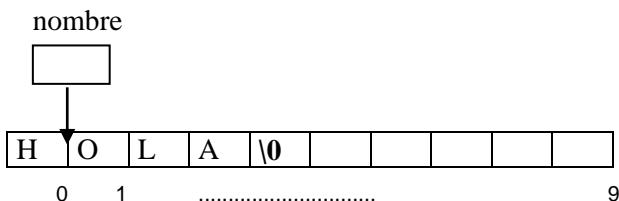
- Definir una función carga que almacene toda la información de la mesa de examen. Por teclado se ingresa la cantidad de alumnos que rindieron el examen.
- Mostrar la nota promedio y el número de registro de quienes obtuvieron en el examen una nota mayor o igual al promedio
- Construir una función que usando la menor y la mayor nota obtenidas, indique la cantidad de alumnos que obtuvieron cada nota entera comprendida en ese rango.
- Realizar el mapa de memoria después de ejecutar la función carga.

Arreglos dinámicos usados como cadena de caracteres

Como se ha visto, la estructura soporte de una cadena de caracteres es un arreglo del tipo char, que se caracteriza por finalizar con el carácter '\0'.

Por lo tanto, siendo char nombre[10] un arreglo, su nombre no es más que la dirección de memoria del primer carácter del arreglo.

Si la cadena de caracteres almacenada es "HOLA", gráficamente se representa:



Del mismo modo que a partir de un puntero a un entero, es posible generar un arreglo dinámico de enteros; a partir de un puntero a un carácter puede generarse un arreglo dinámico de caracteres.

char *nombre; declara a nombre como un puntero a un carácter. El espacio para almacenar la cadena se asigna dinámicamente en tiempo de ejecución, según el tamaño de la misma, evitando de ese modo el sobredimensionamiento de la memoria.

El siguiente ejemplo, muestra la carga de una cadena de caracteres como un arreglo dinámico.

```
void main(void)
{
    char *nombre, *aux;
    int l;
    aux=(char *) malloc( 30* sizeof(char)); // solicita memoria para la variable aux, con un tamaño
    suficientemente grande
    puts("Ingrese Nombre:\n");
    gets(aux);
    l = strlen(aux)+1;
    nombre=(char *) malloc( l* sizeof(char)); //solicita memoria necesaria para almacenar la variable
```

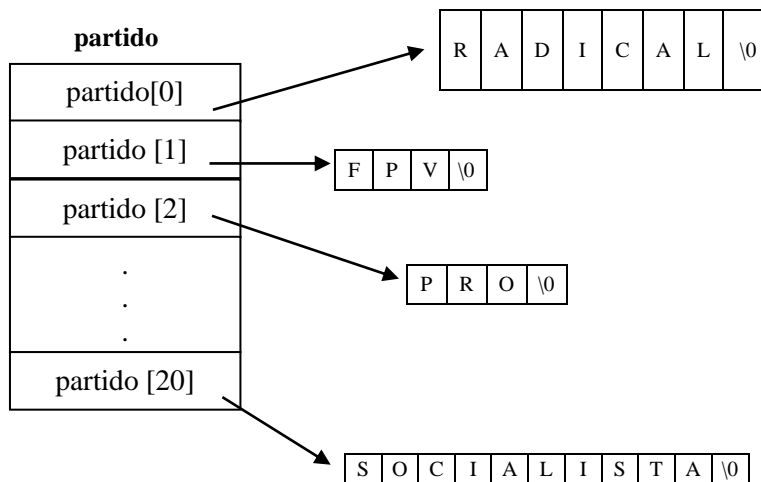

La declaración en este caso es:

```
char * partido[21];
```

donde **partido** es un arreglo de 21 punteros a char.

Cada uno de los 21 punteros contendrá la dirección del primer carácter de una cadena. Esto permite, a diferencia de la declaración anterior, que cada cadena ocupe solo la memoria requerida.

En este caso el almacenamiento es:



El formato para definir un arreglo de punteros a caracteres es el siguiente:

```
char * <identificador arreglo> [tamaño]
```

Ejemplo 4

El siguiente programa muestra una forma de leer y escribir, un conjunto de cadenas de caracteres de distinta longitud.

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

```
void carga(char *nom[21], int cfil)
{ char *aux;
  int i, l;
  aux=(char*) malloc(30*sizeof(char)); // solicita memoria para la variable aux
  puts(" Ingrese Nombres:\n");
  for(i=0; i<cfil; i++)
  { printf("\n Partido %d: ", i+1);
    gets(aux);
    l = strlen(aux)+1;
    nom[i] =(char *) malloc( l* sizeof(char));
    strcpy(nom[i],aux);
  }
  free (aux);
}
```



```
void muestra(char *nom[cfil])
{   printf("\nPartidos Ingresados: ");
    for (int i=0; i<cfil; i++)
        puts(nom[i]);
}
```

```
int main(void)
{
    char* nomb[21];
    carga(nomb,21);
    muestra(nomb,21);
}
```

ACTIVIDAD 4

Suponga que cuenta con los nombres y edad de 50 empleados de una empresa y desea:

a - Mostrar el nombre del empleado de mayor edad.

b - Dado un carácter, decir la cantidad de veces que aparece en los nombres de los empleados

Completar el siguiente programa con las funciones que le permitan resolver la problemática planteada.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
typedef struct
{   char *nom;
    int edad;
} empleados;
.... carga (.....)
.. maximo(...)
```

```
void mostrar(empleados *x,int n, char r)
{
    int i,j,l,c=0;
    for(i=0; i<n; i++)
    {   l=strlen( x[i].nom);
        for(j=0;j<l ; j++)
            if (x[i].nom[j]== r)
                c++ ;
    }
    printf("\ncantidad de letras %c es %d \n",r,c);
}
```

```
void libera (empleados *x,int n)
{
    int i;
    for(i=0; i<n; i++)
    {   free( x[i].nom);
        // x[i].nom= NULL;
    }
}
```

```
void muestra (empleados *x,int n)
{
    int i;
    for(i=0; i<n; i++)
        puts( x[i].nom);
}
```

```

int main()
{
    char e;
    empleados *a;
    int n = 50;
    carga(a,n);
    printf("\ mayor edad %s ", maximo(a,n));
    printf("\n ingrese un caracter %c",e);
    e = getchar();
    mostrar(a,n,e);
    muestra(a,n);
    libera(a,n);
    muestra(a,n);
}

```

Indicar si es necesario colocar la sentencia `x[i].nom= NULL` en la función libera.

Listas

Una lista es una estructura de datos compuesta por una serie de objetos de datos vinculados entre sí, donde se almacenan datos del mismo tipo, con la característica que puede contener un número indeterminado de elementos y que, mantienen un orden explícito (es decir que bajo un criterio se organizan sus posiciones en la estructura).

Las listas son estructuras de datos dinámicos, por tanto, pueden cambiar de tamaño durante la ejecución del programa, aumentando o disminuyendo el número de nodos.

Definiremos como lista a un conjunto de elementos del mismo tipo, que puede crecer o contraerse sin restricciones, todos los elementos son accesibles y se puede insertar y suprimir un elemento en cualquier posición de la misma.

Implementación de listas

Para implementar listas podemos utilizar una estructura de arreglo o bien utilizando variables del tipo apuntador.

Implementación de listas mediante arreglos

La implementación de listas mediante arreglos recibe el nombre de **listas secuenciales**. En este caso, los elementos se almacenan en celdas contiguas de memoria. Esta representación permite recorrer con facilidad una lista y agregarle nuevos elementos al final.

Insertar un elemento en la mitad de la lista, obliga al desplazamiento de una posición a todos los elementos que siguen al nuevo, para concederle espacio. De la misma manera, la eliminación de un elemento, excepto el último, requiere desplazamientos para llenar el vacío formado.

La mayor desventaja de las listas secuenciales es tener que asignar suficiente espacio de memoria, de manera de poder almacenar todos los elementos de la lista cuya cantidad no se conoce a priori. Este espacio permanece asignado, aún cuando la lista use una cantidad menor o incluso ninguno.

Además, redimensionar la lista cambiando el tamaño del arreglo cuando está completo, resulta ser una tarea lenta y tediosa (crear otro arreglo y copiar). Por otro lado, el no asignar más memoria introduce la posibilidad de desborde.

Este tipo de implementación tiene sus ventajas y desventajas, dependiendo de las operaciones que se quieran realizar sobre las listas.

Esta implementación se verá en años superiores al igual que otras implementaciones.

Implementación de listas mediante punteros

Las verdaderas posibilidades que ofrece la creación de variables dinámicas, se ponen de manifiesto al poder crear varios elementos de un mismo tipo de dato que se relacionen entre sí.

Para hacer posible la relación entre ellos, cada elemento deberá contener además de su información intrínseca, un puntero a otro elemento de estructura similar.

Los elementos se crean y/o liberan durante la ejecución del programa de acuerdo a los requerimientos del problema planteado.

Esta implementación de una lista recibe el nombre de **listas enlazadas**, permite almacenar los elementos en celdas de memoria no contiguas, de esta manera desaparece el problema de los costosos desplazamientos de elementos para insertar y eliminar componentes. No obstante, hay que pagar el precio de un espacio adicional para los apuntadores.

En esta representación cada elemento (nodo) de una lista debe tener dos partes:

- una contiene la información de cualquier tipo de dato simple o estructurado excepto FILE
- la otra parte, contiene un enlace o puntero que indica la posición del siguiente elemento.

Por lo tanto, cada elemento (nodo) tiene la estructura de tipo *struct*.

Estructura de un elemento (nodo) de una lista

Datos	Puntero
-------	---------

Es un tipo de dato autoreferenciado porque contienen un puntero o enlace a otro dato del mismo tipo. Las listas pueden concatenarse entre sí o dividirse en sublistas.

NOTA

En adelante se referirá a lista enlazada simplemente como lista, y a los elementos de la lista como nodos.

Ejemplo 5

a) A continuación se muestra gráficamente como se representa la siguiente lista de números enteros, por medio de un arreglo de longitud N.

3 4 -8 9 15 23

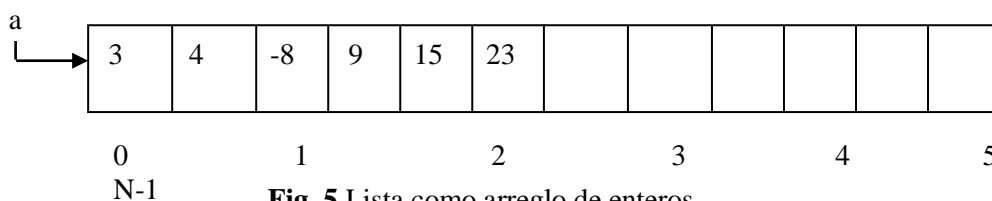


Fig. 5 Lista como arreglo de enteros

En este ejemplo, hemos supuesto que el número máximo de componentes a almacenar es N. Si la cantidad de componentes es superior, entonces para evitar el desborde de memoria, el arreglo debería redimensionarse.

b) Punteros: A continuación se muestra gráficamente una forma de representar una lista de números enteros por medio de punteros.

3 4 -8 9 15 23

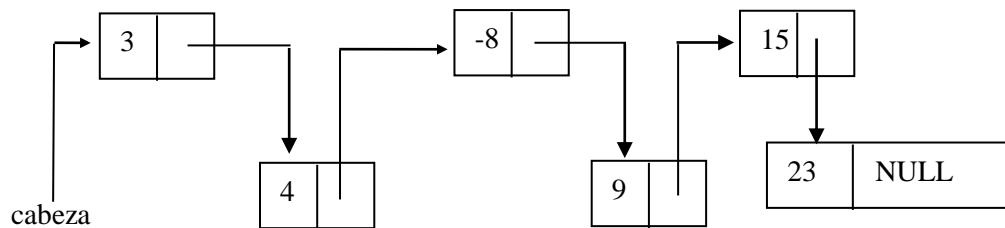


Fig. 6 Lista enlazada de números enteros

Se puede observar que en cada nodo, el campo enlace apunta al siguiente nodo de la lista excepto el último nodo que contiene NULL, para indicar el final de la lista. Como anteriormente se explicó, este valor de apuntador NULL no hace referencia a un lugar de memoria y se puede asignar a cualquier variable apuntador. Como puede inferirse, es ilegal una referencia al contenido de una variable puntero que tiene asignado NULL.

Existe la necesidad de un puntero que contenga en todo momento la dirección del primer elemento de la lista, denominado *cabeza de la lista*. Resguardar la cabeza de la lista garantiza el acceso a la misma.

Definición y declaración de las distintas implementaciones

A continuación se muestra las definiciones de tipo y declaraciones de variables para los dos tipos de implementación de listas de enteros.

a) Lista implementada a través de un arreglo estático

```
int arre[10];
```

b) Lista implementada a través de un arreglo dinámico

```
int * arre, longitud;
```

c) Lista implementada a través de punteros

```
struct nodo
{
    int nro;
    struct nodo *sig;
};
typedef struct nodo * puntero;
```

Luego en la declaración de variables:

```
puntero cabeza;
```

Analizando la definición del registro nodo, se puede observar que es recursiva ya que uno de sus componentes (campo *sig*) es una referencia a si misma, es decir, una referencia al mismo registro.

De este modo, cada nodo se usa para construir la lista de datos, y cada uno mantendrá la relación con el siguiente. Además, para acceder a un nodo de la estructura sólo se necesita el puntero a ese nodo.

Manipulación de listas enlazadas

Las operaciones básicas necesarias para manipular listas enlazadas son:

- Creación
- Inserción
- Recorrido
- Búsqueda
- Modificación
- Supresión
- Ordenamiento
- Eliminación

Creación

La creación consiste en definir la lista vacía, utilizando la constante NULL.

Inserción en una lista

Una vez creada la lista, se puede insertar elementos en cualquier posición como se explica a continuación:

- Al comienzo de la lista: la inserción se hace al principio de ella, es decir se inserta por la cabeza de la lista. Es la opción más utilizada debido a que su codificación es muy sencilla.
- Al final de la lista: el nuevo elemento se coloca a continuación del último elemento de la lista.
- Entre otros nodos: el nuevo elemento se inserta en el medio de la lista.

Ejemplo 6

Supongamos creada una lista de números enteros (vacía), insertar una componente en la lista.

```
#include <alloc.h>
struct nodo
{
    int nro;
    struct nodo *sig;
};
typedef struct nodo * puntero;

void crear (puntero &xp)
{
    xp=NULL;
}

void insertar (puntero & xp)
{
    puntero nuevo;
    nuevo = (puntero) malloc(sizeof(struct nodo));
    printf("\n Ingrese el nuevo valor : ");
    scanf ("%d", &nuevo->nro);
    nuevo->sig = xp;
    xp = nuevo;
    return;
}

void main (void)
{
    puntero cabeza;
    crear (cabeza);
    insertar (cabeza);
    getch();
}
```

Tanto en la función `crear` como `insertar`, el parámetro formal **xp** es una referencia al parámetro actual **cabeza**, por lo tanto todo cambio producido en **xp** afecta a **cabeza**. De ahí, la variable **cabeza** guarda la dirección de la primera componente de la lista.

Ejemplo 7

Supongamos creada una lista de números enteros (vacía), insertar varias componentes. El ingreso de componentes finaliza con cero.

```
#include <alloc.h>
struct nodo
{
    int nro;
    struct nodo *sig;
};
typedef struct nodo *puntero;

void crear (puntero &xp)
{
    xp=NULL;
}

void insertar (puntero & xp)
{
    int dato;
    puntero nuevo;
    printf("\n Ingrese el nuevo valor (0 para terminar): ");
    scanf("%d",&dato);
    while (dato != 0)
    {
        nuevo =(puntero) malloc(sizeof(struct nodo));
        nuevo->nro = dato;
        nuevo->sig = xp;
        xp = nuevo;
        printf("\n Ingrese el nuevo valor (0 para terminar): ");
        scanf("%d",&dato);
    }
    return;
}

void main(void)
{
    puntero cabeza;
    crear(cabeza);
    insertar (cabeza);
}
```

La lista así creada tiene los elementos en un orden inverso a la forma en que se han introducido. Por ejemplo, para el lote: 3, 4, -8, 9, 15, 23, 0, la lista queda almacenada de la siguiente forma.

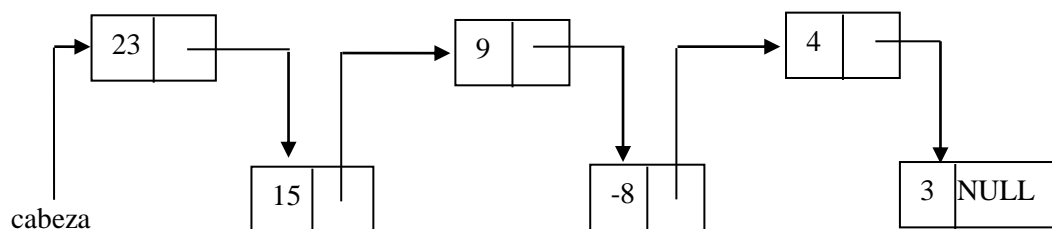


Fig. 7 Lista enlazada de enteros

ACTIVIDAD 5

Definir una función que permita crear la lista anterior, pero con sus componentes en el mismo orden en el que ingresan.

Gestión de almacenamiento

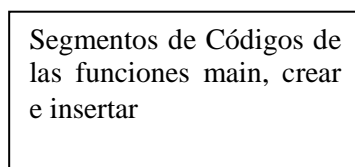
Si en un programa se declaran variables de tipo puntero, cuyo espacio de almacenamiento se asigna dinámicamente a través de la sentencia *malloc*, el sistema debe controlar no sólo el almacenamiento estático y dinámico basado en pila, sino también el almacenamiento dinámico basado en el montículo.

Para mostrar la gestión del almacenamiento, considérese el siguiente programa que genera una lista de números enteros, para el lote de prueba anterior.

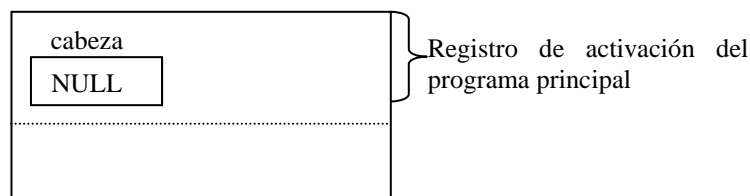
Para interpretar el estado de la memoria durante la ejecución del programa, consideraremos dos situaciones:

1) El estado en que se encuentra la memoria, antes de invocar la función **insertar(cabeza)**

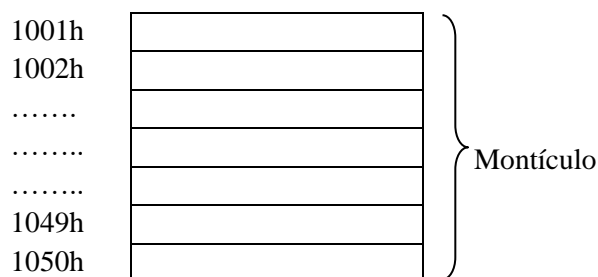
Almacenamiento estático



Almacenamiento dinámico: pila



Almacenamiento dinámico: montículo



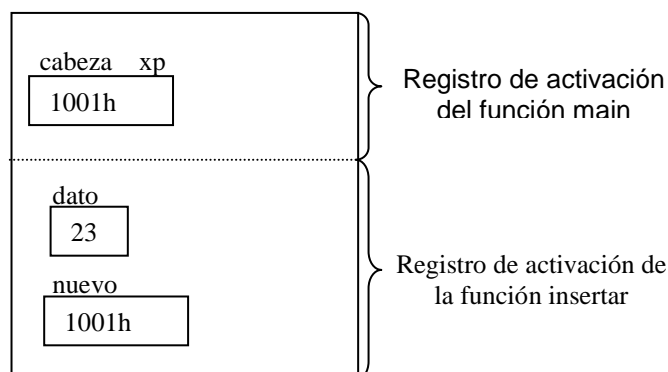
Como puede observarse el montículo está vacío, pues aún no se inició la generación de la lista. El valor NULL de cabeza, en el registro de activación del main, indica que la lista aún está vacía.

2) Una vez que **insertar(cabeza)** es invocado para generar la lista, el estado de la memoria justo antes de la sentencia return, es decir antes de transferir el control al programa principal, es el siguiente:

Almacenamiento estático:

Segmentos de Códigos de
las funciones main, crear,
insertar

Almacenamiento dinámico: pila



Nuevamente se observa que la función insertar recibe por parámetro una referencia de la variable cabeza, es decir que **xp** no es otra cosa que un alias del lugar de memoria referenciado por cabeza.

Almacenamiento dinámico: montículo

	nro	sig
1001h	23	1040h
.....		
1012h	9	1032h
.....		
1006h	4	1050h
1007h		
1032h	-8	1006h
1040h	15	1012h
1050h	3	NULL

Montículo

NOTA: la inserción en lista responde al siguiente algoritmo:

Inicio Bucle

Pedido de memoria

Carga de datos en el nuevo nodo

Enlace

Fin del Bucle

Recorrido de la lista

Para mostrar los elementos de una lista, se la debe recorrer a través de un puntero vaya apuntado sucesivamente a cada uno de los elementos, comenzando desde la cabeza, y mostrando entonces el campo de datos de cada elemento. Este proceso termina cuando se llega al final de la lista.

En general, si **p** apunta a un nodo, para avanzar al próximo se debe realizar la siguiente asignación:

p = p->sig;

Ejemplo 8

La siguiente función permite recorrer una lista como la generada en el ejemplo anterior.

```
void recorre (puntero xp)
{ printf("\n Listado de datos ");
  while (xp != NULL)
  {
    printf("\n %d ",xp->nro);
    xp = xp->sig;
  }
}
```

Cuando la función “recorre” procese la última componente de la lista, la asignación **p=p->sig;** guardará en **p** el valor **NULL** y la iteración terminará.

En el programa principal, la invocación a esta función se realiza a través de la ejecución de la sentencia:

recorre(cabeza); /*invocación a la función que recorre la lista */

NOTA

Se puede observar que los elementos de la lista se muestran en el orden inverso al que ingresaron.

NOTA: El recorrido en lista responde al siguiente algoritmo:

Inicio Bucle
 Se procesa el nodo apuntado
 Se avanza al siguiente nodo
Fin del Bucle

ACTIVIDAD 6

Analizar la función **recorre** y explicar qué resultados obtiene si el parámetro formal **xp** se pasa como una referencia.

Búsqueda de una componente de una lista

Para buscar una componente específica en la lista, se la debe recorrer desde el principio hasta encontrar la componente o hasta llegar al final de la lista si es que la componente buscada no pertenece a la misma.

Ejemplo 9

En este algoritmo, se busca un elemento en una lista enlazada.

```
#include <alloc.h>
struct nodo
{ int nro;
  struct nodo *sig;
};
typedef struct nodo * puntero;

void crear (puntero &xp)
{ xp=NULL;
}
```

```
void insertar (puntero & xp)
{ int dato;
  puntero nuevo;
  printf("\n Ingrese el nuevo valor (0 para terminar): ");
  scanf("%d",&dato);
  while (dato != 0)
  {   nuevo =(puntero) malloc(sizeof(struct nodo));
      nuevo->nro = dato;
      nuevo->sig = xp;
      xp = nuevo;
      printf("\n Ingrese el nuevo valor (0 para terminar): ");
      scanf("%d",&dato);
  }
  return;
}
```

```
void busca (puntero xp, int num)
{
  while((xp != NULL )&& (xp->nro != num))
      xp = xp->sig;
  if (xp == NULL)  printf(" El número no está en la lista");
  else printf(" Se encontró el número en la lista");
}
```

```
void main(void)
{
  puntero cabeza;
  int nn;
  crear(cabeza);
  insertar (cabeza);
  printf("\n Ingrese el número a buscar: ");
  scanf ("%d", &nn);
  busca (cabeza, nn);
}
```

La función *busca* define el parámetro **xp** por valor, para preservar el valor original de la variable que contiene la dirección del primer nodo de la lista.

Modificación de una componente de la lista

Para modificar el valor de algún dato de un nodo de la lista, se debe recorrer desde el comienzo hasta encontrar el componente correspondiente.

Ejemplo 10

En este algoritmo se busca en la lista un número leído previamente y, si se encuentra, se cambia por otro.

```
#include <alloc.h>
struct nodo
{ int nro;
  struct nodo *sig;
};
typedef struct nodo *puntero;
```

```
void crear (puntero &xp)
{   xp=NULL;
}
```

```

void insertar (puntero & xp)
{ int dato;
  puntero nuevo;
  printf("\n Ingrese el nuevo valor (0 para terminar): ");
  scanf("%d",&dato);
  while (dato != 0)
  {
    nuevo =(puntero) malloc(sizeof(struct nodo));
    nuevo->nro = dato;
    nuevo->sig = xp;
    xp = nuevo;
    printf("\n Ingrese el nuevo valor (0 para terminar): ");
    scanf("%d",&dato);
  }
  return;
}

void modifica(puntero xp)
{ int viejo, nuevo;
  printf("\n Ingrese el número que quiere cambiar ");
  scanf ("%d", &viejo);
  printf("\n Ingrese el nuevo número ");
  scanf ("%d", &nuevo);
  while ((xp != NULL) && (xp->nro != viejo))
    xp = xp->sig;
  if (xp != NULL)
    xp->nro = nuevo;
  else
    printf("\n el número no está en la lista");
  return;
}

void main(void)
{
  puntero cabeza;
  int nn;
  crear(cabeza);
  insertar (cabeza);
  modifica (cabeza);
}

```

ACTIVIDAD 7

En el ejemplo anterior, analizar el pasaje de parámetros de las funciones `modifica`. Extraer conclusiones.

NOTA: Las funciones de búsqueda y modificación responden al algoritmo señalado para recorrer la lista:

Inicio Bucle

Se procesa el nodo apuntado (si es el nodo buscado se informa o se modifica y se sale del bucle)

Se avanza al siguiente nodo (si no se encontró el buscado)

Fin del Bucle

Inserción de un nodo en cualquier lugar de la lista

Hasta ahora, las inserciones de nodos en la lista se han realizado por la cabeza. Esto es, cada nodo que se inserta pasa a ser el primero de la lista.

En muchas aplicaciones es conveniente insertar un dato en cualquier lugar de la lista, dependiendo de los requerimientos. Por ejemplo, en la lista antes creada, podría incluirse un nodo al final de la lista de enteros (ver figura 7)

Inserción de un nodo al final de una lista

Ejemplo 11

En el siguiente programa se utiliza la función **insertarAlFinal** para agregar un nodo al final de la lista generada.

```
#include <alloc.h>
struct nodo
{
    int nro;
    struct nodo *sig;
};
typedef struct nodo *puntero;

void insertarAlFinal (puntero &xp)
{
    puntero p, nuevo, anterior;
    nuevo =(puntero) malloc(sizeof(struct nodo));
    printf("\n Ingrese el nuevo valor: ");
    scanf("%d",&nuevo->nro);
    nuevo->sig = NULL;
    if (xp == NULL)          /* controla si la lista está vacía */
        xp = nuevo;
    else
    {
        p = xp;              /* resguarda a xp de las siguientes modificaciones */
        while (p != NULL)
        {
            anterior = p;
            p = p->sig;
        }
        anterior->sig = nuevo;
        printf("El nuevo elemento ha sido insertado al final de la lista.");
    }
    return;
}

void main(void)
{
    puntero cabeza;
    crear(cabeza); // ídem a anterior
    insertar (cabeza); // ídem a anterior
    insertarAlFinal (cabeza);
}
```

Inserción de un nodo cualquier lugar de la lista

Una de las ventajas del uso de listas enlazadas a través de punteros es la flexibilidad de las mismas para aumentar o reducir su tamaño. La inserción de nodos en este tipo de estructura es mucho menos compleja que en el caso de listas soportadas por arreglos.

Por ejemplo, partiendo de una lista de enteros generada como en la figura 7, supóngase que se desea insertar un nodo en la posición inmediata anterior al nodo que tiene almacenado el número 9.

Así, si en la lista se quiere insertar una componente que tenga el número 12 entonces debe crearse un nodo con el número 12 e insertarse antes del nodo que contiene a 9. Esto implica realizar el cambio de punteros necesarios de modo que el nodo que contiene al número 15 apunte al nuevo nodo generado (que contiene el número 12) y éste al nodo correspondiente al número 9.

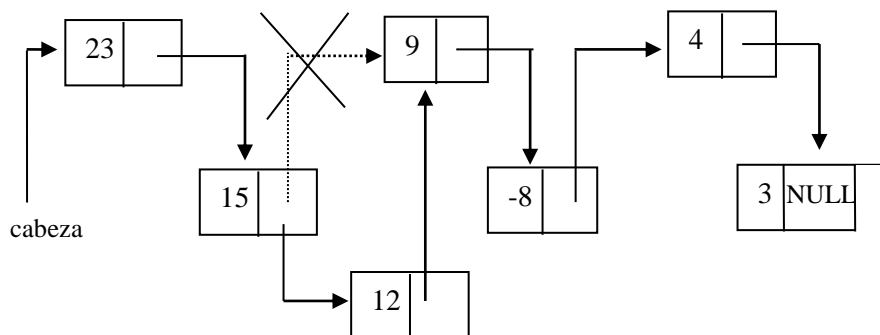


Fig. 8 Inserción del nodo que contiene el número 12

El siguiente código, resuelve la situación planteada:

void insertarAdentro (puntero &xp, int xnum)

```

{ puntero p, nuevo, anterior;
  nuevo =(puntero) malloc(sizeof(struct nodo));
  nuevo->nro=xnum;
  if (xp == NULL)          /* caso 1: la lista está vacía */
  { xp = nuevo;
    nuevo->sig = NULL;
  }
  else
  if (xp->nro == nuevo->nro) /* caso 2: la inserción debe hacerse al comienzo */
  { nuevo->sig = xp;
    xp=nuevo;
  }
  else
  { p = xp;          /* resguarda a xp de las siguientes modificaciones */
    anterior=xp;     /* caso 3: la inserción debe hacerse adentro o al final */
    while ( (p != NULL) && (nuevo->nro > p->nro))
    { anterior = p;
      p = p->sig;
    }
    anterior->sig = nuevo;
    nuevo->sig = p;
    printf("\n El elemento ha sido insertado en el lugar que corresponde");
  }
}

```

xnum: valor de la nueva componente

El programa principal es el siguiente:

```
void main (void)
{
    int num;
    puntero cabeza;
    crear(cabeza);
    insertar (cabeza);
    printf("ingrese número que desea agregar");
    scanf("%d", &num);
    insertarAdentro (cabeza, num);
}
```

ACTIVIDAD 8

Realizar un programa que mediante funciones permita:

- Generar una lista de números enteros positivos, ordenada en forma ascendente. Validar la entrada.
- Escribir un mensaje diciendo si el número del último nodo de la lista es par.

NOTA: el algoritmo de inserción en lista visto anteriormente, también responde a estas dos formas:

Inicio Bucle

Pedido de memoria

Carga de datos en el nuevo nodo

Enlace (se busca la posición en la lista o se va al final si es por cola)

Fin del Bucle

Supresión de un elemento en una lista

Para eliminar una componente en una lista, previamente se debe realizar el encadenamiento de la componente anterior a la que hay que eliminar con la siguiente a la misma.

Suponiendo la lista de enteros generada en la figura 7, la figura 9 muestra el cambio de punteros que se debe realizar para eliminar la componente que contiene el valor 9.

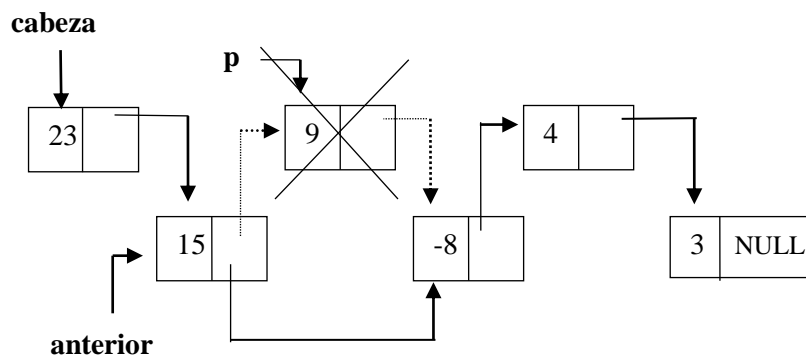


Fig. 9 Eliminación de la componente con el valor 9

Puede observarse, que para eliminar la componente, es necesario ubicar la dirección de ese nodo (**p** en el gráfico) y guardar la dirección del nodo anterior. Luego se enlaza el nodo anterior al que se desea eliminar con el siguiente al mismo. Esto es, el nodo que contiene el valor 15 debe apuntar al nodo que contiene el valor -8. Una vez realizado el cambio de punteros, debe liberarse el espacio de memoria apuntado por **p** mediante la función free.

ACTIVIDAD 9**Investigar las siguientes situaciones:**

¿Qué sucede si el número a suprimir es el primero de la lista?

¿Qué sucede si el número a suprimir aparece repetido más de una vez

El siguiente código corresponde a la función que suprime una componente de una lista enlazada.

```
void suprimir (puntero &cab)
{
    puntero anterior, p;
    int dato;
    printf("\n Ingrese el número a suprimir ");
    scanf("%d",&dato);
    if (cab->nro == dato)
    {
        p=cab;
        cab=cab->sig;
        free (p); /*eliminación de la cabeza de la lista */
    }
    else
    {
        p=cab;
        anterior=cab;
        while ((p != NULL) && (p->nro != dato))
        {
            anterior = p;
            p=p->sig;
        }
        if (p != NULL)
        {
            anterior->sig = p->sig;
            free(p);
            printf("\n El número fue eliminado de la lista");
        }
        else
            printf("\n No se encuentra el número en la lista");
        return;
    }
}
```

NOTA: el algoritmo de supresión, también responde a lógica del recorrido:*Inicio Bucle*

Se procesa el nodo apuntado (si es el nodo buscado se acomodan los enlaces, se libera el nodo y se sale del bucle)

Se avanza al siguiente nodo (si no se encontró el buscado, se resguarda la posición actual en anterior para luego acomodar los enlaces)

Fin del Bucle

Ordenamiento de una lista

Así como para arreglos, el proceso de ordenamiento por un campo determinado también puede aplicarse a listas. Se debe tener en cuenta que al momento de realizar un cambio de componentes, éste debe afectar sólo a la parte de información y no a los campos que contienen los punteros enlaces, de no ser así, se produciría un desencadenamiento de la misma.

ACTIVIDAD 10

A continuación se presenta el código de una función que ordena en forma ascendente una lista de números enteros.

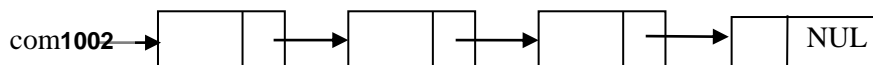
Realizar el seguimiento de la función ordena tomando como lote de prueba, una lista de cinco elementos y una lista vacía.

void ordena (puntero cab)

```
{
    puntero k, cota, p;
    int aux;

    cota = NULL;
    k = NULL;
    while (k != cab)
    {
        k = cab;
        p = cab;
        while (p->sig != cota)
        {
            if (p->nro > p->sig->nro)
            {
                aux = p->sig->nro;
                p->sig->nro = p->nro;
                p->nro = aux;
                k = p;
            };
            p = p->sig;
        }
        cota = k->sig;
    }
}
```

Eliminación de todas las componentes de una lista



Este proceso consiste en ir recorriendo la lista desde el comienzo, e ir eliminando de a una componente por vez, a medida que se avance a lo largo de ella.

Ejemplo 12

El siguiente ejemplo muestra la función que elimina todas las componentes de una lista.

void elimina (puntero & cab)

```
{
puntero p;
while ( cab != NULL )
{
    p = cab;
    cab = cab->sig;
    free (p);
}
return;
}
```

Listas enlazadas que almacenan datos estructurados

Hasta ahora se ha presentado ejemplos de listas enlazadas donde los datos almacenados son simples. A continuación se presenta un ejemplo donde los datos a almacenar son estructurados.

Ejemplo 13

Generar una lista con los datos de los empleados de una empresa: legajo y sueldo. El ingreso de información termina con legajo cero(0).

Mostrar el legajo de los empleados almacenados en la lista, cuyo sueldo es superior a 1000.

```
#include <stdio.h>
#include <alloc.h>
typedef struct
{
    int legajo;
    float sueldo;
} empleado ;

struct nodo
{
    empleado dato;
    struct nodo * sig;
};
typedef struct nodo *puntero;

void insertar (puntero &xco)
{
    puntero nuevo;
    int num;
    printf("\n Ingrese legajo: ");
    scanf ("%d", &num);
    while( num )
    {
        nuevo = (puntero) malloc(sizeof(struct nodo));
        nuevo->dato.legajo = num;
        printf("\n Ingrese sueldo : ");
        scanf ("%f", &nuevo->dato.sueldo);
        nuevo->sig = xco;
        xco = nuevo;
        printf("\n Ingrese legajo : ");
        scanf ("%d", &num);
    }
    return;
}
```

```

void listar (puntero xco)
{ printf("\n Listado de empleados con sueldo mayor de 1000 ");
  while (xco != NULL)
  {
    if (xco->dato.sueldo) > 1000
      printf("\n %d ",xco->dato.legajo);
    xco = xco->sig;
  }
}
void main(void)
{ puntero comienzo;
  comienzo=NULL;
  insertar(comienzo);
  listar(comienzo);
}

```



ACTIVIDAD 11

En el algoritmo anterior, agregar una función recursiva que para un número de legajo conocido, devuelva el sueldo. Si el empleado no está en la lista, la función devuelve 0.

Problemas de gestión de almacenamiento

El hecho de que el programador pueda gestionar memoria, esto es, pedir bloques de memoria y liberarlos cuando considere conveniente, puede crear distintos problemas. Como sabemos la función `free` se usa para liberar el espacio de memoria asignado dinámicamente, a través de la función `malloc`, a una variable. De esta manera, al finalizar el tiempo de vida de una variable, el bloque de memoria asignado a la misma se recupera para ser resignado oportunamente cuando se requiera. Sin embargo se suelen suscitar problemas tales como la generación de referencias bamboleantes o la creación de basura.

Referencias Bamboleantes

Sea `p` un puntero a una variable dinámica, una llamada a `free(p)` hace ilegal una referencia posterior a `p->`, aunque el valor de `p` queda intacto (ya que con `free` solo se modifica la Tabla de direcciones de memoria del Sistema Operativo).

No obstante algunos compiladores no detectan la ilegalidad, por lo tanto a través de `p` se puede acceder a los datos en esa dirección, provocando errores impredecibles, tales como la modificación del almacenamiento asignado previamente a otra variable dinámica. El puntero `p` recibe el nombre de referencia desactivada o bamboleante.

Por todo lo expresado, es una buena “regla de pulgar”, después de liberar el espacio ocupado por una variable dinámica colocar en `NULL` el puntero asociado a ella.

En el ejemplo anterior, la función elimina, libera el espacio ocupado por todas las componentes de la lista y por lo tanto se la recorre hasta que la cabeza de la lista (`cab`) tome el valor `NULL`.

Basura

Otra característica peligrosa asociada a los punteros y el uso incorrecto de la sentencia `free`, es la generación de basura. Se llama basura, al espacio de memoria que no puede ser accedido porque se han perdido todas las referencias a él.

Por ejemplo, supongamos tener generada una lista en la que **cabeza** es la dirección de la primer componente de ésta. Si por alguna circunstancia se hiciera **free(cabeza)**; se liberaría la memoria de la primera componente de la lista, perdiéndose la dirección del segundo nodo y, por lo tanto, todo el resto de la lista. Las celdas que siguen a la primera quedarán como basura, es decir, queda inutilizado todo el espacio de memoria ocupado por ellas. Esto puede provocar que el sistema no pueda seguir operando por falta de espacio cuando en realidad si lo tiene. En estos casos, el sistema invoca a una rutina llamada colector de basura para recuperar el espacio perdido.

NOTA

La recolección de basura (en inglés Garbage Collection - GC) es un mecanismo que proporciona recuperación de memoria automática para bloques de memoria no utilizados. El motor de GC se encarga de reconocer que ya no se usa un bloque particular de memoria asignada y lo vuelve a poner en el área de memoria libre. GC fue presentado por John McCarthy en 1958, como el mecanismo de gestión de memoria del lenguaje LISP. Desde entonces, los algoritmos GC han evolucionado. Varios idiomas se basan nativamente en GC.

Lenguaje C no incluye en forma nativa recolección de basura. Existe la biblioteca GC Boehm-Demers-Weiser (BDW), un paquete que permite a los programadores C y C ++ incluir administración automática de memoria en sus programas. La biblioteca BDW es una biblioteca de acceso libre que proporciona programas C y C ++ con capacidades de recolección de basura.

Manipulación de Listas con Funciones Recursividad

Las listas enlazadas son estructuras ideales para mostrar las ventajas del uso de funciones recursivas. Un ejemplo de ello, es la posibilidad que este mecanismo ofrece para acceder a los datos de la lista en el mismo orden en que ingresaron, para el caso de listas enlazadas generadas por inserción de nodos por la cabeza.

ACTIVIDAD 12

Analizar la siguiente función, realizar el seguimiento con el Lote de prueba: 3, 4, -8, 9, 15, 23, 0

```
#include <alloc.h>
void crear(puntero &xp)
{
    xp=NULL;
}

void listar (puntero xp)
{
    if (xp != NULL)
    {
        listar(xp->sig);
        printf("\n %d ",xp->nro);
    }
    else
        printf(" \n Listado de lista en el orden de ingreso de los datos");
}

void main(void)
{
    crear(cabeza);
    insertar (cabeza)
    listar (cabeza);
}
```

Arreglo de Listas

Ejemplo 14

La Facultad de Ciencias Exactas organizó el Congreso de Informática y necesita contar con un programa que le permita administrar la información relativa a los 10 tutoriales que se proponen en dicho evento. Para ello, el programa deberá permitir, a través de un menú de opciones, los siguientes ítems:

- Ingresar los datos correspondientes a cada tutorial, a saber: número de tutorial (1-10), título.
- Registrar las inscripciones, ingresando el DNI del inscripto y el número de tutorial al que desea asistir.
- Eliminar alguna inscripción, en cuyo caso se ingresarán los mismos datos que en el ítem anterior.
- Dado el número o el título de un tutorial, mostrar los datos específicos del mismo, incluido la cantidad inscriptos.
- Dado el DNI de una persona, informar el/los tutoriales (número y título) en los que se inscribió.

NOTA

Para resolver este ejercicio, se presentan las siguientes definiciones de las estructuras de los datos a utilizar:

struct nodo

```
{
    int dni;
    struct nodo * sig;
};
```

struct datos /* corresponde a cada tutorial */

```
{
    int numero;
    cadena titulo;
    struct nodo * ins; /* puntero a una lista de inscriptos*/
};
```

```
typedef struct nodo * inscriptos;
```

```
typedef char cadena[30];
```

void main(void)

```
{
    int i, n, op;
    long d;
    datos tutorial[fila]; /* arreglo de tutoriales */
    printf("\n ***** Cargar de los tutoriales C\n *****");
    for (i=0;i<fila;i++)
    {
        tutorial[i].numero=1;
        printf("\n Ingrese nombre del tutorial %d :",i+1);
        gets(tutorial[i].titulo);
        crear(tutorial[i]);
    }
    printf("\n ***** MENU DE OPCIONES *****\n");
    do
    {
        clrscr();
        printf("\n 1- Realizar una inscripción: ");
        printf("\n 2- Eliminar una inscripción: ");
        printf("\n 3- Mostrar Datos de un tutorial: ");
        printf("\n 4- Mostrar tutoriales en los cuales se inscribió una persona : ");
        printf("\n 5- Mostrar tutoriales con Datos de Inscriptos : ");
        printf("\n 6- Salir del MenúRealizar una inscripción: ");
    }
```

```

scanf("%d", &op);
switch (op)
{
    case 1: { printf("\n Ingrese numero del tutorial :");
              scanf("%d", &n);
              printf("\n Ingrese DNI :");
              scanf("%d", &d);
              insertar(tutorial[n-1].ins,d);
              break;
            }
    case 2: /*** Completar ***/; break;
    case 3: /*** Completar ***/; break;
    case 4: /*** Completar ***/; break;
    case 5: listar(tutorial) ;
}
} while (op !=6);
getche();
}

```

A continuación se definen las funciones necesarias para responder a la opción 1.

void crear (datos &d)

```

{
d.ins=NULL;
}

```

void insertar (inscriptos &xp, int DNI)

```

{
inscriptos nuevo;
nuevo = (inscriptos) malloc(sizeof(struct nodo));
nuevo->dni=DNI;
nuevo->sig = xp;
xp = nuevo;
return;
}

```

void listar(datos t[])

```

{
    int i;
    inscriptos xp;
    for(i=0; i< fila; i++)
    {
        printf("\n **** Datos de Inscriptos en el Tutorial:%3d ****\n", i+1);
        printf("\n TITULO: "); puts(t[i].titulo);
        printf("\n INSCRIPTOS \n ");
        xp= t[i].ins;
        while (xp!= NULL)
        {
            printf("\n %d",xp->dni);
            xp = xp->sig;
        }
    }
}

```

ACTIVIDAD 13: Escribir las funciones necesarias para dar respuesta a los otros ítems del menú principal.

Bibliografía

- Apuntes de Cátedra Programación Procedural
- Aho Alfred V., John E. Hopcroft, Jeffrey D. Ullman (1988) "Estructura de datos y algoritmos publicación". Addison-Wesley Iberoamericana: Sistemas Técnicos de Edición. México, DF.
- Braunstein y Gioia (1987) "Introducción a la Programación y a la Estructura de Datos" - Eudeba
- Criado Clavero, Mª Asunción.(2006) "Programación en Lenguajes Estructurados". Alfaomega. Rama. México.
- Deitel, H M y Deitel, P J (2003) "Como programar en C/C++" Pearson Educación – Hispanoamericana
- Joyanes Aguilar A y Zahonero Martínez (2001) "Programación en C. Metodología, estructuras de datos y objetos". Mc Graw - Hill.
- Joyanes Aguilar Luis (2004) "Algoritmos y Estructuras de Datos. Una Perspectiva en C". Segunda Edición Mc Graw Hill.

Practico 4

Estructuras Dinámicas

En los ejercicios 1 y 2, para la solución, hacer como mínimo 1 función recursiva, para el resto como mínimo hacer 2.

Ejercicio 1

Escribir un programa dados 2 vectores de N componentes enteras permita:

- Calcular el producto escalar (Realizar un subprograma de carga que sea reusable y que permita ingresar por teclado el tamaño de los arreglos).
- Generar una nueva estructura con los valores impares contenidos en uno de los arreglos (realizar un subprograma que solicite memoria para la nueva estructura y la devuelva cargada).
- Realizar el mapa de memoria con el siguiente lote de prueba, específicamente al momento de la carga de un vector, creación y carga de la nueva estructura.

vector a={ 1, 2, 3 }

vector b={4, 5, 6},

Nota

El producto escalar es una operación donde al multiplicar dos vectores se obtiene un escalar.

$$\vec{A} = (a_x, a_y), \vec{E} = (e_x, e_y) \Rightarrow \vec{A} \cdot \vec{E} = a_x \cdot e_x + a_y \cdot e_y$$

Ejercicio 2

La Federación Argentina de boxeo (FAB) mantiene información de los boxeadores federados: DNI, categoría y el peso (47..90) en kilogramos. Las categorías están codificadas por letras: 'A': Peso cruce-ro, ..., 'H': Peso minimosca. La cantidad de participantes se ingresa por teclado.

Escribir un programa en C que permita:

- Cargar los datos en una estructura adecuada. (Validar el ingreso, suponiendo que código de categoría varía entre 'A' y 'H')
- Para una categoría determinada, mostrar los DNI de los boxeadores que tienen el peso máximo. Generar una estructura auxiliar.
- Realizar un listado que muestre, para cada Peso cuantos participantes existen. Generar una estructura auxiliar.

Ejercicio 3

El Instituto Provincial de la Vivienda ha implementado un sistema que consta de 5 planes de pago distintos, con el fin de que los adjudicatarios de sus viviendas puedan cancelar sus deudas. Por cada uno de los 5 planes, se ingresa en forma ordenada la cantidad de adjudicatarios adheridos y por cada uno de ellos el DNI y monto adeudado.

Se pide:

- Cargar en una estructura de datos adecuada la información que se posee.
- Para un adjudicatario cuyo DNI se ingresa por teclado, indicar el número de plan al cual se adhirió y el monto adeudado.
- Mostrar el mapa de memoria, después de ejecutar la función que carga los datos.

Ejercicio 4

Suponiendo una lista en una estructura enlazada, no vacía, realizar el seguimiento empleando el mapa de memoria para la inserción de 4 nodos. Hacer el mismo proceso en forma recursiva.

```
void investiga (puntero &xp)
{
    puntero p, nuevo, anterior;
    nuevo =(puntero) malloc(sizeof(struct nodo));
    printf(" Ingrese nuevo valor: ");
    scanf("%d",&nuevo->nro);
    nuevo->sig = NULL;
    if (xp == NULL)
        xp = nuevo;
    else
    {
        p = xp;
        while (p != NULL)
        {
            anterior = p;
            p = p->sig;
        }
        anterior->sig = nuevo;
    }
    return;
}

void main()
{
    puntero cabeza;
    crear (cabeza);
    investiga(cabeza);
    muestra(cabeza);
    getch();
}
```

Ejercicio 5

Realizar un programa que mediante funciones recursivas permita:

- Generar una lista enlazada de números enteros positivos, ordenada en forma ascendente. Validar la entrada.
- Escribir un mensaje indicando si el número del último nodo de la lista es par.

Ejercicio 6

El Proyecto Internacional de Código de Barras de la Vida (iBOL, del inglés International Barcode of Life Project) tiene como objetivo la obtención de las “huellas genéticas” de las especies en peligro de extinción. Para ello se registra toda la fauna y flora con el fin de constituir una base de datos global que pueda ser consultada por la comunidad científica de todo el mundo. En particular se registrará información de los países de Argentina, Brasil y Estados Unidos, conociendo de los mismos: país, continente y especies. De cada especie en peligro de extinción se registra: nombre, nombre científico, reino (animal/fauna o vegetal/flora) y cantidad de ejemplares.

Realizar un programa en C que a través de funciones óptimas permita:

- Generar un arreglo de lista a través de punteros con los datos de las especies en extinción para los países indicados. El ingreso de información se encuentra ordenada por país. Para cada país el ingreso de información finaliza con el nombre de la especie igual a FIN.
- Para un nombre de país ingresado por teclado, realizar una función que devuelva al programa principal la cantidad de especies de la flora y cantidad de especies de la fauna en peligro de extinción. Realizar una función recursiva que devuelva un dato por parámetro y el otro que lo calcule la función.
- Incrementar en 200 ejemplares la cantidad de la especie con nombre Petiribí (árbol) en Brasil.
- Indicar en el programa principal cantidad de ejemplares de Petiribí (árbol presente en Argentina y Brasil) considerar solamente los ejemplares de los países indicados.

Nota: Para los distintos países se registra una sola vez las distintas especies.

Ejercicio 7

La Facultad de Ciencias Exactas organizó el Congreso de Informática, y necesita administrar la información relativa a los 10 tutoriales que se proponen en dicho evento.

Realizar un programa, que a través de un menú de opciones permita:

- Ingresar los datos correspondientes a cada tutorial: número de tutorial (1-10) y título.
- Registrar las inscripciones, ingresando el DNI del inscripto y el número de tutorial al que desea asistir.
- Eliminar alguna inscripción, en cuyo caso se ingresarán los mismos datos que en el ítem anterior.
- Dado el número de un tutorial, mostrar su título y la cantidad de inscriptos en él.
- Dado el DNI de una persona, informar el/los tutoriales (número y título) en los que se inscribió.

Nota: Para cada ítem emplear al menos una función recursiva.

Ejercicio 8

La UNSJ todos los años otorga becas, para lo cual se ingresa el número de facultades participantes, de las misma se ingresan los nombres y de cada una las inscripciones de los alumnos aspirantes a las becas de ayuda económica. Se ingresa, en forma ordenada por facultad, los datos de cada alumno: Nombre, promedio y año que cursa.

Se pide, un programa que permita:

- Realizar un listado ordenado por promedio, de los alumnos inscriptos en una determinada facultad cuyo nombre se ingresa por teclado. (Mostrar nombre del alumno, promedio y año que cursa).
- Indicar el nombre de la facultad que tiene menos inscriptos y la cantidad de inscriptos suponer único).
- Mostrar por cada facultad la cantidad de alumnos con promedio mayor o igual a 7, que cursan de segundo año en adelante. Usar una función recursiva.

Ejercicio 9

La biblioteca de la facultad cuenta con una cantidad variable de libros de Programación Procedural en calidad de préstamo en la sala de lectura que puede modificarse. Una vez prestados los libros, de los cuales se registra el código, se confecciona para cada uno una lista de alumnos que están en cola de espera. Por cada alumno se guarda: nombre y carrera (LSI, LCC)

Se pide realizar un programa, que a través de un menú de opciones y mediante el uso de funciones, de respuesta a las siguientes situaciones:

- Crear una lista de listas inicializadas en NULL. para almacenar la información de los libros
- Para un código de libro solicitado, insertar un alumno a la cola de espera correspondiente. Usar una función recursiva.
- Ingresar un nuevo libro para que esté en calidad de préstamo en la biblioteca.
- Suponiendo devuelto un libro cuyo código se lee, realizar un préstamo al primer alumno de la lista correspondiente y actualizar la misma (Esto es eliminarlo de la lista)
- Ingresar un código de libro y una carrera, mostrar los nombres de los alumnos de dicha carrera que están en cola de espera.

Unidad 6: Archivos

Introducción

Hasta ahora los tipos de datos simples y estructurados con que se ha trabajado son de almacenamiento interno. A menudo se necesita guardar datos en forma permanente. La memoria RAM, o memoria principal de la computadora es volátil, por ello, toda información contenida en ella se pierde si se apaga la computadora ya que necesita energía para mantener los datos.

Si se necesita almacenar los datos para que perduren más de una sesión de trabajo, hay que recurrir al almacenamiento secundario, que aunque más lento que la memoria principal, permite almacenarlos en **forma permanente**.

Los lenguajes de programación proporcionan estructuras que permiten el almacenamiento permanente de la información, en el caso de C se denominada FILE.

Esta estructura permite gestionar a través del sistema operativo, los medios necesarios para almacenar información en dispositivos de almacenamiento secundario en forma de archivo de datos. Entre los dispositivos de almacenamiento, secundarios o auxiliares se puede citar por ejemplo disco rígido, discos externos, pendrive, SD, CD, etc.

Los archivos son utilizados por empresas, reparticiones del gobierno, usuarios particulares, etc. ya que en todos los casos deben resguardar información que debe ser accedida cuando se necesite. Los datos a su vez deben conservar su integridad, es decir, lo que se guarda en el archivo no debe sufrir alteración, por la importancia que ellos revisten. Como ejemplos del uso de archivos se cita:

Archivos médicos con historial de pacientes.

Archivos de legajos del personal de una institución

Archivos sísmológicos que permiten guardar la actividad sísmica de una región.

Archivos que almacenan stock de mercaderías.

Archivos con notas de alumnos de una carrera.

Archivos de legislaciones provinciales y/o nacionales.

Definición

Un archivo es una estructura compuesta por datos almacenados en forma organizada en un dispositivo de almacenamiento secundario. Para su correcta manipulación, esta estructura es identificada por un nombre.

La transferencia de información entre la memoria RAM y el almacenamiento secundario se realiza a través de una unidad llamada componente.

Organización de Archivos

El término organización hace referencia a la disposición de los componentes del archivo en el dispositivo físico. Esta organización es determinada en el momento de su creación o utilización.

Organización Secuencial: Un archivo tiene organización secuencial, cuando las componentes que lo conforman se emplazan en posiciones físicas contiguas en el soporte de almacenamiento.

Organización Directa: Un archivo tiene organización directa, cuando las componentes que lo conforman ocupan posiciones físicas relacionadas mediante una clave. Esta clave puede ser un dato de la componente o bien el resultado de un cálculo matemático.

La organización adecuada para un archivo en particular está determinada por las características operacionales físicas del dispositivo de almacenamiento y por los requerimientos de la aplicación.

Clasificación de archivos

A los archivos se los puede clasificar según diferentes criterios:

Por la dirección del flujo de datos

- Archivos de entrada: el programa *lee* los datos desde el archivo.
- Archivos de salida: el programa *escribe* los datos en el archivo.
- Archivos de entrada / salida: los datos pueden ser *escritos o leídos* desde el archivo.

Según el tipo de componente

- **Archivos de texto:** En estos archivos cada símbolo se representa por un byte (ASCII, EBCDIC) y no están permitidos ciertos valores en los bytes ya que tienen un significado especial. Por ejemplo, el valor hexadecimal 0x1A marca el fin de archivo. Si se abre un archivo en modo texto, no será posible leer más allá de ese byte, aunque el archivo sea de mayor longitud.
- **Binarios:** Estos archivos son utilizados para almacenar valores enteros, en coma flotante, imágenes, archivos ejecutables etc. Están permitidos todos los valores para cada byte, los archivos binarios son secuencias de ceros y unos. Trabajar con archivos binarios optimiza el procesamiento de la información en memoria, aunque la visualización de la información debe hacerse desde el programa. En estos archivos, la detección del final del mismo, depende del soporte y del sistema operativo. Los archivos binarios se pueden trabajar de dos maneras: **con formato**, utilizando funciones que interpretan directamente el formato de cada parte de la componente y **sin formato**, utilizando funciones que toman la componente como un bloque de bytes para que el programa interprete cada parte del bloque.

Según el tipo de acceso

El término acceso hace referencia a la técnica que se utiliza para leer o grabar componentes de un archivo. Hay diversas maneras de acceder a las componentes de un archivo: en forma secuencial, directa o aleatoria, por índice y dinámico. En este curso, se considerará tan solo el acceso secuencial y el acceso directo.

- **Acceso secuencial:** Este tipo de acceso permite como única alternativa recorrer el archivo componente a componente, comenzando por la primera hasta llegar a la deseada.
- **Acceso Directo:** Este tipo de acceso permite acceder a una componente en forma directa a través de su posición en el soporte de almacenamiento secundario. Hay dispositivos que permiten acceso directo a una componente determinada, sin tener que recorrer secuencialmente los que lo preceden, tal es el caso de discos, disquete, o CD, los que permiten organización directa y/o secuencial.

Según la longitud de registro

- **Archivos de longitud variable:** Cada registro del archivo puede ser de longitud diferente. La unidad de almacenamiento que se denomina componente, es el byte. La aplicación debe conocer el tipo y longitud de cada dato almacenado en el archivo, para leer y/o escribir las componentes necesarias en cada ocasión. Por ejemplo, en el caso de archivos de texto, se usa como marca de final de registro el carácter especial salto de línea o de retorno de línea.
- **Archivos de longitud constante:** Los datos se almacenan en forma de registro de tamaño constante, es decir que la componente tiene un tamaño constante. C dispone de funciones de librería adecuadas para manejar este tipo de archivos.

Es posible crear archivos combinando las categorías mencionadas, así se puede hablar de archivos secuenciales de texto con longitud de registro variable, que son los típicos archivos de texto. Archivos de acceso aleatorio binarios de longitud de registro constante con organización secuencial, normalmente usados en bases de datos. Existen otras combinaciones menos usadas, tales como archivos con organización secuencial y de acceso secuencial binarios de longitud de registro constante, usados generalmente para cintas de resguardo o los archivos con organización directa de rápido acceso como son los índices de las bases de datos.

Ventajas y desventajas del uso de archivos

- Los datos almacenados en un archivo pueden ser usados por distintos programas, en distintos procesos y momentos.
- La capacidad de almacenamiento de la memoria secundaria es superior a la de la memoria principal, permitiendo de esta manera guardar grandes volúmenes de información.
- El uso de archivos permite tener en memoria principal sólo aquella parte de datos que necesita ser accedida por el programa en un momento dado, sin necesidad de tener toda la información en memoria principal.
- El tamaño de los archivos no es fijo y está limitado sólo por la cantidad de memoria secundaria disponible.

El acceso a los datos de un archivo es mucho más lento que los datos residentes en memoria principal. Los tiempos de procesamiento de datos de la memoria central son tiempos electrónicos (el tiempo de acceso de las memorias RAM están en el orden de los nanosegundos), mientras que el proceso de datos almacenados en periféricos incluye tiempos mecánicos (los tiempos de acceso de los disquetes están en el orden de los milisegundos).

Declaración de un archivo en el lenguaje C

Para manipular información de entrada / salida se proporciona una memoria intermedia que funciona como un canal entre el programa y el dispositivo de E/S, en esta unidad hacemos referencias solo a dispositivos de almacenamiento de datos como E/S.

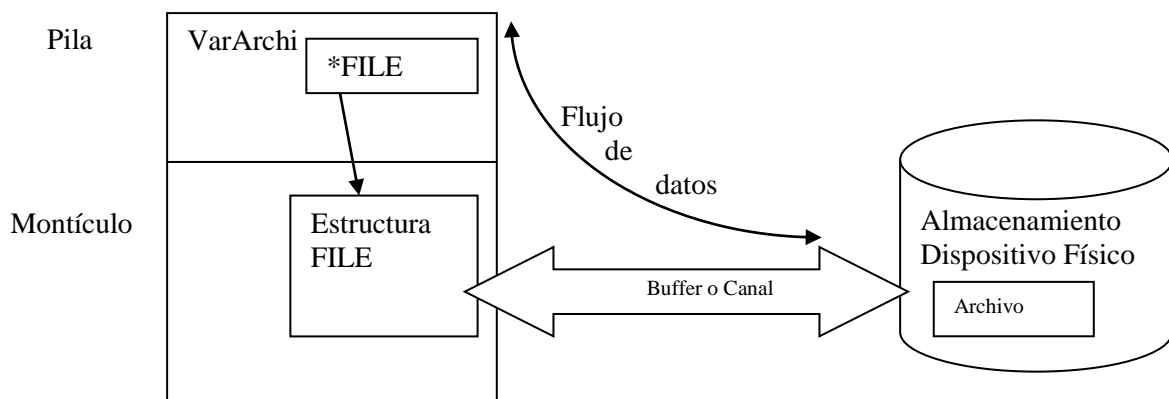
La librería stdio.h proporciona una serie de funciones y una estructura tipo FILE que contiene toda la información necesaria para utilizar un archivo.

La estructura es la siguiente:

```
typedef struct {
    short level;           //nivel de ocupacion del buffer
    unsigned flag;         //indicadores decontrol
    char fd;               //descriptor del archivo
    char hold;             // carácter de ungetc()
    short bsize;           //tamaño del buffer
    unsigned char *buffer; //puntero al buffer
    unsigned char *curp;   //posicion en curso
    short token;           //se emplea para control
} FILE;                  // tipo definido con el nombre FILE
```

Esta estructura se crea al momento en que es creado o abierto un archivo, el contenido es manipulado por las funciones de archivo que proporciona la biblioteca stdio.h. La estructura se destruye cuando se cierra el archivo.

En realidad, una variable de tipo FILE * representa un flujo de datos que se asocia con un dispositivo físico de entrada/salida (el archivo “real” estará almacenado en disco).



Para declarar un archivo se utiliza un puntero al tipo **FILE**:

```
FILE *archi;
```

La definición de la estructura **FILE** depende del compilador, pero en general mantiene un campo con la posición actual de lectura/escritura, un buffer para mejorar las prestaciones de acceso al archivo y algunos campos para uso interno, al programador le interesa su uso.

Operaciones básicas en el manejo de archivos

Para trabajar con archivos desde un programa se debe crear o abrir, según sea el caso. Para ambas situaciones, se crea un manejador lógico o un vínculo entre la variable tipo archivo y el archivo físico.

Para hacer referencia a una componente del archivo el lenguaje proporciona un apuntador de componente o un indicador de componente comúnmente llamado puntero. Al crear el archivo, está vacío y por lo tanto el puntero está en cero, a medida que se van guardando componentes, el archivo va creciendo en tamaño.

Mediante el puntero se puede acceder a distintas componentes, así si su valor es cero se apunta al comienzo de la primera componente, se puede también ubicar en las posiciones intermedias o al final del archivo, esto es al final de la última componente.

Las operaciones habituales en archivos son:

Grabar información: Consiste en guardar una componente en el archivo físico abierto.

Cerrar: Consiste en liberar el vínculo o manejador lógico que une el programa a través de la variable tipo archivo correspondiente, con el archivo físico vinculado.

Recuperar información: Para recuperar información almacenada en un archivo, éste debe primero ser abierto y luego a través de la operación de lectura, almacenar en memoria la componente para su manipulación.

Modificar información: Consiste en alterar una componente en un archivo físico.

Eliminar información: Es el proceso de quitar componentes de un archivo.

Creación y/o apertura de archivos en lenguaje C

Función fopen: Para procesar un archivo, primero se debe vincular una variable del programa con el archivo físico en el dispositivo externo, esto es lo que realiza la función `fopen`.

Formato

```
FILE * fopen (char *nombre, char *modo);
```

Esta función es utilizada para abrir y/o crear archivos en el dispositivo secundario y retorna un puntero a **FILE**.

La función `fopen` tiene dos parámetros que son cadenas de caracteres, el primero es el nombre del archivo físico que contiene los datos y el segundo corresponde al modo de acceso a éste.

nombre: es una cadena de caracteres que contiene un nombre de archivo válido.

modo: especifica la *forma* en que se va utilizar el archivo, esto es si es de lectura, escritura o para agregar información, y *el tipo* de valores permitidos a cada byte, de texto o binarios.



Este parámetro puede tomar los siguientes valores:

Modo	Apertura
“r”	Sólo lectura: el archivo debe existir
“w”	Escritura: se crea uno nuevo o se sobrescribe si ya existe.
“a”	Añadir: el archivo se abre para escritura situándose el puntero al final de éste. Si el archivo no existe, es creado.
“rb”	Lectura: abre un archivo binario
“wb”	Escritura: se crea un archivo binario nuevo o se sobrescribe si ya existe.
“ab”	Añadir: el archivo binario se abre para escritura situándose el puntero al final de éste. Si el archivo no existe, es creado.
“r+”	Lectura y/o escritura: debiendo asegurar la existencia del mismo.
“w+”	Lectura y escritura: se crea un archivo nuevo o se sobrescribe si ya existe.
“a+”	Añadir, lectura y/o escritura: el puntero se sitúa al final del archivo. Si el archivo no existe, es creado, además se dispone para lectura y escritura.

NOTA

Los tipos de archivos son de texto (modo t) o binario (modo b), si no se especifica el tipo, la opción por defecto depende del compilador utilizado, pero en general es modo texto.

El valor que retorna la función `fopen` es un puntero a `FILE` si la operación de la función es exitosa, caso contrario retorna `NULL`.

Un ejemplo es:

```
FILE *archi ;           /*declaración de la variable puntero a FILE*/
archi = fopen("Datos.dat", "w");    /*apertura del archivo Datos para escritura*/
```

Cierre de un archivo

Es importante cerrar los archivos antes de abandonar una aplicación. Esto implica almacenar datos que aún están en el buffer de memoria y actualizar aquellos datos de la cabecera del archivo que mantienen el sistema operativo. Además permite que otros programas puedan abrir el archivo para su uso. A menudo, los archivos no pueden ser compartidos por varios programas.

Función `fclose`

Formato

```
int fclose(FILE *archivo);
```

El parámetro de esta función es un puntero a la estructura `FILE` del archivo que se quiere cerrar, o `NULL` si se quiere cerrar todos los archivos abiertos.

El valor de retorno cero indica que el archivo ha sido correctamente cerrado, caso contrario indicaría que hubo un error.



Archivos organizados secuencialmente

Lenguaje C, permite que los archivos organizados en forma secuencial, puedan accederse secuencialmente o en forma directa. No obstante, hay archivos que se comportan siempre como secuenciales, por ejemplo los de entrada y salida estándar: `stdin`, `stdout`, `stderr` y `stderr`, estos son flujos o canales que se abren automáticamente.

En el caso de `stdin`, que suele estar asociado al teclado, sólo se podrá abrir como de lectura, leer los caracteres en el mismo orden en que fueron ingresados y a medida que estén disponibles. Lo mismo se aplica para `stdout`, asociado al manejo de la pantalla, sólo puede usarse para escritura y el orden en que se muestra la información es el mismo en que es enviada.

Cuando se accede secuencialmente a un archivo secuencial, hay que tener en cuenta algunas características para este tipo de acceso:

- La escritura de nuevos datos se realiza al final del archivo.
- Para leer una componente concreta hay que hacerlo secuencialmente desde el lugar donde está posicionado el puntero. Por lo tanto, es necesario "posicionarse siempre al comienzo" del archivo y recorrerlo secuencialmente hasta encontrar la componente buscada, dado que ella podría encontrarse en una posición anterior a la de la componente la cual esté apuntando en ese momento el puntero.
- Los archivos secuenciales sólo se pueden abrir para lectura o escritura, nunca de los dos modos a la vez.

Se analizarán los siguientes tipos de archivos:

- Archivos de Caracteres
- Archivos de Textos
- Archivos sin formato.



Archivos de caracteres

Los archivos de caracteres son una secuencia de caracteres almacenados en un dispositivo de almacenamiento secundario e identificado por un nombre de archivo.

Grabar información

Para grabar una componente en un archivo de caracteres se utiliza la función **fputc**. En este tipo de archivo la componente es un carácter

Función fputc: Esta función permite guardar un carácter en el archivo.

Formato

```
int fputc(int caracter, FILE *archivo);
```

Los parámetros de entrada de esta función son el carácter a escribir, convertido a `int` y un puntero al `FILE` del archivo en donde se realizará la escritura.

El valor de retorno: es el carácter convertido a `int` escrito, si la operación fue completada con éxito, caso contrario será `EOF`.

NOTA: `EOF` es una constante simbólica entera, definida en el archivo de cabecera `<stdio.h>` que indica si el final de archivo ha sido o no alcanzado. Toma el valor cero si no se ha llegado al final, y distinto de cero si se ha llegado.

Ejemplo

```
#include <stdio.h>
#include <conio.h>
void main(void)
{ FILE *archi ;                               /*declaración de la variable tipo archivo*/
  archi=fopen("archivo.dat", "w");             /*apertura del archivo para escritura*/
  char c;                                       /*declaración de la variable caracter*/
  c ='Z';                                       /*asignación del carácter 'Z' a la variable*/
  fputc(c,archi);                              /*escritura en el archivo */
}
```

Recuperar información

Para recuperar una componente de un archivo de caracteres se utiliza la función fgetc.

Función fgetc: Esta función lee un carácter desde un archivo.

Formato

int fgetc(FILE *archivo);

Parámetro: un puntero a una estructura FILE del archivo del que se hará la lectura.

El valor de retorno es el carácter leído como un unsigned char convertido a int. Si no hay carácter disponible, el valor de retorno es EOF.

Ejemplo

```
#include <stdio.h>
#include <conio.h>
void main(void)
{ FILE *archi ;                               /*declaración de la variable tipo archivo*/
  archi=fopen("archivo.dat", "r");             /*apertura del archivo para lectura*/
  char c ;                                       /*declaración de la variable caracter*/
  c=fgetc(archi);                              /*lectura del caracter del archivo */
  printf(" %c",c);
}
```

Ejemplo

El siguiente programa permite mostrar en la pantalla el código fuente del archivo ejer_escribe programa.cpp.

En este caso, se lee el archivo fuente como archivo de caracteres utilizando la función fgetc, y a medida que se lee se genera un archivo de caracteres de salida en la pantalla utilizando la función fputc.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{ char c;
  FILE *archivo;                             // declara la variable archivo de tipo file
  archivo = fopen("ejercicio_escribe programa.cpp", "r"); // abre el archivo ejercicio_escribe programa.cpp
  printf(" listado de mi programa fuente \n");           // y se posiciona al comienzo
  while (!feof(archivo))                                // itera el archivo hasta que llegue al final y envía
  { c=fgetc(archivo);                                    // los Caracteres leídos a la salida standard
    fputc(c, stdout);                                    // stdout, es el archivo de salida
  }                                                       // por defecto es el monitor
  fclose(archivo);
}
```

Archivos de textos

Un archivo de texto contiene un conjunto de “líneas” de caracteres de longitud variable y están separadas por un salto de línea.

El salto de línea está formado por dos caracteres especiales: avance de línea (line feed - carácter ASCII 10) y retorno de carro (CR carriage return - carácter ASCII 13)²⁵.

El final del archivo se indica mediante el carácter ASCII 26, que también se expresa como ^Z o EOF.

Grabar información

La función que se utiliza para guardar una componente en un archivo de textos es **fputs**. Esta función permite grabar una componente (línea) en un archivo de cadenas caracteres.

Función fputs: Esta función graba una cadena de caracteres especificada como parámetro, es decir, fputs copia la cadena terminada en null en el archivo; (No se añade el carácter de retorno de línea ni el carácter nulo final).

Formato

int fputs(const char *s, FILE *archivo);

Los parámetros son:

s: la cadena que se va a grabar en el archivo.

archivo: un puntero a FILE identificando el archivo donde se guardará la cadena.

El valor de retorno de la función es un *número no negativo*, si la escritura en el archivo fue exitosa, caso contrario, si hubo algún error devuelve la constante EOF.

Ejemplo

El siguiente ejemplo genera un archivo de textos Mi_texto.txt. Para esto, se ingresa un texto por teclado y se supone que cada línea ingresada no supera los 200 caracteres.

```
#include <stdio.h>
#include <string.h>
void main ()
{ char texto[200];
  FILE *archi;
  archi=fopen ("Mi_Texto.txt","w");
  printf("Ingrese el texto a almacenar en el archivo\n");
  if (archi!=NULL)
  {
    gets(texto);
    while(strcmp(texto,"FIN"))
    { fputs (texto,archi);
      gets(texto);
    }
    fclose(archi);
  }
}
```

NOTA: Un archivo de texto se puede visualizar utilizando un procesador de texto como Word, bloc de notas. etc.

²⁵ En una máquina de escribir convencional, para pasar a la siguiente línea se necesita dos movimientos: uno para desplazar el carro hacia la izquierda, y otro para mover el papel una línea hacia arriba. En los ordenadores se siguió esta analogía y se definieron los dos caracteres: CR para el retorno de carro, y LF para el salto de línea. Entre los dos (CRLF), se conseguía que una impresora hiciera estos movimientos para poder escribir una nueva línea de texto

Recuperar información:

La función que se utiliza para recuperar una componente en un archivo de textos es: **fgets** permite recuperar una componente (línea) de un archivo de cadenas de caracteres.

Función fgets

Esta función lee caracteres desde un archivo y los coloca en un arreglo de caracteres (variable tipo cadena). La lectura de caracteres termina cuando la cantidad de caracteres leídos alcanzó el número de caracteres especificado, o bien cuando se lea un carácter salto de línea o bien cuando no hayan más caracteres para leer, en este caso retorna NULL

Formato

char *fgets(char *s, int n, FILE * archivo);

Los parámetros son:

s: un puntero a la variable donde se guardará en memoria la cadena leída. **n:** la cantidad de caracteres que se desea leer.

archivo: un puntero a un FILE del archivo a leer.

El valor de retorno de la función es un puntero a la cadena leída o NULL si hubo algún error.

Ejemplo

```
#include <stdio.h>
#include <string.h>
void main ()
{ char texto[200];
  FILE *archi;
  archi=fopen ("Mi_Texto.txt","r");
  if (archi!=NULL)
  {
    printf("\n TEXTO INGRESADO \n ");
    while(!feof(archi))
    {
      fgets(texto,200, archi);
      puts(texto);
    }
    fclose(archi);
  }
}
```

Otras funciones para leer y grabar archivos de texto**fscanf**

La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un archivo de texto en lugar del teclado. Lee la cadena de caracteres hasta el primer espacio o salto de línea que encuentra. Permite especificar de qué tipo es el dato que se está leyendo. Una vez llevada a cabo la lectura, el puntero del archivo se sitúa en el carácter inmediatamente próximo al último espacio en blanco encontrado.

Formato

int fscanf(FILE *fichero, const char *formato, argumento, ...);

El prototipo correspondiente de fscanf es:

`int fscanf(archivo, formato, argumento, ...);`

Donde:

archivo: es la variable archivo de tipo FILE;

formato: es un especificador de formato (%d, %f,%4f,%s)

argumento: es la variable que recibe la información leída del archivo de texto, es del tipo indicado por el especificador de formato.

Ejemplo

Abrir el documento "fichero.txt" en modo lectura y mostrar su contenido en pantalla.

```
#include <stdio.h>
void lectura (FILE *fp)
{   char buffer[50];
    fp=fopen("fichero.txt","r");
    if (fp == NULL)
        printf ("Error de apertura de archivo");
    else{
        while (!feof(fp) )
        {
            fscanf(fp, "%s", buffer);
            puts(buffer);
        }
        fclose(fp);
    }
}
```

En el ejemplo se lee una cadena 50 caracteres y se muestra en pantalla, se puede en el mismo momento del `fscanf` hacer un cast, para esto, antes se deben declarar variables de los tipos de datos que se quiere obtener: modificando el ejemplo anterior donde en vez de mostrar en pantalla se cargue un arreglo de 40 componentes se obtiene el siguiente código:

```
#include <stdio.h>
typedef struct
{
    int día;
    char nombre[30];
    float sueldo;
} datos;

int lectura (FILE *fp, datos d[40])
{   int i=0;
    fp=fopen("fichero.txt","r");
    if (fp == NULL)
        printf ("Error de apertura de archivo");
    else{
        while ((!feof(fp))&&(i!=40))
        {
            fscanf(fp,"%d", &d[i].dia);
            fscanf(fp,"%s", d[i].nombre);
            fscanf(fp,"%f", &d[i].sueldo);
            i++;
        }
        fclose(fp);
        return (i);
    }
}
```

fprintf

La función `fprintf` funciona igual que `printf` en cuanto a parámetros, pero la salida se dirige a un archivo de texto en lugar de la pantalla.

Formato

int fprintf(FILE *archivo, const char *formato, argumento, ...);

El prototipo correspondiente de `fprintf` es:

```
int fprintf(archivo, formato, argumento, ...);
```

Donde:

archivo: es la variable *archivo* de tipo `FILE`;

formato: es un especificador de formato (`%d, %f,%4f,%s`)

argumento: es la variable que contiene la información que se va a escribir en el archivo de texto, es del tipo indicado por el especificador de formato.

Ejemplo

Abrir el documento "fichero.txt" en modo lectura/escritura, leer de teclado escribir en él.

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

```
void escritura (FILE *fp)
```

```
{
    char buffer[50];
    fp=fopen("fichero.txt","r+");
    if (fp == NULL)
        printf ("Error de apertura de archivo");
    else{
        gets(buffer);
        while (strcmp(buffer,"fin"))
        {
            fprintf (fp, "%s" ,buffer);
            gets(buffer);
        }
        fclose(fp);
    }
}
```

En el ejemplo se graban cadenas 50 caracteres. Se pueden usar para guardar en el archivo datos de cualquier tipo pero al momento de la escritura `fprintf` hará un cast y convertirá su contenido en tipo `char`. Modificando el ejemplo anterior donde se escribe en el archivo lo que se lee de teclado, se guardará el contenido de un arreglo de 40 componentes:

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

```
typedef struct
```

```
{
    int día;
    char nombre[30];
    float sueldo;
} datos;
```

```
void lectura (FILE *fp, datos d[40])
```

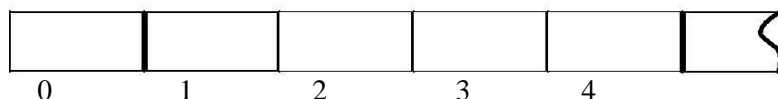
```
{
    fp=fopen("fichero.txt","r");
    if (fp == NULL)
        printf ("Error de apertura de archivo");
    else {
        for (int i=0; i < 40; i++)
        {
            fprintf (fp,"%d", &d[i].dia);
            fprintf (fp,"%s", d[i].nombre);
            fprintf (fp,"%f", &d[i].sueldo);
        }
        fclose(fp);
    }
}
```

La información grabada en el archivo será solo de tipo `char` es decir solo texto.

Archivos sin formato

Estos archivos son tratados como un bloque de datos binarios (en bytes) que tan solo tienen significado si se conoce la distribución y los tipos de datos de los mismos, es decir si se conoce la estructura.

Respecto a la generación, es importante tener en cuenta que cuando se crea un *archivo secuencial*, a través de lenguaje C, los bloques se guardan uno a continuación de otro, ocupando posiciones contiguas en el almacenamiento. Estas posiciones se numeran a partir de 0.



En este tipo de archivo el acceso puede ser secuencial o directo. El acceso directo a los datos de un archivo se hace mediante su posición, es decir el lugar que ocupa en el archivo.

Por medio de un ejemplo se estudiarán las funciones a utilizar para realizar las operaciones básicas con archivos sin formato.

Ejemplo

A través de un programa en lenguaje C, generar un archivo de nombre “alumnos.dat”, que contenga la siguiente información correspondiente a los alumnos que cursan la materia Programación Procedural: Nombre y Número de registro y resultado de un parcial- A aprobado- R Reprobado.

Construir un nuevo programa que utilice el archivo “alumnos.dat” que permita:

- 1- Realizar un listado que contenga el número de registro y nombre de todos los alumnos aprobados.
- 2- Dado el nombre de un alumno, indicar si está aprobado.

El problema puede ser resuelto codificando tres funciones:

- a) **Cargar**, que permita generar el archivo. Para ello se lo debe abrir para escritura y grabar en él las componentes.
- b) **Listar**, para generar el listado de aprobados. Para esto, se debe abrir el archivo en modo lectura y recuperar su información. Esto es, se debe recorrerlo secuencialmente, mostrando de entre las componente leídas los datos de los alumnos aprobados
- c) **Buscar**, para la búsqueda de la información correspondiente a un alumno. En esta situación, se debe abrir el archivo en modo lectura y recorrerlo secuencialmente hasta encontrar el alumno buscado o el final del archivo en caso que el mismo no se encuentre

A continuación se muestran los formatos de las funciones provistas por el lenguaje para realizar las operaciones mencionadas:

Grabar información en el archivo

Función fwrite: Esta función permite trabajar con bloques de bytes de longitud constante. Es capaz de escribir en un archivo uno o varios bloques de la misma longitud almacenados a partir de una dirección de memoria determinada.

Formato

```
size_t fwrite(void *puntero, size_t tamaño, size_t nregistros, FILE *archivo);
```

Los parámetros de la función son:

- Un puntero a la zona de memoria donde están almacenados los datos a grabar.
- El tamaño de cada bloque.
- El número de bloques a grabar.
- Un puntero a FILE del archivo correspondiente.
- El valor de retorno: Es el número de bloques escritos.

NOTA: **size_t** es un tipo de dato definido por el lenguaje que se utiliza con la función **sizeof**, el valor que puede tomar la variable del tipo **size_t** pertenece al rango de entero sin signo.

El siguiente programa resuelve el apartado a) del ejemplo.

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

typedef struct
{
    char nombre[30];
    int reg;
    char nota;
} alumno;

void cargar (FILE * archi)
{ alumno c;
  printf("\n ingrese el nombre del alumno (termina con FIN): ");
  gets(c.nombre);
  while( strcmp((c.nombre),"FIN"))
  {
      printf("\n ingrese el número de registro: ");
      scanf("%d",&c.reg);          /* Guarda en la variable tipo componente */
      printf("\n ingrese La Nota (A=aprobado R=reprobado) : ");
      fflush(stdin);
      c.nota=getch();              /* Guarda en la variable tipo componente */
      fwrite(&c,sizeof(c),1,archi); /* Guarda información en el archivo */
      printf("\n ingrese el nombre del alumno (termina con FIN): ");
      fflush(stdin);
      gets(c.nombre);
  }
}

void main(void)
{
    FILE *archi;
    char nom[30];

    if ((archi=fopen("alumnos.dat","w"))==NULL)          /* Se abre para escritura */
        printf("hay error1 \n");
    else
    { cargar(archi);          //Apartado a)
      fclose(archi);
    }
}
```

Recuperar de información desde el archivo

Función fread

Esta función está pensada para trabajar con bloques de longitud constante. Es capaz de leer desde un archivo uno o varios bloques de la misma longitud, y guardarlos en una dirección de memoria determinada.

Formato

size_t fread(void *puntero, size_t tamaño, size_t numero, FILE *archivo);

Los parámetros son:

- un puntero a la zona de memoria donde se almacenarán los datos a recuperar del archivo
- el tamaño de cada bloque.
- el número de bloques a leer.
- un puntero al FILE correspondiente.

El valor de retorno es el número de bloques leídos.

Función feof

Para recorrer de manera secuencial el archivo es necesario saber cuando se ha procesado la totalidad del mismo, para ello se cuenta con la función feof.

Formato

int feof(FILE *archivo);

El parámetro de la función es un puntero al FILE del archivo.

El valor de retorno es cero si no se ha alcanzado el fin de archivo y distinto de cero en caso contrario.

Función fflush

Para mejorar las prestaciones del manejo de archivos se utilizan buffers, almacenes temporales de datos en memoria. Las operaciones de salida se hacen a través del buffer, y sólo cuando éste se llena, se realiza la escritura en el disco efectivamente.

En ocasiones hace falta vaciar el buffer manualmente, para eso se usa la función fflush.

Esta función permite descargar los datos acumulados en el buffer del archivo.

Formato

int fflush(FILE *archivo);

El parámetro es un puntero a FILE del archivo. Si es NULL se hará el vaciado de todos los buffers asociados a los archivos abiertos.

El valor de retorno es cero si la función se ejecutó con éxito y distinto de cero si hubo error.

El siguiente es el código que permite resolver los ítems b) y c) del ejemplo.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef struct
{
    char nombre[30];
    int reg;
    char nota;
} alumno;
```

void mostrar(FILE* xarchi)

```
{ alumno c;
  rewind(xarchi);
  fread(&c,sizeof(c),1,xarchi);          //Se debe hacer una lectura anticipada
  while(feof(xarchi)==0 )
  { if(c.nota=='A')
      printf("\n Nombre Alumno %s Registro %d nota %c",c.nombre,c.reg,c.nota);
      fread(&c,sizeof(c),1,xarchi);
  }
}
```



```
void buscar(FILE *xarchi,char nom[30])
{ alumno c; int b=0;
  rewind(xarchi);
  fread(&c,sizeof(c),1,xarchi);
  while(!feof(xarchi) )&&(b==0))
  {
    if (strcmp(c.nombre,nom)==0)
    {   printf("\n Alumno %s Encontrado Registro %d nota %c",c.nombre,c.reg,c.nota);
        b=1;
    }
    else  fread(&c,sizeof(c),1,xarchi);
  }
  if(b==0)
    printf("\n Alumno %s NO se encontro",nom);
}

void main(void)
{
  FILE *archi;
  char nom[30];
  if ((archi=fopen("alumnos.dat","r"))==NULL)           /* Se abre para lectura */
    printf ("hay error1 \n");
  else
  {  mostrar(archi);           //Apartado 1
    printf("\n ingrese nombre del alumno a buscar: "); gets(nom);
    buscar(archi, nom);
    fclose(archi);
  }
}
```

Nota 1: Es posible posicionarse al comienzo del archivo a través de la función `rewind`, cuyo prototipo es: `void rewind(FILE *fp);`

Nota 2: Como la función `fread` retorna un valor distinto de cero cuando se ha leído un bloque, y cero cuando no hubo más bloques para leer (o no pudo leer), se puede recorrer el archivo secuencialmente sin usar la función `feof`, como se muestra a continuación:

```
void mostrar(FILE* xarchi )
{
  alumno c;
  rewind(xarchi);
  while(fread(&c,sizeof(c),1,xarchi))
  {   if(c.nota=='A')
      printf("\n Nombre Alumno %s Registro %d nota %c", c.nombre, c.reg, c.nota);
  }
}
```

¿Cómo optimizaría el código anterior?

Para optimizar el código realizado y no recorrer dos veces el archivo en busca de los alumnos aprobados, se puede recorrer una vez el archivo para generar una lista con los alumnos aprobados y luego recorrerla para resolver los ítems b) y c)

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
#include <malloc.h>
typedef char nombre[30];

typedef struct
{
    nombre nom;
    int reg;
    char nota;
} alumno;

struct nodo
{
    alumno datos;
    struct nodo * sig ;
};

typedef struct nodo * puntero;

void mostrar(FILE* xarchi )
{
    alumno c;
    rewind(xarchi);
    while(fread(&c,sizeof(c),1,xarchi)!=0 )
        printf("\n %5s %10d %10c",c.nom,c.reg,c.nota);
}

void generalista (FILE * xarchi, puntero &cab)
{
    alumno c;
    puntero nuevo;
    rewind(xarchi);
    while(fread(&c,sizeof(c),1,xarchi)!=0 )
    {
        if(c.nota=='A')
        {
            nuevo=( puntero) malloc(sizeof(struct nodo));
            nuevo->datos = c;
            nuevo->sig = cab;
            cab = nuevo;
        }
    }
}

void mostrarApobados( puntero xp)
{
    while (xp != NULL)
    {
        printf("\n %5s %10d %10c",xp->datos.nom, xp->datos.reg, xp->datos.nota);
        xp = xp->sig;
    }
}
```

int buscar(puntero xp, nombre nomb)

```
{ alumno c;
  while((xp!= NULL) && (strcmp(xp->datos.nom,nomb)!=0))
    xp = xp->sig;
  if (xp != NULL)
    return 1 ;
  else return 0;
}
```

void main(void)

```
{ FILE *archi;
  puntero cab;
  nombre nom;
  int b;
  if ((archi=fopen("alumnos.dat","r"))==NULL)          /* Se abre para lectura */
    printf("hay error1 \n");
  else
  { printf("\n LISTADO COMPLETO ALUMNOS, DESDE ARCHIVO \n");
    printf("\n NOMBRE REGISTRO NOTA");
    mostrar(archi);
    cab=NULL;
    generalista( archi,cab);
    printf("\n LISTADO ALUMNOS APROBADOS, DESDE LISTA\n");
    printf("\n NOMBRE REGISTRO NOTA");
    mostrarAprobados (cab);
    printf("\n ingrese nombre del alumno a buscar: "); gets(nom);
    b= buscar(cab,nom);
    if(b==1)
      printf("\n El Alumno ESTA APROBADO ");
    else printf("\n El Alumno NO está Aprobado ");
    fclose(archi);
  }
}
```

Lectura de varias componentes de un archivo

El lenguaje C permite en la función `fread` la lectura de varias componentes del archivo. El valor de retorno de `fread` es el número de componentes efectivamente leídas, el primer parámetro es la zona de memoria donde se ubicarán los datos leídos y el tercer parámetro es la cantidad de componentes que intenta leer. En el ejemplo anterior se ha realizado el procesamiento de una componente por vez. Se puede leer un grupo de componentes de una sola vez por ejemplo a un arreglo para luego ser procesadas

Ejemplo

Utilizando el archivo “alumnos.dat, mostrar en forma ordenada alfabéticamente los 20 primeros alumnos.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
const int m=20;
typedef struct
{ char nombre[30];
  int  reg;
  char nota;
} alumno;
```

```
void ordena(alumno c[m], int n)    //ordenamiento en un arreglo
{
    alumno auxi;
    int i,cota,k ;
    cota=n-1;
    k=1;
    while (k!=-1)
    {
        k=-1;
        for(i=0;i<cota;i++)
            if(strcmp(c[i].nombre,c[i+1].nombre)>0)
            {
                auxi=c[i];
                c[i]=c[i+1];
                c[i+1]=auxi;
                k=i;
            }
        cota=k;
    }
}
```

```
void muestra( alumno c[m], int n)
{
    int i;
    for (i=0;i<n; i++)//muestra el arreglo ordenado
        printf("/n",          c[i].nombre,
        c[i].reg);
}
```

```
void listar_primeros(FILE* xarchi )
{
    alumno c[m];
    int cant, i;
    rewind(xarchi);
    cant= fread(c, sizeof(alumno), m, xarchi);    /* Se realiza la lectura de 20 Alumnos*/
    printf("Se leyeron %d registros \n",cant);
    ordena(c,cant);    //Se invoca la función para ordenar el arreglo
    printf(" LISTADO %d Alumnos ordenados alfabéticamente \n");
    printf("Nombre          Registro   \n");
    muestra(c,cant);
}
```

```
void main(void)
{
    FILE *archi;
    if ((archi=fopen("alumnos.dat","r+"))==NULL)    /* Se abre para lectura y escritura */
        printf("hay error1 \n");
    else
    {
        listar_primeros(archi);
        fclose(archi);
    }
}
```

Archivos organizados secuencialmente con acceso directo

Lenguaje C, permite manipular algunos archivos organizados en forma secuencial con acceso directo. Este es el caso de archivos con organización secuencial, que tienen en la componente un elemento llamado clave, que está en relación con la posición física del almacenamiento. Esto permite la ubicación precisa de la componente en el archivo.

Cuando se accede en forma directa a un archivo de acceso directo, hay que tener en cuenta algunas características:

- La escritura de nuevos datos se realiza al final del archivo sin que se pierda la relación de la clave con la posición física de la componente.
- Para leer una componente concreta se utiliza la clave para ubicar dicha componente

Funciones que permiten acceder en forma directa a un archivo organizado secuencialmente

Las siguientes son algunas funciones que posee el lenguaje C para gestionar el acceso a distintas posiciones de un archivo:

Función fseek

Esta función sirve para ubicar el puntero en el archivo, para lectura o escritura en el lugar deseado.

Formato

int fseek(FILE *archivo, long int desplazamiento, int origen);

Los parámetros son:

un puntero al FILE del archivo que se quiere modificar,

el valor del desplazamiento, expresado en bytes. Puede ser positivo o negativo.

el punto de origen desde el que se calculará el desplazamiento.

El valor de retorno: es cero si fue exitosa la función, y un valor distinto de cero si hubo error.

El parámetro *origen* puede tomar tres posibles valores:

Nombre	Valor	Descripción
SEEK_SET	0	el desplazamiento se cuenta desde el principio del archivo. El primer byte del archivo tiene un desplazamiento cero.
SEEK_CUR	1	el desplazamiento se cuenta desde la posición actual del puntero.
SEEK_END	2	el desplazamiento se cuenta desde el final del archivo.

Por ejemplo, si se quiere posicionar el puntero en el registro anterior se puede usar la función FSEEK, en la cual el desplazamiento se calcula así :

Desplazamiento= posición_actual_puntero_archivo – tamaño_bloque

Función fgetpos

Esta función se utiliza para obtener la posición actual del puntero del archivo. Dicha posición está dada en bytes.

Formato:

int fgetpos(FILE *archivo, fpos_t *posicion);

Los parámetros son, un puntero a FILE del archivo y la ubicación del registro dentro del archivo expresada en bytes.

El valor de retorno es cero si la función se ejecutó con éxito, y distinto de cero si hubo error.

NOTA: **fpos_t** es un tipo de dato definido por el lenguaje que se utiliza para hacer referencia a las posiciones de los punteros de archivos.

Acceso a una componente cuya posición en el archivo es conocida**Ejemplo**

El siguiente código genera un archivo de productos con los siguientes datos: código de producto, descripción, precio unitario y stock. (El código de producto está ordenado en forma ascendente y consecutiva a partir de 100).

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef struct
{
    int cod;
    char descrip[30];
    float pu;
    int stock;
} prod;
```

void cargar (FILE * xarchi)

```
{
    prod p;
    fpos_t x;
    printf("\n ingrese el nombre del producto (termina con FIN): "); fflush(stdin);
    gets(p.descrip);
    while( strcmp((p.descrip),"FIN"))
    {
        fseek(xarchi,0,
        SEEK_END);
        fgetpos(xarchi,&x);
        int cod=(int)(x/sizeof(prod))+100;
        printf("El codigo del nuevo producto es: %d",cod);  p.cod=cod;
        printf("\n ingrese el stock : ");  scanf("%d",&p.stock);
        printf("\n ingrese el precio unitario : "); scanf("%f",&p.pu);
        fwrite(&p,sizeof(p),1,xarchi);
        printf("\n ingrese el nombre del producto (termina con FIN): "); fflush(stdin);
        gets(p.descrip);
    }
}
```

void mostrar(FILE* xarchi)

```
{
    prod p;
    if ((xarchi=fopen("producto.dat","rb"))==NULL)
        printf(" Error al abrir el archivo \n");
    else
    {
        while( fread(&p,sizeof(p),1,xarchi))
            printf("\n Nombre %s codigo %d Stock %d precio %f",p.descrip,p.cod,p.stock,p.pu);
    }
}
```

```
void main(void)
{ FILE *archi;
  int cod;
  if ((archi=fopen("producto.dat","w"))==NULL)          /* Se abre para escritura */
      printf("hay error1 \n");
  else
      { cargar(archi);
        fclose(archi);
        mostrar(archi);
        fclose(archi);
      }
}
```

Ejemplo

A partir del archivo de productos antes generado se pide, realizar un algoritmo en C que permita:

- a) Agregar un nuevo producto (el código se debe generar en forma automática).
- b) Dado el código de producto mostrar su stock.
- c) Mostrar la información de todos los productos.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef struct
{ int cod;
  char descrip[30];
  float pu;
  int stock;
} prod;
```

void mostrar(FILE * xarchi)

```
{ prod p;
  rewind(xarchi);
  printf("\n Nombre      codigo   Stock    precio ");
  while( fread(&p,sizeof(p),1,xarchi))
      printf("\n %20s %3d %10d %18.2f ", p.descrip, p.cod, p.stock, p.pu);
}
```

void agregar (FILE * xarchi)

```
{
  prod p;
  fpos_t x;
  printf("\n ingrese el nombre del producto: "); fflush(stdin);
  gets(p.descrip);
  fseek(xarchi,0, SEEK_END);      // se posiciona al final del archivo
  fgetpos(xarchi,&x);
  int cod=(int)(x/sizeof(prod))+100;
  printf("El codigo del nuevo producto es: %d",cod);  p.cod=cod;
  printf("\n ingrese el stock : ");  scanf("%d",&p.stock);
  printf("\n ingrese el precio unitario : "); scanf("%f",&p.pu);
  fwrite(&p,sizeof(p),1,xarchi);
}
```

```
void mostrarstock(FILE *xarchi, int pos)
{ prod p;
  rewind(xarchi);
  fseek(xarchi, pos*sizeof(prod),SEEK_SET);
  fread(&p,sizeof(p),1,xarchi);
  printf("\n Nombre producto %s  stock %d",p.descrip,p.stock);
}

void main(void)
{ FILE *archi;
  int cod, opcion;
  if ((archi=fopen("producto.dat","a+"))==NULL) // lectura con posibilidad de agregar
      printf("hay error1 \n");
  else
  { do {
      printf("\n 1. agregar un producto");
      printf("\n 2. mostrar stock de un producto");
      printf("\n 3. mostrar todos los productos");
      printf("\n 4. salir\n");
      scanf("%d", &opcion);
      switch(opcion)
      { case 1: agregar(archi); break;
        case 2:
          { printf("\n Ingrese codigo de producto para mostrar su stock "); //Apartado b
            scanf("%d",&cod);
            mostrarstock(archi,cod-100);
            break;
          } ;
        case 3: mostrar(archi);
        case 4: break;
      } // fin del switch
    } while (opcion !=4); // fin del do
  } fclose(archi);
}
```

Modificación de una componente del archivo

Primer caso: Cuando se ingresa la ubicación de una componente

En este caso el archivo está organizado secuencialmente y está ordenado en forma ascendente y consecutiva a partir de un número natural (por ejemplo, si trabajamos con el archivo “producto.dat” antes generado).

La modificación de una componente requiere un proceso que comprende

1. leer directamente la componente, a partir de la posición de la misma
2. modificar la información en la variable
3. ubicar el puntero en la posición anterior
4. grabar la variable en el archivo.

¿Por qué debe realizarse el paso 3?

Después de cada operación de lectura o escritura, el puntero del archivo se actualiza automáticamente a la siguiente posición. Por lo tanto si se necesita grabar una componente en el archivo que ha sido modificada en memoria, como el puntero está posicionado en la componente siguiente, hay que retroceder tantos bytes como el tamaño del registro. De esta manera, la componente modificada se graba en la posición que le corresponde.

Ejemplo

A continuación se muestra la función que actualiza el stock de un producto del archivo “producto.dat” cuyo código se ingresa por teclado.

void modistock(FILE *xarchi,int cod)

```
{ prod p;
  fseek(xarchi,(cod-100)*sizeof(prod), SEEK_SET); // Se ubica el puntero en la posición deseada
  fread(&p,sizeof(p),1,xarchi);
  printf("\n Nombre producto %s stock %d", p.descrip, p.stock);
  printf("\n Ingrese el nuevo stock");
  scanf("%d",&p.stock); //Se modifica el valor de la componente
  fseek(xarchi,(cod-100)*sizeof(prod), SEEK_SET); //Se ubica el puntero de nuevo
  fwrite(&p,sizeof(p),1,xarchi); //Se graba la componente en el archivo
}
```

Segundo caso: Cuando se ingresa un dato específico de una componente (no la ubicación)

El archivo está organizado secuencialmente y puede o no presentar un orden específico.

La modificación de una componente requiere un proceso que comprende

1. leer el archivo hasta encontrar la componente buscada
2. modificar la información en la variable
3. ubicar el puntero en la posición anterior
4. grabar la variable en el archivo.

Ejemplo

A continuación se muestra la función que actualiza el stock de un producto del archivo “producto.dat” cuyo nombre se ingresa por teclado.

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

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

```
typedef struct
```

```
{ int cod;
  char descrip[30];
  float pu;
  int stock;
} prod;
```

void modistock_pr_nombre(FILE *xarchi,char *n)

```
{ prod p; fpos_t x;
  rewind(xarchi);
  fread(&p,sizeof(p),1,xarchi);
  while (!feof(xarchi) && strcmp(p.descrip,n)) fread(&p,sizeof(p),1,xarchi);
  if(!feof(xarchi))
  { printf("\n Nombre producto %s stock %d",p.descrip,p.stock);
    printf("\n Ingrese el nuevo stock");
    fgetpos(xarchi,&x); //Obtiene la posición actual del puntero
    scanf("%d",&p.stock); //Se modifica el valor de la componente
    fseek(xarchi,x- sizeof(p),SEEK_SET); //Se ubica en la posición anterior
    fwrite(&p,sizeof(p),1,xarchi); //Se graba la componente en el archivo
    fclose(xarchi);
    printf("\n Cambio realizado");
  }
  else printf("\ el producto no esta en el archivo");
}
```

```
void main(void)
{ FILE *archi;
  char n[30];
  if ((archi=fopen("producto.dat","r+"))==NULL)      /* Se abre para lectura y escritura */
    printf("hay error \n");
  else {
    printf("\n Ingrese nombre del producto para Actualizar su stock "); gets(n);
    modistock_pr_nombre(archi,n);
    fclose(archi);
  }
}
```

Eliminación de información del archivo

La eliminación de registros de un archivo secuencial es complicada, debido a que no existe función de librería estándar que lo haga.

Para eliminar registros, se debe ubicar en el registro que se desea eliminar, leerlo, marcar dicha componente y grabarlo nuevamente. Luego se deberá crear un archivo auxiliar donde se graben los registros sin marcas. Finalmente se borra el archivo original y se renombra el creado recientemente.

rename

Cambia el nombre a un archivo físico de un dispositivo de almacenamiento. Cambia el nombre actual por un nuevo nombre

Formato

int rename(const char* viejo_nombre , const char* nuevo_nombre);

Los parámetros son dos cadenas, el nombre que el archivo tiene actualmente y otra cadena con el nuevo nombre para dicho archivo físico.

El valor de retorno es cero si la función se ejecutó con éxito, y distinto de cero si hubo error.

remove

Borra o elimina un archivo físico de un dispositivo de almacenamiento.

Formato

int remove(const char *nombre_archivo);

El parámetro es, una cadena nombre_archivo que contiene el nombre del archivo físico que se quiere eliminar.

El valor de retorno es cero si la función se ejecutó con éxito, y distinto de cero si hubo error.

Ejemplo

Se tiene un archivo **clientes.dat** con los siguientes datos: nombre del cliente y número de cuenta. Se pide eliminar del archivo los clientes cuyo número de cuenta se ingresa por teclado, el ingreso finaliza con número de cuenta igual a cero.

```
#include <stdio.h>
typedef struct
{   char nombre[30];
    int  cuen-
    ta;
} cliente;
```

```
void marcar (FILE *xarchi);
void compactar (FILE *xarchi, FILE *xauxi);
```

```
void main(void)
{
FILE *archi, *auxi;
if ((archi=fopen("clientes.dat","r+"))==NULL)    // Se abre para lectura y escritura
    printf("hay error \n");
else
{
    marcar(archi);                                /* Marca los registros a borrar */
    auxi=fopen("auxiliar.dat","wb");              /* crea un archivo auxiliar para grabar */
    compactar(archi,auxi);                        /* pasa los registros válidos a un archivo auxiliar */
    fclose(archi);                                /* cierra el archivo */
    fclose(auxi);                                  /* cierra el archivo */
    remove("clientes.dat");                        /* Borra el archivo físico clientes.dat */
    rename("auxiliar.dat","clientes.dat");        /*cambia el nombre */
}
}
```

```
void marcar (FILE *xarchi)
{ cliente c;
  int nro,b;
  printf("\n Ingrese un número de cuenta a eliminar: ");
  scanf("%d",&nro);
  while(nro!=0)
  {   b=0;
      fseek(xarchi, 0, SEEK_SET);
      while((b==0)&&
      fread(&c,sizeof(c),1,xarchi))
          if(nro==c.cuenta) b=1;
      if(b==1)
      {   fseek(xarchi,- sizeof(c),SEEK_CUR);
          c.cuenta= -1;                // coloca -1 en el número de cuenta a eliminar
          fwrite(&c,sizeof(c),1,xarchi);
          printf("\n El Cliente %s ha sido Marcado para Borrarse", c.nombre);
      }
      else printf("\n La cuenta no existe");
      printf("\n Ingrese un número de cuenta a eliminar: "); scanf("%d",&nro);
  }
}
```

```
void compactar(FILE *xarchi,FILE *xauxi)
{ cliente c;
  fseek(xarchi, 0, SEEK_SET);
  fread(&c,sizeof(c),1,xarchi);
  while( !feof(xarchi))
  {   if (c.cuenta!=-1)
      fwrite(&c,sizeof(c),1,xauxi);
      fread(&c,sizeof(c),1,xarchi);
  }
}
```

Nota: Estos pasos se pueden optimizar cuando son secuenciales, ya que en vez de marcar, se copian los registros a conservar en el nuevo archivo y luego este se renombra.

Cálculo de la longitud de un archivo

Ejemplo

A continuación se muestra un algoritmo para el cálculo de la longitud de un archivo.

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

```
typedef struct
```

```
{ char nombre[30];
```

```
  int cuenta;
```

```
} cliente;
```

```
void main(void)
```

```
{ long nRegistros;
```

```
  fpos_t nBytes;
```

```
  cliente cli;
```

```
  FILE *archivo;
```

```
  archivo=fopen("clientes.dat", rb)           //se abre para lectura
```

```
  fseek(archivo, 0, SEEK_END);                // Coloca el puntero al final del archivo
```

```
  fgetpos(archivo,&nBytes);                    // Tamaño en bytes
```

```
  nRegistros = nBytes/sizeof(cli);             // Tamaño en registros
```

```
  printf("La cantidad de registro en el archivo es : %d", nRegistros);
```

```
}
```

Bibliografía

- Apuntes de Cátedra Programación Procedural
- Aho Alfred V., John E. Hopcroft, Jeffrey D. Ullman (1988) "Estructura de datos y algoritmos publicación". Addison-Wesley Iberoamericana: Sistemas Técnicos de Edición. México, DF.
- Braunstein y Gioia (1987) "Introducción a la Programación y a la Estructura de Datos" - Eudeba
- Cairó Osvaldo (2006) "Metodología de la Programación. Algoritmos, diagramas de flujo y programas". 3º Edición Alfaomega. México.
- Criado Clavero, Mª Asunción.(2006) "Programación en Lenguajes Estructurados". Alfaomega. Rama. México.
- Deitel, H M y Deitel, P J (2003) "Como programar en C/C++" Pearson Educación – Hispanoamericana
- Joyanes Aguilar Luis (2004) "Algoritmos y Estructuras de Datos. Una Perspectiva en C". Segunda Edición Mc Graw Hill.

Practico 5

Archivos

Ejercicio 1

Codificar un programa en C que permita leer un archivo de caracteres cualesquiera e indique cuál de las cinco vocales del abecedario tiene mayor frecuencia. Realice un menú de opciones que permita leer:

- Archivo de caracteres con extensión .cpp
- Archivo de caracteres con extensión .dat
- Archivo de caracteres con extensión .txt

Ejercicio 2

Realizar en Lenguaje C los siguientes programas:

Programa 1:

Generar un archivo alumnosPP.dat que contiene la siguiente información correspondiente a los alumnos que cursan la materia Programación Procedural: Nombre, Numero de Registro y Resultado de un parcial ('A': Aprobado – 'R': Reprobado). La información se ingresa ordenada por Numero de Registro.

Codificar una función que permita mostrar la información de cada uno de los alumnos

Programa 2:

Generar un archivo alumnosAL.dat que contiene la siguiente información correspondiente a los alumnos que cursan la materia Algebra Lineal: Nombre, Numero de Registro y Resultado de un parcial ('A': Aprobado – 'R': Reprobado). El archivo debe estar ordenado por Número de Registro.

Codificar una función que permita mostrar la información de cada uno de los alumnos

Programa 3:

Codificar una función que permita mostrar Numero de Registro y Nombre de alumnos que aprobaron ambas materias.

Ejercicio 3

Un estudiante de la carrera de Geofísica, en desarrollo de su Trabajo Final sobre peligrosidad sísmica, desea unificar y procesar diferentes catálogos de sismos de acuerdo con ciertos criterios. Para lo cual, se proveen dos archivos correspondientes a catálogos de distintas fuentes, de una zona de estudio en particular:

Catalogo1.txt: sismos históricos registrados por el Centro Sismológico Nacional de la Universidad de Chile. La información que contiene es la siguiente:

País	Año	Mes	Día	Hora	Lat.	Long.	Prof.	Magnitud mb	Magnitud ms
------	-----	-----	-----	------	------	-------	-------	-------------	-------------

catalogo2.txt: sismos históricos registrados por el Servicio Geológico de EE. UU. La información que contiene es la siguiente:

Año	Mes	Día	Hora	Lat.	Long.	Prof.	Magnitud	Tipo de Magnitud
-----	-----	-----	------	------	-------	-------	----------	------------------

Realizar un programa en C que, utilizando subprogramas óptimos y estructuras adecuadas, permita:

Unificar ambos archivos generando un nuevo archivo con el siguiente formato:

ID	Año	Mes	Día	Hora	Lat.	Long.	Prof.	Magnitud	Tipo Magnitud
----	-----	-----	-----	------	------	-------	-------	----------	---------------

Observación:

El ID comienza en 1 y se va incrementando a medida que se agregan eventos sísmicos. La primera parte del archivo tiene que estar constituido por los sismos extraídos de catalogo1.txt y a continuación los de catalogo2.txt.

El Tipo de Magnitud almacenado en el nuevo archivo debe ser mw, por lo tanto, deben ser convertidas según lo indicado a continuación:

Si es mb entre 3.5 y 6.2, entonces $mw = 0.85 * mb + 1.03$

Si es ms entre 3.0 y 6.1, entonces $mw = 0.67 * ms + 2.07$

Si es ms entre 6.2 y 8.2, entonces $mw = 0.99 * ms + 0.08$

Dado un archivo ya unificado, cuyo nombre se ingresa por teclado (incluida la extensión), cambiar signo de la profundidad de cada evento sísmico, y guardarlo en un nuevo archivo.

Ejercicio 4

Un video club procesa diariamente el archivo “TITULOS.DAT”. Este archivo almacena la información para cada película: Código de la película (de 1 a 1500), Título, Director y Cantidad de ejemplares disponibles en alquiler. El archivo no tiene un orden particular.

Se pide realizar un programa óptimo que a través del uso de funciones genere un menú de opciones que responda a las siguientes solicitudes:

- Listar por cada película, el título y la cantidad de películas disponibles.
- Dado el código de una película, mostrar el título y la cantidad de películas disponibles.
- Para un director cuyo nombre se ingresa por teclado, listar el total de películas realizadas por el director y la cantidad de películas con menos de 2 ejemplares disponibles.
- Para este ítem, generar una estructura que almacene el código de película y cantidad de ejemplares disponibles en alquiler de cada una de sus películas.
- Utilizando esta estructura, codificar una función recursiva que devuelva al programa principal el total de películas realizadas por el director y la cantidad de películas con menos de 2 ejemplares disponibles

Nota: Antes de resolver, realice un programa que genere el archivo “TITULOS.DAT. Para eso ingrese de manera desordenada los títulos de las películas que están propuestas en el sitio

Ejercicio 5

Un video club procesa diariamente el archivo, “TITULOS.DAT”. Este archivo almacena la información para cada película: Código de la película (de 1 a 1500), Título, Director y Cantidad de ejemplares disponibles en alquiler. El archivo está ordenado por código de la película.

Se pide realizar un programa óptimo que a través del uso de funciones genere un menú de opciones que responda a las siguientes solicitudes:

- Dado el código de una película, mostrar el título y la cantidad de películas disponibles.
- Dado el título de una película, eliminarla de la lista. Luego, mostrar la cantidad de películas distintas que alquila el video club (usar longitud del archivo).
- Generar una lista con los nombres de los distintos directores y cantidad de películas de cada uno.

Ejercicio 6

Una consultora contable realiza la liquidación de haberes de los empleados de varias PYMES. Para ello posee un archivo con información de empleados “EMPLEADOS.dat” de diferentes empresas: Nombre del Empleado, Nombre de la Empresa, DNI, CUIT y sueldo neto. El archivo está ordenado por nombre de empresa.

Se pide realizar un programa que:

- a) Emita un listado ordenado por empresa con la liquidación de haberes de cada empleado. Además el listado debe incluir el total a pagar en concepto de sueldo por cada empresa.

***** LISTADO DE LIQUIDACION *****

Lista de empleados de JUMBO SA

	DNI	Nombre	Sueldo
1	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
2	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
3	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx

Total pagado por JUMBO SA es \$xxx

Lista de empleados de WALMART

	DNI	Nombre	Sueldo
1	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
2	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx

Total pagado por WALMART es \$xxx

Lista de empleados de CARREFOUR

	DNI	Nombre	Sueldo
1	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
2	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
3	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx
4	xx.xxx.xxx	xxxxxxxxxxxxxxxxxxxxxx	\$xxx

Total pagado por CARREFOUR es \$xxx

- b) Generar el archivo “EMPRESAS.dat” que almacena para cada empresa la siguiente información:
Nombre, total de empleados, total pagado en concepto de liquidación.

Unidad 7: Diseño Modular

Introducción

A menudo se necesita guardar datos en forma permanente. La memoria RAM, o memoria principal de El diseño del software se encuentra en el núcleo técnico de la ingeniería del software. Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas - *diseño, generación de código y pruebas* - que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que dé lugar por último a un software de computadora validado.

Desde que el primer programa fue dividido en (partes) módulos, los sistemas de software han tenido arquitecturas, y los programadores han sido responsables de sus interacciones a través de módulos y de las propiedades globales de ensamblaje. Los buenos desarrolladores de software han adoptado, a menudo, uno o varios modelos de arquitectura como estrategias de organización del sistema, pero utilizaban estos modelos de modo informal.

Hoy, la arquitectura de software operativa y su representación y diseño explícitos se han convertido en temas dominantes de la ingeniería de software.

La arquitectura de software de un sistema de programas es la estructura del sistema, la cual comprende los componentes del software, las propiedades de esos componentes visibles externamente, y las relaciones entre ellos.

La arquitectura no es el software operacional. Más bien, es la representación que capacita al diseñador para:

- analizar la efectividad del diseño para la consecución de los requisitos fijados,
- considerar las alternativas arquitectónicas en una etapa en la cual hacer cambios en el diseño es relativamente fácil, y
- reducir los riesgos asociados a la construcción del software.

¿Por qué es importante la arquitectura?

Se identifican tres razones clave por las cuales la arquitectura de software es importante:

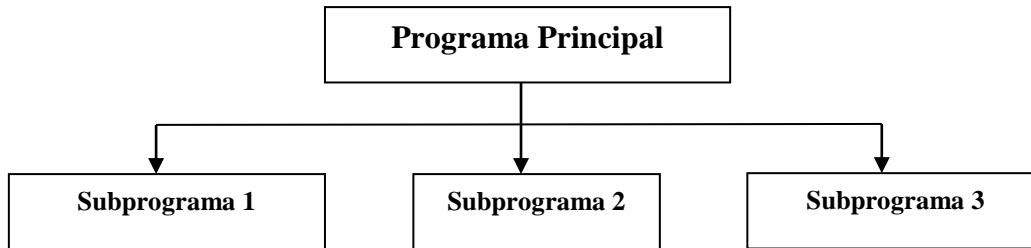
- Las representaciones de la arquitectura de software facilitan la comunicación entre todas las partes (partícipes) interesadas en el desarrollo de un sistema basado en computadora.
- La arquitectura destaca decisiones tempranas de diseño que tendrán un profundo impacto en todo el trabajo de ingeniería del software que sigue, y es tan sumamente importante en el éxito final del sistema.
- La arquitectura constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo está estructurado el sistema y de cómo trabajan juntos sus componentes.

Diseño Modular

Como hemos visto, cuando construimos la solución (programa) a un determinado problema, indirectamente hemos usado alguna técnica o metodología que nos permite una solución adecuada.

En general, para pensar en la construcción de un programa en lenguaje C hicimos uso de la técnica de diseño Top-Down, a fin de descomponer el problema en partes menos complejas. Cada una de esas partes respondía a procesos que eran codificados y agrupados en distintas funciones.

*En la fase de diseño del ciclo de vida de un programa, la solución a un problema suele venir dada por un programa representado por un módulo principal, el cual se descompone en subprogramas (submódulos), los cuales, a su vez, también se pueden fraccionar, y así sucesivamente, es decir, el problema se resuelve de arriba hacia abajo. A este método se le denomina **diseño modular** o descendente (top-down)*



¿Qué criterios tenemos en cuenta al momento de descomponer el programa, al determinar la comunicación entre las distintas partes?

Estas y otras preguntas tienen respuesta en función de la metodología usada.

En general las metodologías de diseño nos permiten realizar un modelo de solución a un determinado problema, independiente del lenguaje con el cual se implemente.

Muchas de las metodologías para diseñar algoritmos se usan en el diseño de sistemas. Entre ellas cabe citar, el Diseño Modular y Diseño Orientado a Objetos. Ambas metodologías responden a paradigmas que han marcado fuertemente la evolución de los lenguajes de programación. Estas dos metodologías pueden utilizarse en distintos contextos, a nivel de análisis de sistemas o en el ambiente de programación (en la etapa de desarrollo de software).

El Diseño Modular consiste en encontrar la solución de un problema a través de la construcción de distintos módulos o cajas negras, los cuales pueden ser codificados por distintos programadores y son compilados independientemente.

Diseño y arquitectura de software.

El diseño del software se encuentra en el **núcleo técnico de la ingeniería del software**.

Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas:

1. Diseño
2. Generación de código
3. Pruebas

Lo necesario para construir y verificar el software.

La arquitectura de software de un sistema de programas, es la estructura del sistema, la cual comprende los componentes del software, las propiedades de esos componentes visibles externamente, y las relaciones entre ellos.

Es la representación que capacita al diseñador para:

- analizar la efectividad del diseño para la consecución de los requisitos fijados, considerar las alternativas arquitectónicas en una etapa en la cual hacer;
- cambios en el diseño es relativamente fácil, y;
- reducir los riesgos asociados a la construcción del software.

La arquitectura de software es importante porque:

- Las representaciones de la arquitectura de software facilitan la comunicación entre todas las partes (partícipes) interesadas en el desarrollo de un sistema basado en computadora.
- La arquitectura destaca decisiones tempranas de diseño que tendrán un profundo impacto en todo el trabajo de ingeniería del software que sigue, y es sumamente importante en el éxito final del sistema.
- La arquitectura constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo está estructurado el sistema y de cómo trabajan juntos sus componentes.

Objetivos del diseño modular

1. Medir la calidad de un conjunto de módulos, planteados como solución de una determinada problemática.
2. Diseñar y codificar, de manera eficiente, la solución a problemas planteados.
3. Justificar, con sustento teórico, el diseño elegido para el abordaje de un problema.

Descomposición Modular

Se refiere al software cuando se divide en componentes nombrados y abordados por separado.

Los siguientes criterios permiten evaluar un método de diseño en relación con la habilidad de definir un sistema modular efectivo:

Capacidad de descomposición modular. Si un método de diseño proporciona un mecanismo sistemático para descomponer el problema en subproblemas, reducirá la complejidad de todo el problema, consiguiendo de esta manera una solución modular efectiva.

Capacidad de empleo de componentes modulares. Si un método de diseño permite ensamblar los componentes de diseño (reusables) existentes en un sistema nuevo, producirá una solución modular con componentes que ya se han construido.

Capacidad de comprensión modular. Si un módulo se puede comprender como una unidad autónoma (sin referencias a otros módulos) será más fácil de construir y de cambiar.

Continuidad modular. Si pequeños cambios en los requisitos del sistema provocan cambios en los módulos individuales, en vez de cambios generalizados en el sistema, se minimizará el impacto de los efectos secundarios de los cambios.

Protección modular. Si dentro de un módulo se produce una condición que provoque errores y sus efectos se limitan a ese módulo, se minimizará el impacto de los efectos secundarios.

Finalmente, es importante destacar que un sistema se puede diseñar modularmente, incluso aunque su implementación deba ser "monolítica" (*es decir todas las funcionalidad quedan contenidas en el mismo archivo*).

El diseño modular propone dividir el sistema en partes diferenciadas (*módulos*) y definir sus interfaces (*formas de comunicación*), sus ventajas, claridad, reducción de costos y reutilización. Los pasos a seguir son:

- Identificar los módulos
- Describir cada módulo
- Describir las relaciones entre módulos

Una descomposición modular debe poseer ciertas cualidades mínimas para que se pueda considerar suficiente adecuada (es decir validada).

- Independencia funcional
- Ocultamiento de la Información
- Acoplamiento
- Cohesión
- Comprensibilidad
- Adaptabilidad

Independencia Funcional



La independencia funcional es una propiedad deseable del diseño modular y consiste en generar módulos con una función específica que tengan baja interacción con otros módulos.

Dos módulos se dicen completamente independientes si cada uno de ellos puede completar su función sin la participación del otro.

Los módulos independientes son más fáciles de mantener y probar porque:

- Se limitan los efectos secundarios provocados por modificaciones en el código de alguno de ellos.
- Se reduce la propagación de errores.
- Es posible reutilizarlos.

Para simplificar, la **independencia funcional** es la clave del buen diseño y el diseño es la clave de la *calidad* del software.

La calidad de un software está influenciada por *factores internos* que solo afectan a los informáticos, y *factores externos* que perciben los usuarios finales. Estos factores externos son los más importantes. Los factores internos son condiciones para que se den *factores externos* favorables. Algunos factores internos que podemos considerar son:

- **Confiabilidad:** Es el factor de calidad por excelencia. Este factor hace referencia a que el sistema cumpla con las especificaciones y a la capacidad de reaccionar ante situaciones excepcionales.
Lograr estas características en un sistema basado en un diseño modular, es posible en cuanto el sistema es descompuesto en partes altamente independientes entre sí, lo que permite un "testing" más íntegro ya que cada parte es menos compleja que el todo.
Además, los errores pueden aislarse del todo, limitándolos a unos pocos subconjuntos del programa.
- **Variabilidad o Extensibilidad:** Es la medida del costo de cambiar o extender partes de un programa. Esto es, la facilidad para adaptarse a cambios de especificación. En el diseño modular se reduce el efecto "ondulante de errores" que pudieren ser provocados por cambios, puesto que éstos generalmente afectan a pocos módulos.
- **Generalidad:(Reusabilidad)** Caracterizada por el alcance que tiene la función para la cual ha sido concebido el programa.
- **Usabilidad:** Se caracteriza por la facilidad con que un software puede ser utilizado por distintos usuarios (personas de diferentes formaciones y capacidades).
- **Eficiencia:** Es la medida de rendimiento de un software, como también, su conducta. El rendimiento se mide en términos de velocidad de ejecución y espacio utilizado en memoria.

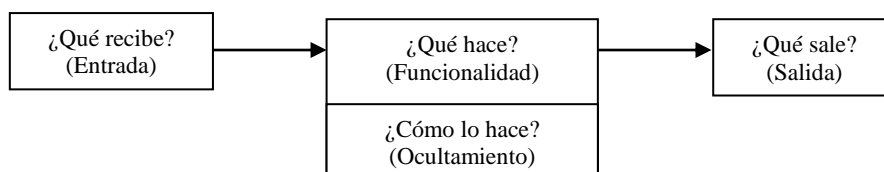
En principio, cada función será realizada en un módulo distinto. Si las funciones son independientes los módulos tendrán independencia funcional.

Cada módulo debe realizar "*una función concreta*" o "*un conjunto de funciones afines*". Es recomendable reducir las relaciones entre módulos al mínimo.

Para medir la independencia funcional hay dos criterios: acoplamiento y cohesión.

Ocultamiento de la información

Cada módulo debe ser considerado como una *caja negra*. De ella sabemos cuál es su entrada, cuál es su salida, que es lo que hace, pero no como lo hace. Esto es lo que se conoce como **principio de ocultamiento de la información**.



Cohesión

Es necesario lograr que el contenido de cada módulo tenga la máxima coherencia. Para que el número de módulos no sea demasiado elevado y complique el diseño se tratan de agrupar elementos afines y relacionados en un mismo módulo (este tema se amplía más adelante).

Acoplamiento

El grado de acoplamiento mide la interrelación entre dos módulos, según el tipo de conexión y la complejidad de la interface (este tema se amplía más adelante).

Comprensibilidad

Para facilitar los cambios, el mantenimiento y la reutilización de módulos es necesario que cada uno sea comprensible de forma aislada. Para ello es bueno que posea independencia funcional, pero además es deseable:

- *Identificación:* el nombre debe ser adecuado y descriptivo
- *Documentación:* debe aclarar todos los detalles de diseño e implementación que no queden de manifiesto en el propio código
- *Simplicidad:* las soluciones sencillas son siempre las mejores

Adaptabilidad

La adaptación de un sistema al cambio se produce cuando sus componentes tienen o se acercan a la independencia funcional, es decir, con bajo acoplamiento y alta cohesión, y el diseño es bastante comprensible.

Algunos factores que facilitan la adaptabilidad son:

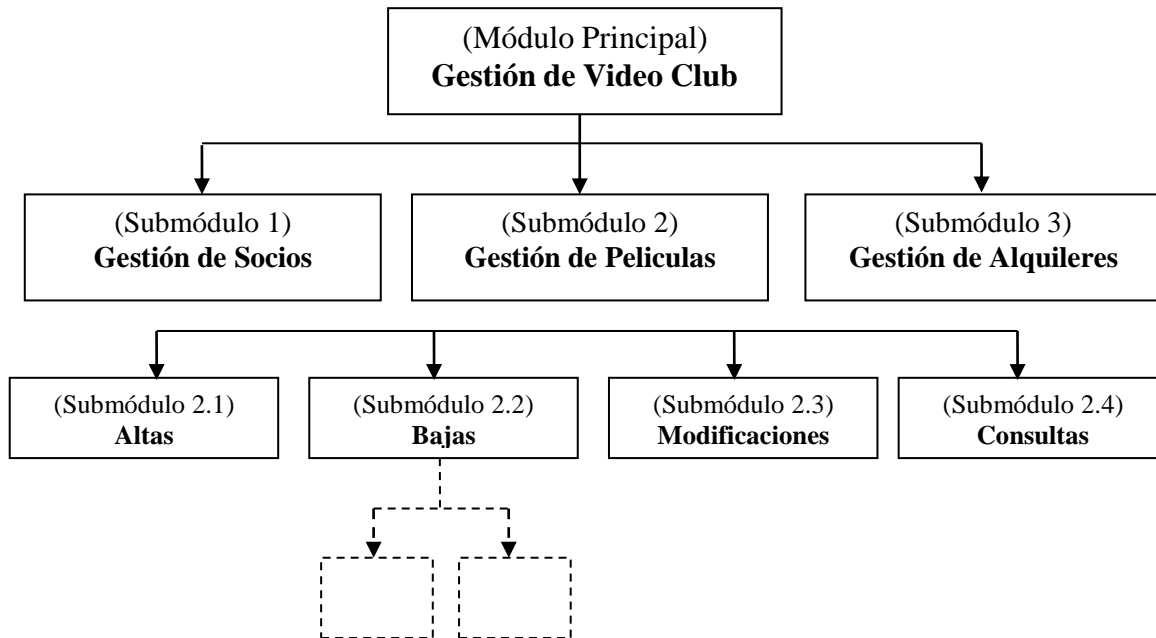
- *Previsión:* es necesario prever que aspectos del sistema pueden ser susceptibles de cambios en el futuro, y poner estos elementos en módulos independientes, de manera que su modificación afecte al menor número de módulos posible
- *Accesibilidad:* debe resultar sencillo el acceso a los documentos de especificación, diseño, e implementación para obtener un conocimiento suficiente del sistema antes de proceder a su adaptación
- *Consistencia:* después de cualquier adaptación se debe mantener la consistencia del sistema, incluidos los documentos afectados.



Modularidad

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.

Un **módulo** es un conjunto de sentencias contiguas que realizan una determinada tarea, con un nombre que lo identifica, el cual se compila independientemente y puede ser invocado desde otros programas o módulos. La siguiente grafica muestra este concepto:



El objetivo del diseño modular es lograr una descomposición del problema para obtener el **mejor** conjunto de módulos.

Algunos aspectos a tener en cuenta son:

- El problema debe ser particionado de modo que la función de cada módulo sea fácil de comprender.
- Las conexiones entre los módulos deben ser las más simples y escasas posibles.

Una vez que se distinguen los módulos, éstos se desarrollan, codifican y compilan independientemente. Luego se los organiza jerárquicamente para obtener la solución del problema general.

Se ha dicho que modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable. El software monolítico (compuesto por sólo un módulo) no puede ser fácilmente abarcado por un lector. El número de caminos de control, la expansión de referencias, el número de variables y la complejidad global podrían hacer imposible su correcta comprensión.

La modularidad se deriva naturalmente de un principio elemental para manejar la complejidad: **divide y vencerás**.

Una vez que el módulo ha sido escrito, es posible usarlo sin necesidad de interiorizarse acerca de las particularidades de su algoritmo, sólo se necesita conocer la acción que realiza y una descripción de los parámetros que maneja.

Cohesión Modular y Acoplamiento Inter-Modular

La calidad del diseño debe ser una meta para el diseñador. El diseño estructurado ofrece guías para apoyar al diseñador a determinar módulos, y sus interconexiones, que mejor realizarán los requerimientos especificados por el analista. Las dos reglas más importantes son las referentes al acoplamiento y la cohesión.

La cohesión y el acoplamiento son **criterios cualitativos** que permiten medir la independencia funcional.

Cohesión: es una medida cualitativa de refleja cuan estrechamente relacionados están las actividades internas de un módulo. Se refiere a cómo las actividades dentro de un módulo están relacionadas entre sí.

Entendemos por **actividades** a los procesos que están incluidos en el módulo. En la práctica, esto significa que el diseñador debe asegurarse de no fragmentar los procesos esenciales en módulos, y también debe asegurarse de no juntar procesos no relacionados en módulos sin sentido.

La cohesión modular puede verse como el cemento que amalgama a los elementos de procesamiento dentro de un mismo módulo. Es el factor más crucial y el de mayor importancia en un diseño modular efectivo.

La cohesión representa la técnica principal que posee un diseñador para mantener su diseño lo más semánticamente próximo al problema real, o dominio de problema.

El principio de cohesión puede ponerse en práctica con la introducción de la idea de un *principio asociativo*.

En la decisión de poner ciertos elementos de procesamiento en un mismo módulo, el diseñador, utiliza el principio de que ciertas *propiedades* o *características* relacionan a los elementos que las poseen. Debe tenerse en mente que la cohesión se aplica sobre todo el módulo, es decir sobre todos los pares de elementos. Así, si Z está relacionado a X e Y, pero no a A, B, y C, los cuales pertenecen al mismo módulo, la inclusión de Z en el módulo, redundará en baja cohesión del mismo.

Acoplamiento: es una medida cualitativa que refiere al grado de independencia entre módulos. En la práctica indica el grado de conocimiento que un programador necesita acerca de otro módulo.

La independencia funcional óptima se logra cuando se maximiza la relación entre los elementos de un mismo módulo (máxima cohesión) y se minimiza la relación entre distintos módulos (mínimo acoplamiento).

Queremos mínimo acoplamiento y máxima cohesión porque:

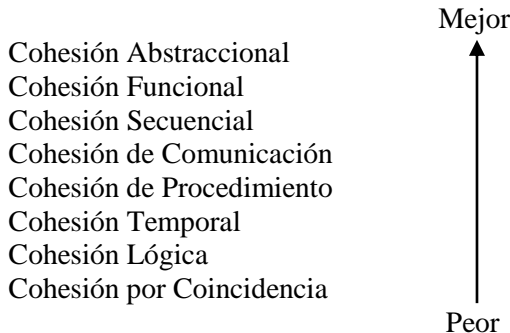
- Las bajas conexiones entre dos módulos hacen menor el chance de propagación de errores.
- Podemos querer ser capaces de cambiar un módulo, con el mínimo riesgo de tener que cambiar otro módulo, y que cada cambio del usuario afecte a tan pocos módulos como sea posible.
- Queremos que un programa sea tan simple y comprensible como sea posible.
- Una manera de asegurar el menor acoplamiento entre módulos es asegurando que todos los módulos tengan buena cohesión.



Cohesión Modular

Diferentes principios asociativos fueron desarrollándose a través de los años por medio de la experimentación, argumentos teóricos, y la experiencia práctica de muchos diseñadores.

Existen siete niveles de cohesión distinguibles por siete principios asociativos. Estos se listan a continuación en orden decreciente del grado de cohesión, de mayor a menor relación funcional:



Dentro del "espectro" mostrado, siempre se intenta conseguir una mejor cohesión, aunque un punto medio del rango del espectro es frecuentemente aceptado. La escala de cohesión no es lineal, esto es, una cohesión baja es mucho peor que la del rango medio, la cual es casi tan buena como la mejor.

La cohesión será analizada en función del flujo de datos y el flujo de control.

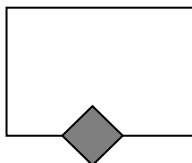
- **Flujo de datos:** es cuando hay transferencia de datos entre las actividades que constituyen el módulo y
- **Flujo de control:** es cuando interesa el *orden* de ejecución de las actividades que constituyen el módulo. Cuando se implementa el diseño, una actividad o tarea equivale a un subprograma (en lenguaje C es una función).

Notación Gráfica:

En los ejemplos siguientes se usará para la representación de los módulos los siguientes elementos:



Rectángulo: representa un Módulo. Puede tener escrito en su interior el nombre, las actividades (tareas o procesos) que realiza. Si se escriben ambas cosas se colocan una línea divisoria, dejando en la parte superior solo el nombre del módulo.



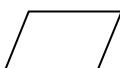
Rectángulo con selector: representa un módulo que opcionalmente se conecta con otros.



Flecha con conector blanco: representa datos usados para realizar una tarea.



Flecha con conector negro: representa un control o dato pasado como una opción.



Rectángulo inclinado: representa un archivo, en su interior se puede escribir el nombre del mismo.

Cohesión por Coincidencia

Un módulo tiene cohesión por coincidencia cuando existe poca o ninguna relación entre las actividades que realiza.

Ejemplo

Irónicamente puede ilustrarse este tipo de cohesión con el siguiente ejemplo. Supongamos tener un programa de 20.000 líneas y se lo quiere descomponer en módulos de 2.500 líneas aproximadamente, una forma de encontrar los módulos sería tomar un lápiz y una regla y cada 2.500 líneas dibujar una línea en el programa fuente. Cada uno de los lotes de sentencias constituiría módulos **con cohesión por coincidencia**.

En general esta cohesión puede resultar a partir de dos situaciones diferentes, a saber:

- Cuando un programa es desdoblado en partes, sin tener en cuenta que función cumple los elementos que integran cada parte. Por ejemplo, cuando un programa es particionado para usar menos cantidad de memoria.
- Cuando cada módulo es creado porque se quiere evitar la duplicación de una secuencia de instrucciones que son utilizadas en diferentes lugares del programa.

Nota: Identifique si hay flujo de control y flujo de datos en este tipo de cohesión.

Cohesión Lógica

Un módulo con cohesión lógica es aquel que realiza un conjunto de tareas relacionadas, eligiendo una sola de ellas cada vez que es llamado. Las tareas están relacionadas por el hecho de pertenecer a la misma clase o categoría.

Los elementos de un módulo están *lógicamente* asociados si puede pensarse en ellos como pertenecientes a la misma clase lógica de funciones, es decir aquellas que pueden pensarse como juntas lógicamente.

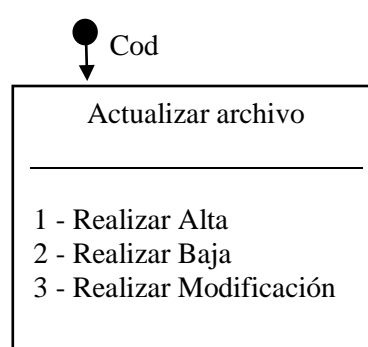
La cohesión lógica es más fuerte que la de coincidencia, debido a que *representa un mínimo de asociación entre el problema y los elementos del módulo*. Sin embargo podemos ver que un módulo lógicamente cohesivo no realiza una función específica, sino que abarca una serie de funciones. Esto implica tener un código compartido, que degrada los propósitos de un buen diseño.

Ejemplo

Un módulo que realice las operaciones de actualización de un archivo (altas, bajas, modificaciones), en función de la opción elegida.



Representación del módulo



Representación del módulo
con más refinamiento

En esta cohesión hay que conocer la lógica interna del módulo, por lo tanto se pierde el concepto de caja negra y su reusabilidad es muy limitada.

Nota: Identifique si hay flujo de control y flujo de datos en este tipo de cohesión.

En síntesis:

La cohesión por coincidencia se da en aquellos módulos cuyos elementos contribuyen a actividades con ninguna relación significativa entre ellas.

Un módulo de este tipo es similar a uno lógico: sus actividades no están relacionadas ni por el flujo de datos ni por el flujo de control.

Sin embargo, las actividades de un módulo lógico están al menos en la misma categoría lo que no ocurre en los módulos con cohesión por coincidencia.

Ambos son similares: ninguno posee función bien definida (no se puede decir con claridad que hace). Ambos son cajas blancas.

Cohesión Temporal

La cohesión temporal hace referencia a que todos los elementos de procesamiento de un módulo ocurren en el mismo período de tiempo durante la ejecución del sistema. Debido a que dicho procesamiento debe o puede realizarse en el mismo período de tiempo, los elementos asociados temporalmente pueden combinarse en un único módulo que los ejecute en un momento determinado.

Un módulo tiene Cohesión Temporal si sus elementos están involucrados en actividades cuya relación principal es el tiempo.

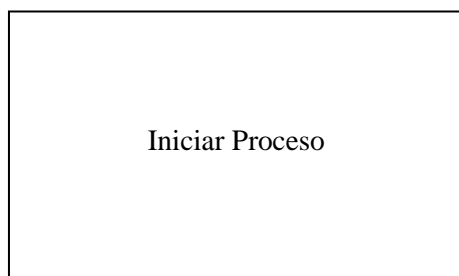
Un ejemplo común de cohesión temporal son las rutinas de inicialización (start-up) comúnmente encontradas en la mayoría de los programas, donde se leen parámetros de control, se abren archivos, se inician variables contadores y acumuladores, etc.

Nota: Identifique si hay flujo de control y flujo de datos en este tipo de cohesión.

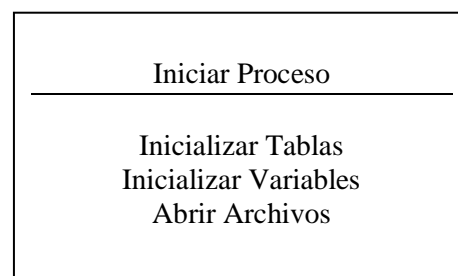
La cohesión temporal es más fuerte que la cohesión lógica, ya que implica un nivel de relación más, pues todos los elementos del módulo son ejecutados en el mismo periodo de tiempo, mientras que en la cohesión lógica sólo uno de ellos se ejecuta durante la ejecución del sistema. Sin embargo la cohesión temporal aún es pobre en nivel de cohesión y acarrea inconvenientes en el mantenimiento y modificación del sistema.

Ejemplo

Módulo Iniciar Proceso



Representación del módulo



Representación del módulo con más refinamiento

Los módulos con cohesión temporal tienden a mantener relaciones fuertes con otros módulos de un mismo programa. Por ejemplo, el módulo INICIO del ejemplo anterior tiene una relación obvia con el módulo AGREGAR UNA ENTRADA EN LA TABLA. En la organización de los módulos, primero debe estar presente el módulo INICIO y luego el AGREGAR UNA ENTRADA EN LA TABLA.

Por otra parte, debido a que este módulo realiza actividades específicas, su reusabilidad sólo suele estar presente en el contexto del sistema en el cual se define.

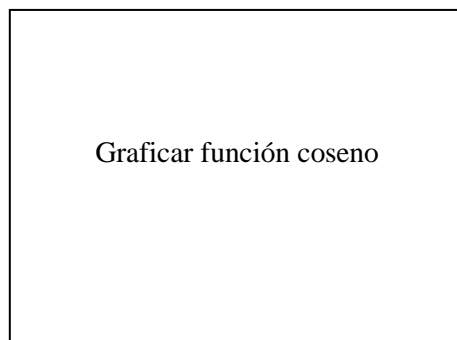
Cohesión de Procedimiento

Un módulo con cohesión de procedimiento es aquel cuyos elementos están implicados en actividades diferentes y posiblemente no relacionadas, en el cual el control fluye de cada actividad a la siguiente. Es decir, si la única relación existente entre las actividades del módulo es el orden específico de ejecución, la cohesión es procedimental.

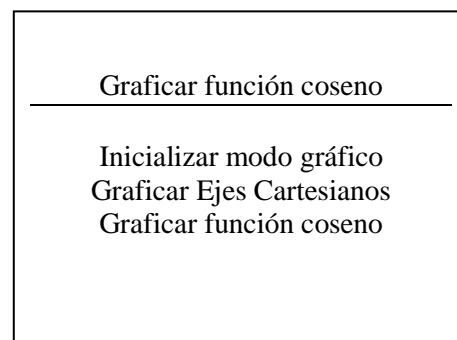
Los módulos temporales y procedimentales son similares en cuanto se ejecutan todas las funciones que constituyen el módulo, la diferencia es que el orden de ejecución de las actividades solo es importante en los módulos con cohesión procedimental.

Ejemplo

Graficar función coseno.



Representación del módulo



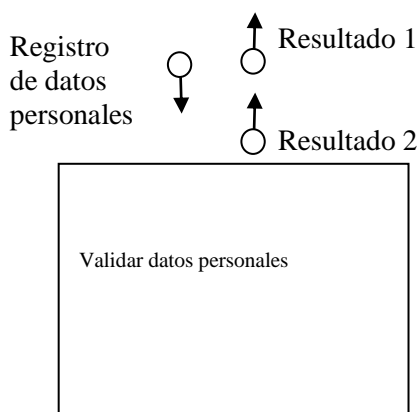
Representación del módulo con más refinamiento

Cohesión de Comunicación

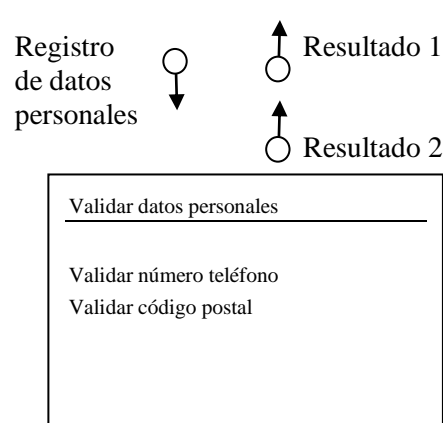
Un módulo tiene cohesión de comunicación cuando las actividades del mismo utilizan un área de una misma estructura de datos²⁶. En este caso, no hay orden impuesto.

Ejemplo 4

Validar datos personales



Representación del módulo





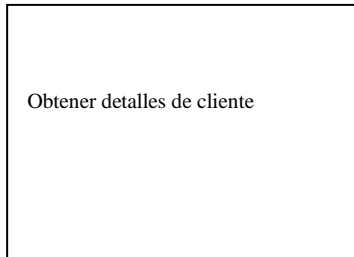
Representación del módulo con más refinamiento

²⁶ Presman, Ingeniería de Software.



Ejemplo

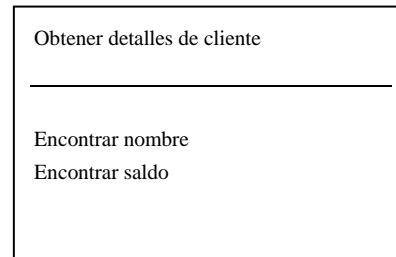
Obtener información de cliente.

Número de  de  Nombre
cuenta Saldo



Representación del módulo

Número de  de  Nombre
cuenta Saldo



Representación del módulo con más refinamiento

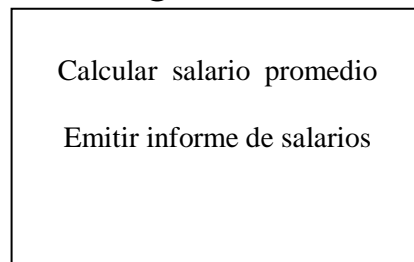
Ejemplo

Calcular salario promedio y emitir informe de salarios por categoría

Tabla de salarios de
empleados



Salario promedio



Nota: Identifique si hay flujo de control y flujo de datos en este tipo de cohesión.

Algunos Inconvenientes

Los módulos con cohesión de comunicación son fáciles de mantener, aunque pueden traer problemas. Supongamos en el ejemplo antes expuesto, que dentro del mismo sistema hay otro módulo que necesita validar número de teléfono, pero no necesita validar el código postal. Ese módulo, o bien necesitará descartar la validación del código postal (acoplamiento innecesario), o necesitará un módulo independiente para validar número de teléfono (duplicación de la función).

Otro inconveniente es que el programador se ve tentado de compartir el código de las distintas tareas que realiza el módulo. En el módulo del ejemplo anterior, se podría realizar las dos tareas dentro de un mismo proceso iterativo. Luego de un tiempo puede que se necesite un informe solo de las diez primeras categorías lo cual implicaría modificaciones de código más complicadas que las que se hubiese necesitado si el código no hubiere sido compartido.

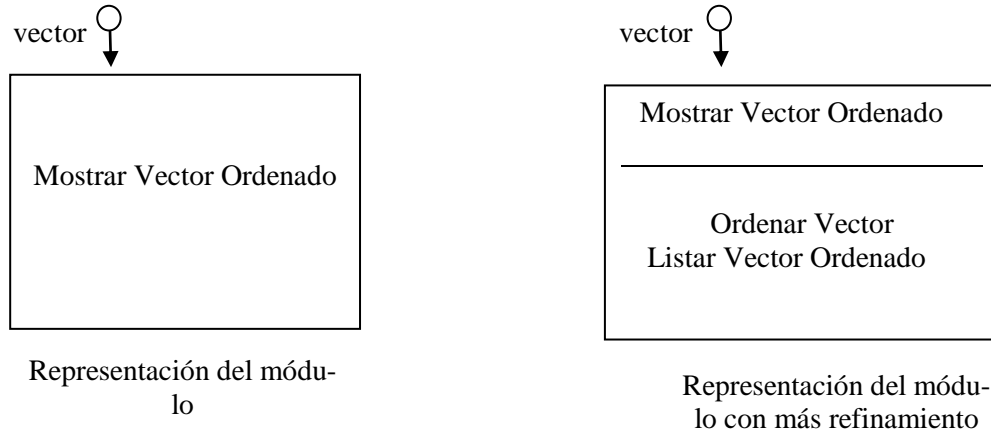
Cohesión Secuencial

Si las actividades del módulo están implicadas en procesos tales que, los datos de salida de una actividad sirven como entrada para la siguiente, el módulo presenta una cohesión secuencial.

Un módulo con cohesión secuencial usualmente posee poco acoplamiento con otros módulos. Una ligera desventaja de la cohesión secuencial es que es de baja reusabilidad, ya que las actividades en general, no son útiles juntas.

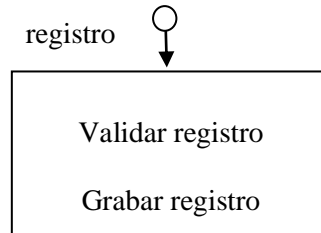
Ejemplo

Dado un vector, mostrarlo en forma ordenada.



Ejemplo

Dado el registro de un alumno, grabarlo en caso de ser es válido



En los módulos secuenciales hay flujo de control, y lo más importante es que también tiene flujo de datos, los datos fluyen de una actividad a otra, esto es que los datos de salida de una actividad son la entrada de la siguiente.

En comparación con los módulos con cohesión procedimental, se puede decir que ambos tienen flujo de control y la diferencia está en que, la cohesión secuencial tienen flujo de datos.

Cohesión Funcional

Un módulo tiene cohesión funcional si realiza una actividad específica. En un módulo funcional cada elemento de procesamiento, es parte integral de, y esencial para, la realización de una función simple. Los elementos del módulo trabajan juntos con un mismo fin.

Los ejemplos más claros y comprensibles provienen del campo de las matemáticas. Un módulo para realizar el cálculo del factorial ciertamente será altamente cohesivo, y probablemente, completamente funcional. Es improbable que haya elementos superfluos más allá de los absolutamente esenciales para realizar la función matemática.

Cuando se agrupan unidades de software teniendo en cuenta que todas ellas contribuyen a realizar un mismo fin. Es decir, cuando todas las unidades agrupadas, trabajan juntas para alcanzar un objetivo muy específico.

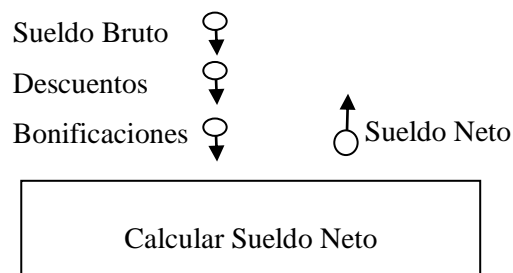
Un módulo funcional contiene elementos que contribuyen a la realización de una tarea simple. En ellos se observa que son módulos más reutilizables ya que solo es necesario conocer la función que cumplen (lo que hacen), el nombre de este tipo de módulos suele indicar claramente la función que realizan, por ejemplo:

- Calcular suma
- Validar campo numérico
- Calcular seno del ángulo

Este tipo de cohesión representa el mayor grado de independencia que puede tener un módulo. Por todo esto es la cohesión más deseada.

Ejemplo

El siguiente módulo calcula el sueldo neto de un empleado de una fábrica.



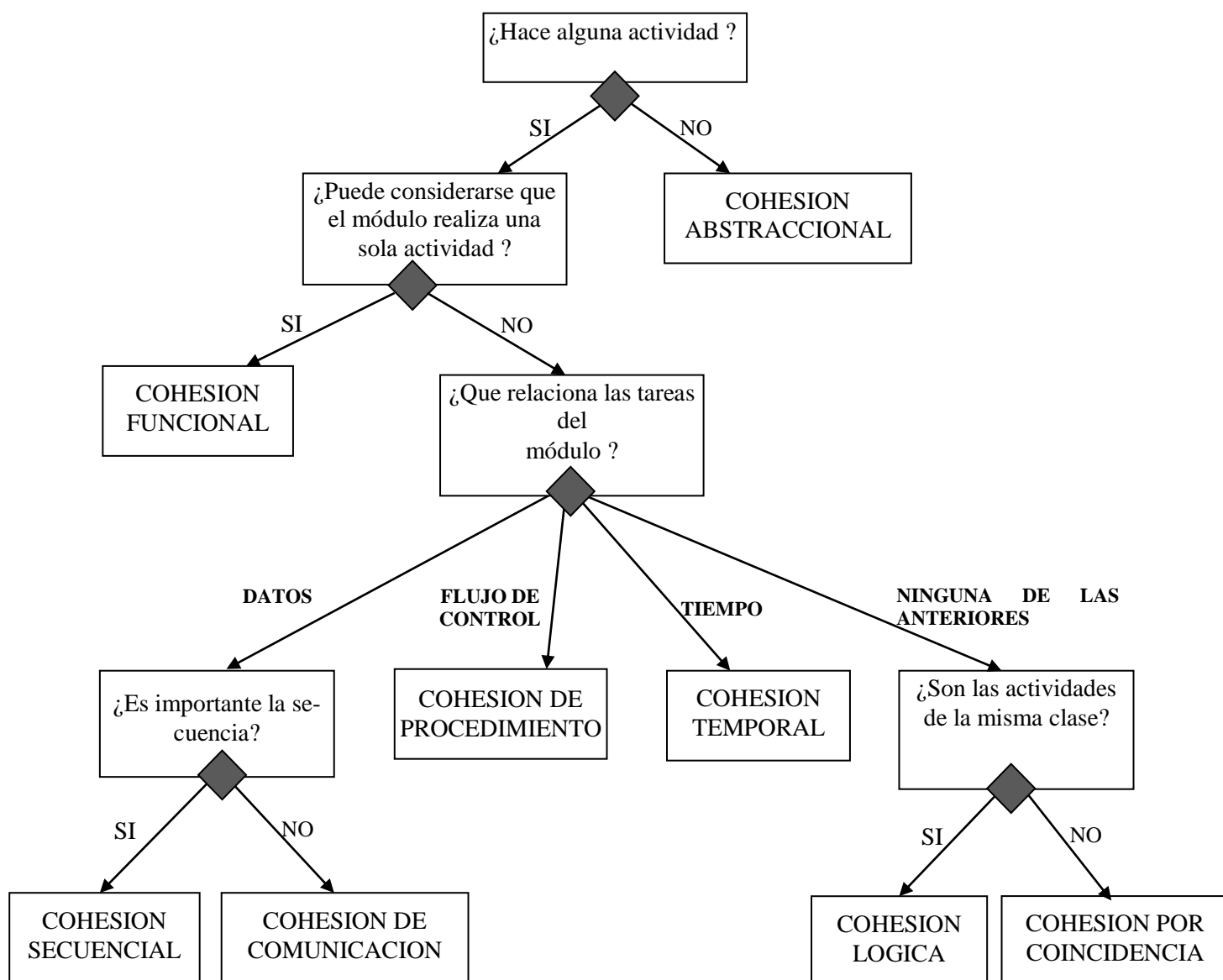
Cohesión Abstraccional

Se logra cuando se diseña el módulo como tipo abstracto de datos. Hace referencia a un módulo de definición de tipos usados por el sistema.

```

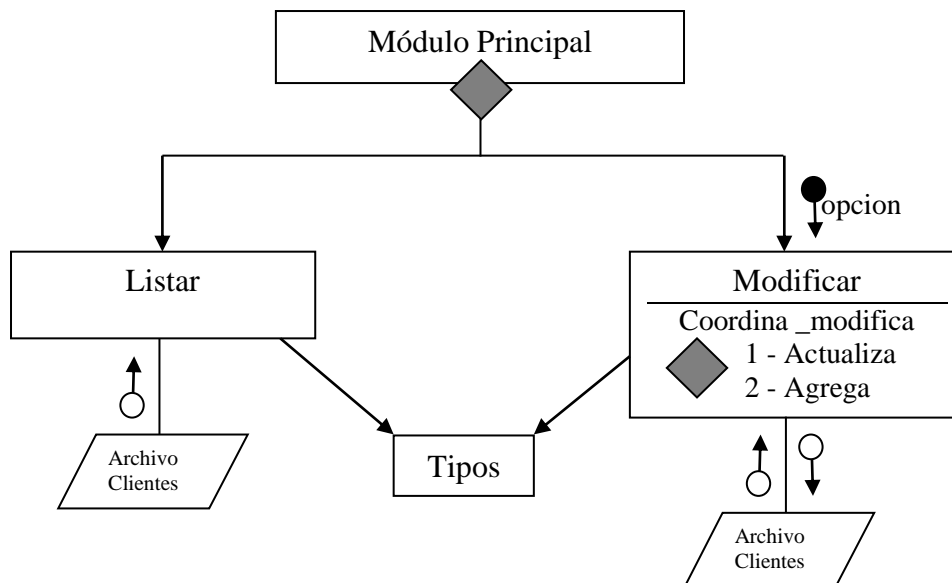
typedef struct
{
    int codigo;
    char descripcion[30];
    float precio;
    int stock;
} prod;

struct nodo
{
    prod producto;
    struct nodo *sig;
};
typedef struct nodo * puntero;
  
```

Árbol de decisión para hallar la cohesión de un módulo

Ejemplo

Los siguientes módulos corresponden a un sistema de manejo de archivos. El módulo principal corresponde a una interfaz a través de la cual se puede invocar al módulo MODIFICA o al módulo LISTADO. El módulo MODIFICA puede actualizar un cliente o agregar uno nuevo según el código que se envíe desde el programa principal. El módulo LISTADO muestra los datos de los clientes.

**Codificación en lenguaje C**

En principio, como los módulos MODIFICA y LISTADO trabajan con un archivo de clientes se define un archivo de cabecera que define el tipo de dato de las componentes.

Archivo Tipos.h
Módulo con Cohesion Abstraccional

```

#ifndef tipo // esta etiqueta indica al preprocesador que si lo de definido a
#define tipo // continuacion no existe, defina los tipos de datos que se señalan;
            // con lo cual la definición se hace una sola vez.

typedef struct
{
    char nombre[30];
    int cuenta;
} cliente;

#endif // cierra la directiva #ifndef
  
```


1) A continuación se define el módulo “manejo-archivos”, que se comporta como la interfaz a través de la cual interactúa el usuario.

Módulo Principal: **manejo-archivos.cpp**
Módulo con Cohesion Lógica

```
int main(void)
{ int opcion,op;
  do { system("CLS");
      printf("1- Listar archivo \n");
      printf("2 - Modificar o Agregar \n");
      printf("3 - Salir \n");
      printf("Opcion:"); scanf("%d", &opcion);
      switch (opcion)
      {
        case 1:{ listar();
                  break;
                }
        case 2:{ printf("\n ingrese opcion 1- Modifica 2 - Agrega");
                  scanf("%d", &op);
                  coordina _modifica (op); // Se invoca a la función que coordina
                                              // las actividades dentro del módulo
                  break;
                }
      }
  } while (opcion!=3);
}
```

2) El módulo LISTAR, muestra el contenido del archivo cliente. Se codifica y se graba con el nombre Listar_Archivo.cpp

Módulo Modifica: **listar_archivo.cpp**
Módulo con Cohesion Funcional

```
void listar () ,
{ cliente c;
  FILE * archi;
  if ((archi=fopen("Clientes","r+"))==NULL)
    printf("error de acceso \n");
  else {
    rewind(archi);
    while(fread(&c,sizeof(c),1,archi))
      printf("\n %20s %10d",c.nombre,c.cuenta);
    fclose(archi);
  }
}
```

3) El módulo MODIFICAR, codifica la función que coordina la actividad del módulo y desde la cual se llama a actualizar o agregar un cliente, según la opción que se haya recibido del módulo principal. Se codifica y se graba con el nombre `modifica_archivo.cpp`

Módulo Modifica: **modifica_archivo.cpp**
Módulo con Cohesion Lógica

```
static void actualizar (FILE *a, cliente cli)
{
    fpos_t pos;
    cliente r;
    fseek(a,0,SEEK_SET);
    while( fread(&r,sizeof(r),1,a) && (strcmp(r.nombre,cli.nombre)));
    {
        fseek(a, - sizeof(cli),SEEK_CUR);
        fwrite(&cli,sizeof(cli),1,a);
    }

static void agregar (FILE * a, cliente cli)
{
    fseek(a,0, SEEK_END);
    fwrite(&cli,sizeof(cli),1,a);
}

void coordina _modifica (int op)    // función coordinadora de las actividades dentro,
{                                     // del módulo a traves de la cual se invoca
    cliente c;
    FILE * archi;

    if ((archi=fopen("Clientes","r+"))==NULL)
        printf("error de acceso \n");
    else {
        printf ("\n ingrese el nombre del cliente "); gets(c.nombre);
        printf ("\n ingrese nuevo número de cuenta"); scanf("%d",&c.cuenta);
        if (op==1)
            actualizar(archi, c);
        else
            agregar(archi, c);
        fclose(archi);
    }
}
```

Nota: la cláusula `static` se usa para que la función declarada solo pueda ser accedida dentro del módulo.

Actividad

- Investigue qué realiza la instrucción del preprocesador `#ifndef #endif`
- Analice qué ventajas y/o desventajas tienen las operaciones de apertura y cierre en el interior del módulo.
- ¿Qué significado tiene el prefijo **static** en la función *agrega* y *actualiza*?
- ¿Es posible desde el módulo principal acceder a la función *agrega*?

Acoplamiento Inter - modular

El acoplamiento es el grado de relación o interdependencia que existe entre los módulos de una estructura de programas. Para medirlo se toma en cuenta el número y tipo de conexiones y la información comunicada a través de ellos. Mientras menos sea el número de conexiones y más simples sean éstas será el indicador de un mejor diseño.

La conexión simple entre módulos dará como resultado un software fácil de comprender y menos propenso a un efecto en cadena que es causado cuando los errores ocurren en un lugar y se propagan a través del sistema.

La idea es minimizar el acoplamiento de manera de hacer los módulos de la mejor manera posible. Un bajo acoplamiento va a ser una indicación de un programa bien particionado.

Muchos aspectos de la modularización pueden ser comprendidos sólo si se examinan módulos en relación con otros.

En general veremos que a mayor número de interconexiones entre dos módulos, se tiene una menor independencia.

El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de la información.

La cuestión aquí es: ¿cuánto debe conocerse acerca de un módulo para poder comprender otro módulo?

Cuanto más debamos conocer acerca del módulo B para poder comprender el módulo A, menos independientes serán A de B.

Módulos altamente "acoplados" estarán unidos por fuertes interconexiones, módulos débilmente acoplados tendrán pocas y débiles interconexiones, en tanto que los módulos "desacoplados" no tendrán interconexiones entre ellos y serán independientes.

El acoplamiento es un concepto abstracto que nos indica el grado de interdependencia entre módulos.

En la práctica podemos materializarlo como la probabilidad de que en la codificación, depuración, o modificación de un determinado módulo, el programador necesite tomar conocimiento acerca de partes de otro módulo. Si dos módulos están fuertemente acoplados, existe una alta probabilidad de que el programador necesite conocer uno de ellos, al intentar realizar modificaciones al otro.

Un acoplamiento bajo entre dos módulos indica un sistema bien particionado y puede ser obtenido considerando que, ningún módulo tiene que preocuparse por los detalles de construcción interna de cualquier otro.

Dos módulos pueden estar acoplados por más de un tipo de acoplamiento, o por el mismo tipo un número de veces. En estos casos el acoplamiento se define por el peor de los acoplamientos que se exhiba.

Una manera de evaluar el acoplamiento de un diseño es suponer que cada módulo debe ser codificado por diferentes programadores.

¿Cuán independientemente pueden trabajar los programadores?

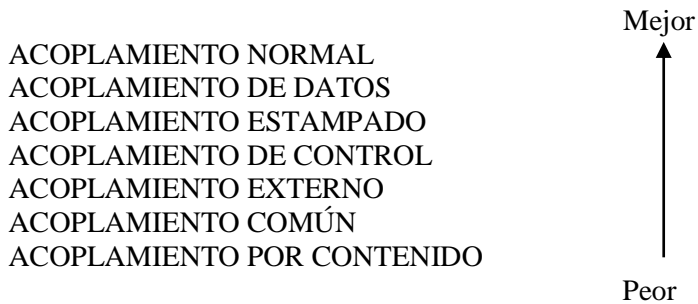
¿Hay alguna manera en que un cambio puede ser aislado a un solo módulo?

Contestando estas preguntas determinaremos cuales son probables para modificaciones requeridas por un gran número de módulos.

Claramente, el costo total del sistema se verá fuertemente influenciado por el grado de acoplamiento entre los módulos.

- El acoplamiento entre módulos se da de varias maneras: de control de flujo y de datos.
- El acoplamiento de control implica la transferencia del control de un módulo a otro.
- El acoplamiento de datos se refiere a compartir de datos entre los módulos.

Se analizarán los siguientes tipos de acoplamiento:

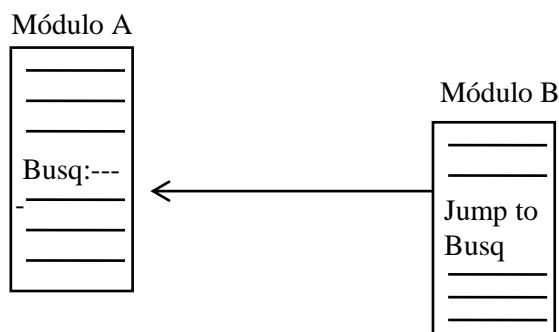


Acoplamiento por Contenido

Dos módulos tienen Acoplamiento por Contenido si uno refiere al interior del otro de cualquier manera. Por ejemplo, si un módulo bifurca y cae dentro de otro, o si un módulo se refiere a datos dentro de otro. Este acoplamiento no tiene en cuenta el concepto de caja negra, ya que fuerza al módulo a conocer en forma explícita la implementación y el contenido del otro.

La mayoría de los lenguajes de alto nivel no permiten implementar este tipo de acoplamiento, sólo el lenguaje ASSEMBLER permite realizarlo.

Ejemplo



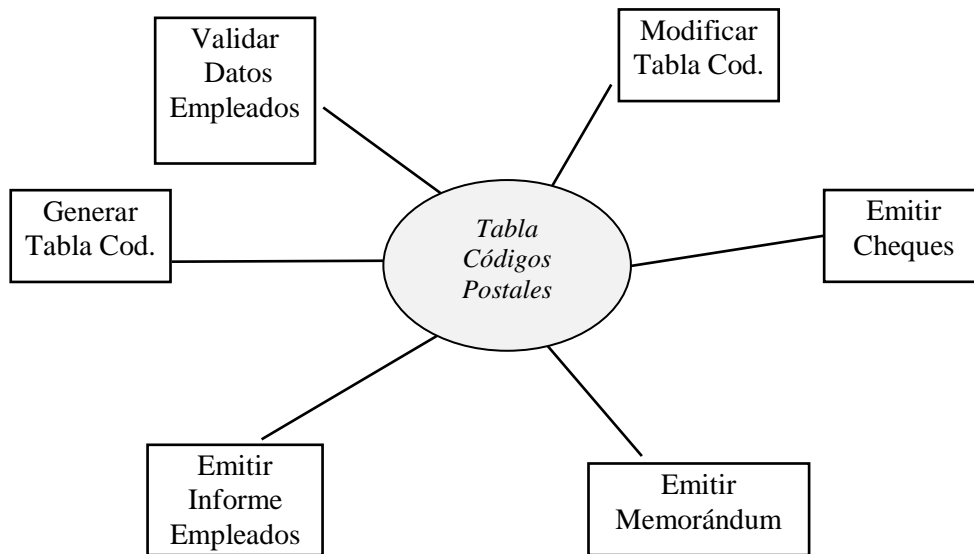
Si no existiera el Módulo B, el programador del Módulo A tendría completa libertad de recodificarlo, pero en este caso no puede cambiarlo, es decir no puede efectuar modificaciones sin consultar a B.

Acoplamiento Común

Dos o más módulos tienen Acoplamiento Común si ellos hacen referencia a una *estructura de datos global* compartida.

El acoplamiento común provoca los siguientes inconvenientes en los módulos así relacionados:

- Una modificación en sólo uno de ellos puede provocar impactos no deseados en los demás. Por ejemplo, si se quiere modificar la longitud de un campo de una estructura global compartida, habrá que recompilar todos los otros módulos que hacen referencia a esa estructura.
- Si un módulo hace referencia a un área común, se hace problemática su reusabilidad en otros contextos.

Ejemplo

Una solución es la utilización de pasajes de parámetros y utilizar de estructuras de datos globales en la menor cantidad de módulos posible.

Cuando un algoritmo se implementa con un lenguaje en particular, muchas veces surgen problemas de acoplamiento común que no pueden ser resueltos debido a las restricciones del lenguaje.

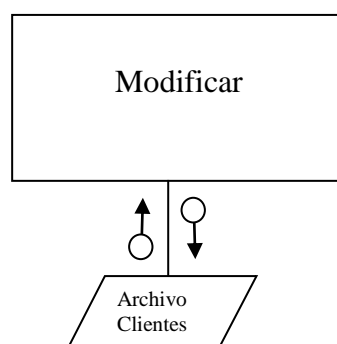
Acoplamiento Externo

Se produce cuando el módulo está ligado a algún dispositivo externo (como ser un disco, un sensor, canal de comunicación, etc.). En este caso, el formato de los datos que maneja el dispositivo exige que ante cualquier modificación se le haga también se modifiquen los módulos que lo usan.

El acoplamiento externo es inevitable, por esto es que en el diseño se debe buscar la forma de tener la menor cantidad posible de módulos con este acoplamiento.

Los módulos están ligados a un entorno externo al software. Este acoplamiento es esencial pero deberá estar limitado a unos pocos módulos.

Esto está básicamente relacionado con la comunicación a herramientas externas y dispositivos. Se puede considerar como dispositivos vinculados a monitores e impresoras, pero por ser dispositivos de uso permanente (sobre todo la pantalla) en el diseño se limita a aquellos que afectan de un modo directo al comportamiento del módulo.

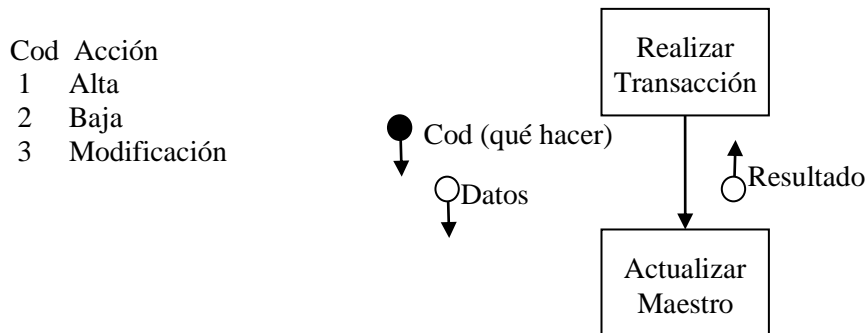
Ejemplo

Acoplamiento de Control

Dos o más módulos están Acoplados por Control si uno pasa al otro una pieza de información destinada al control de la lógica interna del otro. Los elementos típicos de control están representados por flags, códigos, llaves, etc.

En otras palabras se puede decir que dos o más módulos están Acoplados por Control si uno envía al otro además de los datos, información de cómo se desea que se procesen dichos datos.

Ejemplo



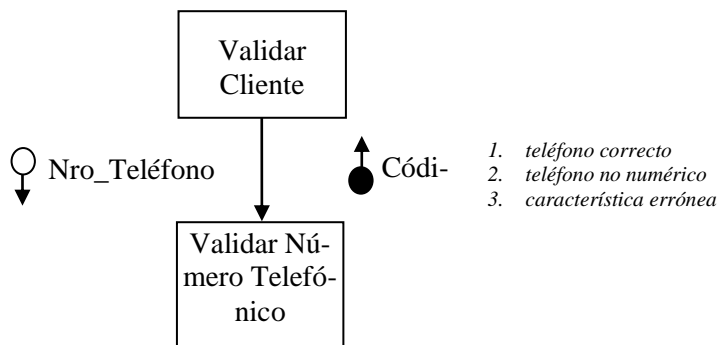
En este tipo de Acoplamiento, el módulo llamante debe conocer cómo está organizada la lógica del módulo subordinado, a fin de enviar la señal correcta.

Inversión de Autoridad

En el ejemplo anterior hemos detectado la presencia de un módulo padre (módulo llamante) que envía señales de control a un módulo hijo (módulo llamado o subordinado) que termina la tarea indicada por el módulo padre.

Si es el subordinado el que envía órdenes a su superior hemos realizado una falla en la partición que se llama "inversión de autoridad".

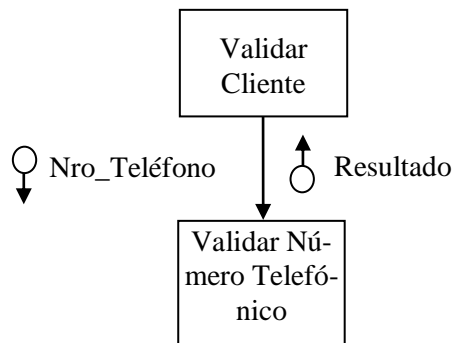
Ejemplo



En este ejemplo es el módulo padre es el que termina la tarea, ya que es él quien debe escribir el mensaje de error en función de la orden enviada por el módulo hijo. Luego, el módulo hijo deberá conocer la lógica interna del módulo padre a fin de enviar la señal correcta.

Cómo solucionamos la inversión de autoridad?

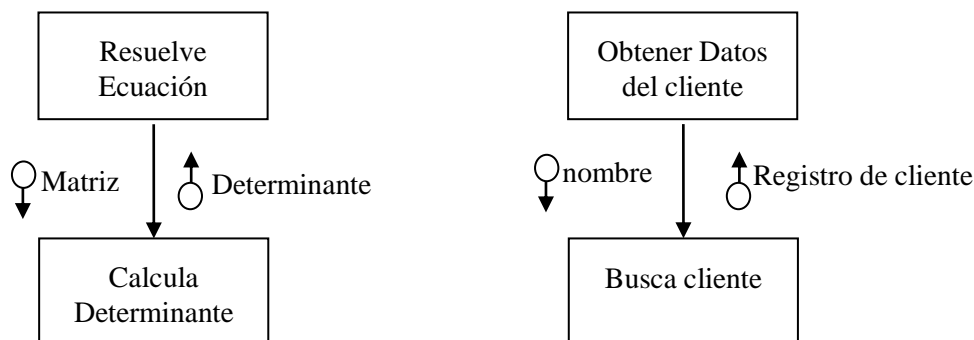
- a) El módulo subordinado escribe los mensajes de error
- b) El módulo subordinado reporta el estado de la tarea y el superior decide qué hacer, pero sin que el subordinado conozca cómo.

Ejemplo**Acoplamiento Estampado**

Dos módulos tienen Acoplamiento Estampado cuando hacen referencia a la misma "estructura de datos", la cual no está definida como global.

En el Acoplamiento Estampado, al no conocerse la ubicación o el nombre de la estructura de datos por no estar definida como global, es necesario que sea pasada como parámetro. Este acoplamiento crea dependencia entre módulos, que hasta este momento no estaban relacionados. Además tiende a exponer al módulo a más datos de los que necesita con posibles consecuencias no deseadas.

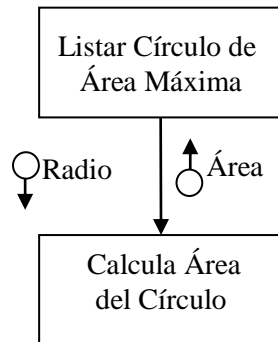
El Acoplamiento Estampado reduce los efectos negativos del Acoplamiento Común, tales como adaptación del módulo a diferentes contextos debido a que utiliza estructuras de datos de las cuales hay múltiples versiones en el programa (una por cada módulo que la utiliza).

Ejemplo**Acoplamiento de Datos**

Dos módulos tienen Acoplamiento de Datos, si todos los argumentos pasados desde y hacia los módulos están representados por elementos de datos que no cumplen funciones de control, ni tampoco constituyen estructuras de datos. El Acoplamiento de Datos significa relación por medio de elementos de datos simples y no por medio de estructura de datos.

Una estructura de datos como puede ser un registro de un archivo o un arreglo, implica Acoplamiento estampado.

Los módulos se comunican por parámetros y como los módulos deben comunicarse de alguna manera, el acoplamiento de datos es inevitable y bastante inofensivo.

Ejemplo***Pasaje de Punteros como Parámetros***

Supongamos un puntero P que apunta a una estructura S. Si el módulo A llama al módulo B pasando a P como parámetro, entonces el acoplamiento de A y B es "Estampado" y no de "Datos", pues si bien P es un dato simple, apunta a una estructura de datos que es a lo que realmente se hace referencia cuando pasamos P como parámetro, esto se observa al enviar un arreglo, una lista o un archivo.

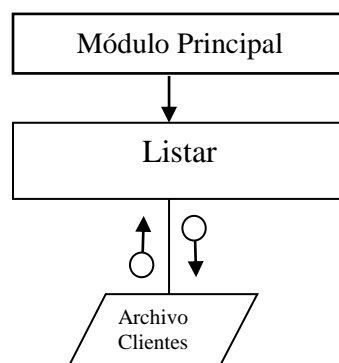
Acoplamiento Normal

Dos módulos están acoplados normalmente cuando no se pasan ningún parámetro entre ellos, sólo existe la llamada de uno a otro.

Dos módulos A y B están normalmente acoplados sí: un módulo A llama a otro B, y B retorna el control a A.

Ejemplo

Se observa entre el Módulo Principal y el Módulo Listar el acoplamiento Normal (en la grafica también se ve un acoplamiento externo en Listar)



Bibliografía

- Apuntes de Cátedra Programación Procedural.
- Braunstein Silvia y A. Gioia (1987) “Introducción a la Programación y a la Estructura de Datos”. Eudeba. Buenos Aires.
- Brassard, G. & Bratley, P.(1997)"Fundamentos de Algoritmia". Prentice-Hall. México.
- Cairó Osvaldo (2006) “ Metodología de la Programación. Algoritmos, diagramas de flujo y programas". 3º Edición Alfaomega. México.
- Criado Clavero, Mª Asunción (2006) “Programación en Lenguajes Estructurados” . Alfaomega. Ra-Ma. Madrid
- Calero, Moraga y Piattini (2010) "Calidad del Producto y Proceso Software". Editorial Ra-Ma. Madrid.
- Jesús Barranco de Areba (2001) "Metodología del análisis estructurado de sistemas". Universidad Pontifica Comillas. 2da Edición. Madrid.
- Kenneth y Kendall (2005) "Análisis y diseño de sistemas". Pearson Educación. 6ta Edición. México.
- Morales, Roberto Cortés (1992) "Introducción al Análisis de Sistemas y la Ingeniería de Software". EUNED. Costa Rica.
- Fontela, Carlos (2003) "Programación Orientada a Objetos. Técnicas Avanzadas de Programación". Editorial Nueva Librería. Buenos Aires.

Practico 6

Diseño Modular

Estudio Dirigido

Propuesta de trabajo

La siguiente guía tiene como propósito orientar en la comprensión y aplicación de contenidos de cohesión y acoplamiento, a través de actividades prácticas que permitan integrar los temas vistos.

Mediante este estudio se propone fomentar la capacidad de análisis, síntesis y de trabajo colaborativo, que se reflejará a través del desarrollo de un Proyecto en Lenguaje C, que tenga en cuenta características del Diseño Modular como Metodología de Diseño de Software.

Objetivos

A través del siguiente trabajo se espera que cada alumno:

- Conozca una forma de aprendizaje a través del análisis y estudio de material vinculado al diseño estructurado como metodología de diseño de algoritmos y la discusión activa de hallazgos en un grupo de estudio.
- Potenciar el desarrollo del aprendizaje autónomo.
- Conocer un Modelo de Trabajo Grupal, donde se asignen tareas y responsabilidades para el logro de un objetivo determinado.

Contenidos Curriculares

1. Diseño Modular. Cohesión y acoplamiento como criterios cualitativos para medir la calidad de software.
2. Administración de archivos.
3. Construcción de Proyectos en Dev-C++.

Modalidad de Trabajo

La técnica de estudio dirigido que hemos elegido como método de abordaje de este trabajo consiste en que se analice grupalmente el tema.

El grupo debe estar constituido por 3 integrantes como máximo y la presentación de la tarea es individual.

Planteamiento del problema

Para cada actividad planteada, cada grupo deberá:

- a) Buscar una solución y elaborar el diseño modular que responde a la misma.
- b) Especificar la cohesión modular y el acoplamiento intermodular de los módulos construidos.
- c) Codificar cada uno de los módulos
- d) Utilizar Proyectos en Dev-C++

Evaluación

En el laboratorio de computación cada alumno deberá presentar su propuesta, justificando el diseño elegido. El código debe ejecutar correctamente y responder al diseño realizado. El docente podrá solicitar modificaciones u optimizaciones.

Actividades a presentar

Para llevar a cabo estas actividades se realizará el siguiente proceso:

- a) Buscar entre todos los integrantes una solución a la problemática planteada
- b) Elaborar el diseño modular que responde a la solución elegida.
- c) Especificar la cohesión modular y el acoplamiento intermodular.
- d) Codificar cada uno de los módulos usando funciones óptimas, al menos una recursiva
- e) Utilizar proyectos en Dev-C++

ACTIVIDAD 1

Una consultora contable trabaja en la liquidación de haberes de los empleados de varias empresas pequeñas.

La empresa posee en un archivo “EMPRESAS.dat” con información general de cada una de las empresas: código (número secuencial, generado automáticamente a partir de 1200), nombre, CUIT y dirección.

Para el cálculo de sueldos de los empleados se realiza de la misma forma, se trabaja con un archivo de empleados ordenado ascendentemente por Código de empresa.

El archivo de empleados “EMPLEADOS.dat” posee la siguiente información: Nombre del Empleado, código de la empresa donde trabaja, DNI, sueldo básico, antigüedad.

Los empleados además tienen descuentos fijos sobre su remuneración (sueldo básico más antigüedad), estos son: Jubilación 11%, Obra Social 3%, Seguro de Vida \$ 3,80.

Se pide generar diferentes módulos que permitan:

Módulo 1: Genera un archivo con información de las distintas empresas que trabajan con el estudio contable.

Módulo 2: Generar un archivo con información de los empleados.

Módulo 3: Emita un listado ordenado por empresa con la liquidación de haberes de cada empleado, que incluya además el Total Pagado en concepto de haberes y el Total Pagado en concepto de Descuento:

Empresa: xxxxxx (nombre de la empresa)

Nombre de Empleado	Remuneración
-----	-----
-----	-----
-----	-----
Total Pagado en concepto de haberes:	-----
Total Pagado en concepto de Descuento:	-----

Empresa: xxxxxx (nombre de la empresa)

Nombre de Empleado	Remuneración
-----	-----
-----	-----
-----	-----
Total Pagado en concepto de haberes:	-----
Total Pagado en concepto de Descuento:	-----

ACTIVIDAD 2

Módulo 1: Generar un archivo “alumnosPP.dat” que contenga la información correspondiente a los alumnos que cursan la materia Programación Procedural: Nombre y Número de registro y resultado de un parcial (A: aprobado - R: Reprobado). La información se ingresa ordenada por matrícula.

Módulo 2: Generar un archivo “alumnosAL.dat” que contiene la siguiente información correspondiente a los alumnos que cursan la materia Álgebra Lineal: Nombre y Número de registro y resultado de un parcial (A: aprobado - R: Reprobado). La información se ingresa ordenada por matrícula.

Módulo 3: Recibir un código ‘A’ o ‘R’, Si es ‘A’ se genera una lista enlazada con los alumnos que aprobaron ambas asignaturas. Si es ‘R’ genera una lista enlazada con los alumnos que reprobaron ambas asignaturas. La lista debe crearse de manera ordenada por matrícula usando la técnica de mezcla de archivos.

Módulo 4: Este módulo debe diseñarse de modo que permita generar un archivo “UNA.dat” con información de los alumnos que sólo aprobaron una asignatura.

Nota: La codificación de los módulos debe ser óptima.

ACTIVIDAD 3

La biblioteca de la Facultad de Ciencias Exactas, Físicas y Naturales posee un sistema que incluye módulos, a través de los cuales desarrolla varias funcionalidades.

Para ello utiliza distintos archivos que a continuación se describen:

LIBROS, este archivo posee datos de los libros que posee la biblioteca.

- Nombre: es el nombre del libro.
- Código: es un campo clave que se inicia en 1 para el primer libro.
- Cantidad: cantidad total de ejemplares de un libro.
- Disponible: cantidad de ejemplares de un libro disponibles para préstamo.
- Categoría: temática a la que pertenece el libro.

Nota: los libros se encuentran ordenados secuencialmente por código.

SOCIOS, este archivo posee datos de los socios de la biblioteca.

- DNI: es el número de documento del socio de la biblioteca.
- Número: es el número de identificación de cada socio de la biblioteca. Inicia en 100.
- Nombre y apellido: es el Nombre y Apellido del socio de la biblioteca.
- Ejemplares: cantidad de libros que posee o no el socio en su haber.

PRÉSTAMOS, este archivo posee datos de los distintos préstamos que la biblioteca realiza.

- Número de socio: es el número de identificación de cada socio de la biblioteca.
- Código de libro: es el código que identifica al libro que se entregó o recibió.
- Operación: (P: préstamo; D: devolución) indica la operación que realiza el socio en la biblioteca para cada uno de los libros.

NARRATIVA

La biblioteca de la FCEFyN posee diferentes tipos de libros, de los cuales se pueden realizar préstamos y/o consulta de los mismos. La información de los libros se almacena en el archivo "**Libros.dat**".

Diariamente se crea un arreglo con la información de este archivo. *Este arreglo se actualiza cada vez que solicita el préstamo de un libro y cuando se devuelve algún libro por parte de un socio, por lo tanto esto implica constante actualización.* Es muy importante tener en cuenta que cuando sólo queda un ejemplar de un libro, este no puede ser entregado en calidad de préstamo sino que el socio puede consultarlo en el lugar.

El archivo "**Libros.dat**" se actualiza al salir del sistema.

Por otro lado la biblioteca posee un archivo con los datos de los socios "**Socios.dat**", que pueden solicitar préstamos o realizar consultas. Los socios pueden retirar hasta 3 libros como máximo y no puede llevar más de un ejemplar por libro.

Es necesario contar con una estructura auxiliar (lista) que permita almacenar las operaciones que realizan los socios con la biblioteca (préstamos, consultas y devoluciones) por lo tanto se solicita registrar número de socio, código de libro y tipo de operación. Esta estructura es necesaria para actualizar el archivo "**Prestamos.dat**" al salir del sistema.

Cuando un Socio hace una devolución, se actualiza:

- El arreglo de Libros.
- El archivo de Socios.
- La lista diaria de operaciones de Socios: si se devuelve un libro que fue prestado con anterioridad a la fecha actual, se registra en la lista una devolución. Si se prestó en la fecha actual debe estar en la lista y debe ser eliminado de la misma. Si corresponde a una consulta (el libro se prestó en el momento), se elimina de la lista.

Funcionamiento

Cuando una persona solicita un libro, el bibliotecario verifica si es socio, si no fuera así lo ingresa como nuevo socio. Todo socio nuevo debe esperar 24 horas para poder hacer uso de la biblioteca.

Una vez verificada la calidad de socio, se consulta si está habilitado para solicitar libros (considerar que los socios pueden tener en su haber solo 3 libros). Si el socio se encuentra en condiciones de solicitar material, se le pide el nombre del libro, allí el bibliotecario verifica la existencia del mismo y la cantidad de ejemplares disponibles como para cubrir el pedido. Si la cantidad de ejemplares de determinado libro fuera superior a uno (1), puede realizarle la operación de préstamo, caso contrario se le comunica al socio la negativa del pedido del ejemplar y se le notifica que se encuentra habilitado solo para consulta. En caso de préstamo esto debe quedar registrado para la correcta actualización de los datos, lo mismo en el caso de que un socio quiera realizar la devolución del o los ejemplares que posee.

Diseño modular

- Represente gráficamente el diseño en general, y particularmente cada módulo especificando en forma detallada las tareas que se realizan (Refinamiento), indicando datos de entrada y salidas.
- Indique tipo de cohesión y acoplamiento.
- Incluya en el diseño al menos 3 módulos: uno que contenga cohesión lógica, otro secuencial y otro funcional.

Implementación

Para administrar las funcionalidades del sistema debe incluir módulos que permitan:

- Crear los archivos indicados.
- Registrar el ingreso de nuevos libros.
- Registrar el ingreso de nuevos ejemplares para libros existentes. (solo pueden ser usados 24 horas después de su carga)
- Registrar el ingreso de nuevos socios (están habilitados 24 horas después de su carga).
- Realizar un listado de libros por temática.
- Informar si un libro está disponible para préstamo o consulta conociendo el nombre del mismo.
- Informar si un libro está disponible para préstamo o consulta conociendo su código.
- Realizar un listado de los libros que se encuentren en consulta.
- Registrar la devolución de un libro.
- Actualizar el archivo de préstamos.

NOTA: Es importante optimizar el código. Realizar un menú de opciones y validar los datos de entrada.

Presentación

Cada Grupo presenta una carpeta con el desarrollo de cada actividad. Esta carpeta contiene:

- Caratula: Apellido y Nombre, Registro y Carrera (de cada integrante del grupo).
- Grafico del diseño Modular de cada actividad.
- Cada Modulo debe estar graficado de modo tal que se observe su refinamiento, entradas y salidas. Respecto al Diseño modular deben indicarse cohesión y acoplamiento sin incluir definiciones teóricas.