Programación I (Plan 1999)
Algoritmos y Estructuras de Datos II (Plan 2009)

Unidad 1: Sustentación de conceptos básicos sobre algoritmos

Tema I. Sustentación de estructuras de datos simples. Funciones y Procedimientos. Archivos. Algoritmos. Concepto y Características. Concepto de programa. Escritura de algoritmos/programas. Tratamiento de arreglos. Programación Estructurada. Funciones y Procedimientos. Búsqueda de máximos y mínimos. Archivos: Operaciones simples.

Mgter. Oscar Adolfo Vallejos
FaCENA - UNNE

Solución de problemas mediante programas

- Los computadores desempeñan una gran variedad de tareas
- La tarea de la programación consiste en describir lo que debe hacer el computador para resolver un problema concreto en un lenguaje de programación.
- Se identifican de momento las siguientes fases:

I. Análisis del problema, estableciendo con precisión lo que se plantea.

II. Solución conceptual del problema, describiendo un método (algoritmo) que lo resuelva.

III. Escritura del algoritmo en un lenguaje de programación.

Concepto de algoritmo

El término “algoritmo" tiene connotaciones más formales que cualquier otro debido a su origen: se trata de una acomodación al castellano del nombre de Muhammad ibn Musa al Jwarizmi, matemático persa que popularizo su descripción de las cuatro reglas (algoritmos) de sumar, restar, multiplicar y dividir.

DEFICION DE ALGORITMO …

Características de los algoritmos

1. Precisión: Un algoritmo debe expresarse de forma no ambigua. La precisión afecta por igual a dos aspectos:

   - Al orden (encadenamiento o concatenación) de los pasos que han de llevarse a cabo.
   - Al contenido de las mismas, pues cada paso debe “saberse realizar“ con toda precisión, de forma automática.

2. Determinismo: Todo algoritmo debe responder del mismo modo ante las mismas condiciones.

3. Finitud: La descripción de un algoritmo debe ser finita.

Cualidades deseables de un algoritmo

1. Generalidad

Es deseable que un algoritmo sirva para una clase de problemas lo más amplia posible.

2. Eficiencia

Hablando en términos muy generales, se considera que un algoritmo es tanto más eficiente cuantos menos pasos emplea en llevar a cabo su cometido y/o utilice menos memoria en su proceso.

Aspectos de interés sobre los algoritmos

- Computabilidad
  - hay problemas no computables

    (adviértase que, si no se conoce algoritmo que resuelva un problema planteado, sólo se puede asegurar “que no se conoce algoritmo que resuelva ese problema"; en cambio, establecer que un problema “no es computable" requiere demostrar que nunca se podrá encontrar ningún algoritmo para resolver el problema planteado).

  Décimo problema de Hilbert
  Problema de la parada
  Números pedrisco

- Corrección de algoritmos

  Es imposible detectar los posibles errores de una forma sistemática. Con frecuencia, la búsqueda de errores (depuración) consiste en la comprobación de un algoritmo para unos pocos juegos de datos.

  La verificación consiste en la demostración del buen funcionamiento de un algoritmo con respecto a una especificación.

- Complejidad de algoritmos

  Resulta de gran interés poder estimar los recursos que un algoritmo necesita para resolver un problema. En máquina secuenciales, estos recursos son el tiempo y la memoria.

  Como el tiempo requerido por los programas depende en gran medida de la potencia de la máquina ejecutora, es frecuente medir en “pasos" (de coste fijo) el coste del algoritmo correspondiente.

Cualidades de Lenguajes algorítmicos y de programación

- Tienden un puente entre la forma humana de resolver problemas y su resolución mediante programas de computador.

- Tienen cierta independencia de los lenguajes de programación particulares, de modo que están libres de sus limitaciones y así los algoritmos escritos en ellos se pueden traducir indistintamente a un lenguaje de programación u otro.

- Constituyen una herramienta expresiva con una libertad y flexibilidad próxima a la de los naturales y con el rigor de los de programación.

- Las únicas restricciones que deberán imponerse a estos lenguajes proceden de las carácteristicas que tienen los algoritmos: expresar sus acciones (qué deben realizar y cuándo) con la precisión necesaria, y que estas acciones sean deterministas.

- Las acciones, se expresan mediante instrucciones, que son comparables a verbos en infinitivo: asignar. . , leer. . , escribir. . .y otras.

- La concatenación de las instrucciones expresa en qué orden deben sucederse las acciones; esto es, cómo se ensamblan unas tras otras.

Herramientas de Programación

Las herramientas de programación mas utilizadas comúnmente para diseñar algoritmos son:

- Pseudocódigos
- Diagramas N-S
- Diagramas de flujo

Siendo el pseudocódigo el mas popular por su sencillez y su parecido a el lenguaje humano.

Desarrollo sistemático de programas

- **Planificación:** En esta fase inicial hay que constatar la verdadera necesidad del producto que se va a desarrollar y valorar los recursos humanos y técnicos que precisa su desarrollo.
- **Análisis**: En la fase de análisis se establecen cuáles deben ser la funciones que debe cumplir la aplicación y cómo debe realizarse el trabajo conjunto de los diferentes módulos en que se va a dividir.
- **Diseño**: En esta fase se diseña el conjunto de bloques, se dividen en partes y se asignan a los equipos de programadores.
- **Codificación**: La fase de codificación se confunde muchas veces con la programación y consiste en escribir los algoritmos en un lenguaje de programación. Es un proceso casi automático.
- **Validación**: La fase de validación consiste en aplicar el sistema de pruebas a los módulos, a las conexiones entre ellos (prueba de integración) y, por último, a la totalidad de la aplicación (prueba de validación).
- **Mantenimiento**: En la fase de mantenimiento se redacta la documentación actualizada, tanto para el programador como para el usuario. Se inicia la explotación de la aplicación y se detectan y corrigen los errores y deficiencias no advertidas en las fases anteriores, lo que puede suponer un coste añadido importante.

Tipos de Datos

Los diferentes objetos de información con los que un programa trabaja se denominan *datos*.

- Todos los datos tienen un tipo asociados con ellos que nos servirá para poder conocer con que información trabajaremos.

La asignación de tipos a los datos tiene dos objetivos principales:

- Detectar errores de operaciones aritméticas en los programas
- Determinar como ejecutar las operaciones

Tipos de Datos Comunes

Estos son los tipos de datos mas utilizados en los lenguajes de programación:

- Numéricos: Dentro de estos tipos se puede hacer mención de los tipos enteros, reales o de coma flotante, y de los exponenciales.

- Carácter: Se dividen también en caracteres ASCII, como por ejemplo: a A & * , etc.. El otro grupo de caracteres son los strings o cadenas de caracteres, como por ejemplo: "Hola Mundo".

- Lógicos: Pueden tomar los valores verdadero o falso

El tipo de una expresión

El resultado de cada expresión tiene un tipo determinado, independiente de su valor, y que puede conocerse aun ignorando los valores de sus elementos componentes, teniendo en cuenta sólo sus tipos.

Así por ejemplo, pueden existir:

Expresiones numéricas

- enteras (como 2 + 3)
- reales (como 3.14 * Sqr(2.5)

Expresiones booleanas (como 2 + 2 = 5)

Expresiones de cadena (como Succ('a')).

Identificadores

Representan los nombres de los objetos de un programa (constantes, variables, tipos de datos, procedimientos, funciones, etc.).

Es un identificador es un método para nombrar a las celdas de memoria en la computadora, en lugar de memorizarnos una dirección de memoria.

Se utilizan para nombrar variables, constantes, procedimientos y funciones.

Identificadores

Constantes

Son valores que no pueden cambiar en la ejecución del programa. Recibe un valor en el momento de la compilación del programa y este no puede ser modificado.

Variables

Son valores que se pueden modificar durante la ejecución de un programa. Al contrario de las constantes estos reciben un valor, pero este valor puede ser modificado durante la ejecución o la compilación del programa.

Funciones/Procedimientos

Conjunto de instrucciones y/o variables para la obtención de un resultado. Necesario para el proceso global. Subrutina.

Operadores utilizados en Programación

Operadores aritméticos: Una *expresión* es un conjunto de datos o funciones unidos por operadores aritméticos (suma, resta, multiplicación, división, exp, mod, etc.)

Operadores Lógicos: En ocasiones en los programas se necesitan realizar comparaciones entre distintos valores, esto se realiza utilizando los operadores relaciones.

Entrada y Salida (I/O) de Datos

La lectura de datos permite asignar valores desde dispositivos hasta archivos externos en memoria, esto se denomina operación de entrada o lectura.

A medida que se realizan calculos en el programa, se necesitan visualizar los resultados. Está se conoce como operación de escritura o salida.

Tipo de Datos

Datos Simples o Primitivos	estándar	entero(integer)
		real (real)
		carácter (char)
		lógico (boolean)
	definido por el programador (no estándar)	subrango (subrange)
		Enumerativo (enumerated)
Datos Estructurados o Datos compuestos	estáticos	Arreglos (vectores/matrices)
		registro (record)
		ficheros (archivos)
		conjuntos (set)
		cadenas (string)
	dinámicos	listas (pilas/colas)
		listas enlazadas
		Árboles y grafos

Tipos de Datos

- Los tipos de datos simples o primitivos: son aquellos que no están definidos en términos de otros tipos de datos. Se denominan “primitivos” también porque son los tipos de datos originales que proporcionan la mayoría de los lenguajes de programación.

- Los tipos de datos estructurados o compuestos están construidos en base a los tipos de datos primitivos. Un ejemplo, es la cadena o string de caracteres.

- A su vez, las estructuras compuestas pueden ser:

- Estáticas: cuando el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa.

- Dinámicas: no tienen limitaciones o restricciones en el tamaño de memoria ocupada (este tipo de estructura no se contempla en esta asignatura).

Tipos de Datos

Diferencia entre los tipos de datos

- Los tipos de datos simples tienen como característica común que cada variable representa un elemento. Por ejemplo: Edad, Temperatura, Importe, etc.

- Los tipos de datos estructurados tienen como característica común que un identificador (nombre) puede representar múltiples datos individuales, pudiendo cada uno de éstos ser referenciado independientemente. Por ejemplo: Vector de temperaturas diarias, Nombre y Apellido, Registro de alumnos, Matriz Identidad, etc.

Arreglos

- Tipo de datos estructurado array, que será útil para trabajar con estructuras de datos como, por ejemplo, vectores, matrices, una sopa de letras rectangular, una tabla de multiplicar, o cualquier otro objeto que necesite una o varias dimensiones para almacenar su contenido.

Arreglo

Un arreglo (matriz o vector) es un conjunto finito y ordenado de elementos homogéneos.

- La propiedad ordenado, significa que el elemento primero, segundo,.., enésimo de un arreglo puede ser identificado.

- La propiedad homogéneo, quiere decir que los elementos son del mismo tipo de datos. Por ejemplo, un arreglo puede tener todos sus elementos de tipo entero, o todos sus elementos de tipo char.

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]
12	5	-7	14.5	20	1.5	2.5	-10

Arreglos

Descripción del tipo de datos array

- Los arrays son tipos de datos estructurados ampliamente utilizados, porque permiten manejar colecciones de objetos de un mismo tipo con acceso en tiempo constante, y también porque han demostrado constituir una herramienta de enorme utilidad.

- Imaginemos que queremos calcular el producto escalar, U * V de R(3).

  U*V = u1 *v1 + u2*v2 + u3*v3

- Con los datos simples que conocemos hasta ahora, tendríamos que definir una variable para cada una de las componentes de los vectores.

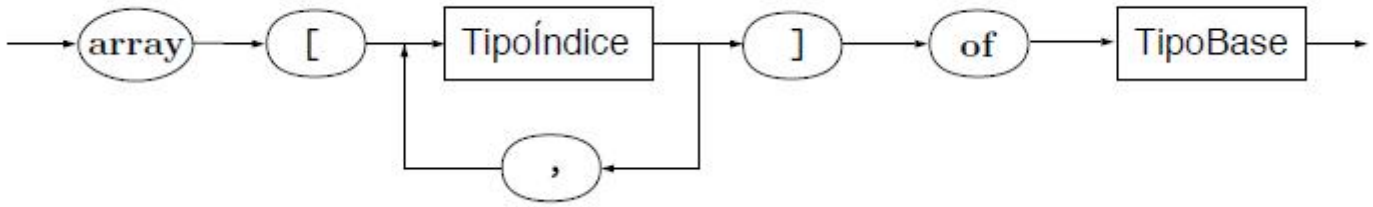
  var

  u1, u2, u3, v1, v2, v3: real;

Para este caso sería más natural disponer de una variable estructurada que agrupe en un solo objeto las componentes de cada vector.

Arreglos

- El diagrama sintáctico de la definición de un array es:



- Su definición en C es:

tipo nombre_arr [tamaño] ➔ int listanum[10];

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

La forma como pueden ser accesados los elementos de un arreglo:

```
listanum[2] = 15;      /* Asigna 15 al 3er elemento del arreglo listanum*/
num = listanum[2];   /* Asigna el contenido del 3er elemento a la variable num */
```

C permite arreglos con más de una dimensión, el formato general es:

tipo nombre_arr [tam1][tam2] ... [tamN]; ➔ int tabladenums[50][50];

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

tipo nombre_arr[tam1][tam2] ... [tamN] = {listavalores};

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

Arreglos

Operaciones con vectores:

- Asignación
- Lectura/escritura
- Recorrido (acceso secuencial)
- Actualizar (añadir, borrar, insertar)
- Ordenación
- Búsqueda

Arreglos

- Operaciones del tipo array y acceso a sus componentes
  - Acceso y asignación a componentes de un array
    - Se accede a sus elementos mediante el uso de tantos índices como dimensiones tenga el array, siguiendo el siguiente esquema:

      idArray [expres1, expres2, ..., expresL]

      idArray [expres1][expres2]...[expresL]
    - Para dar valores a las componentes de un array se usa la instrucción de asignación:

```
v[2]:= 3.14
m[i,2 * i - 1]:= 8
c[1][2][1]:= 'b'
d[mercurio, pluton]:= 3.47E38
costaMar[12,3,'B']:= True
```

Arreglos

Características generales de un array

- Los arrays son estructuras homogéneas, en el sentido de que sus elementos componentes son todos del mismo tipo.

- El tamaño del array queda fijado en la definición y no puede cambiar durante la ejecución del programa, al igual que su dimensión.

- Los datos de tipo array se pueden pasar como parámetro en procedimientos y funciones, pero el tipo que devuelve una función no puede ser un array (ya que un array es un tipo de datos compuesto).

Arreglos: Vectores

- En términos generales, un vector es una secuencia, de longitud fija, formada por elementos del mismo tipo.

```
type
  tNumeros = 1..10;
  tDiasSemana = (lun,mar,mie,jue,vie,sab,dom);
  tVectorDeR10 = array[tNumeros] of real;
  tFrase = array[1..30] of char;
  tVectorMuyGrande = array[integer] of real;
```

Arreglos: Vectores

Veamos ahora un ejemplo de manejo de vectores en C.

Para indicar cuántos viajeros van en cada uno de los 5 vagones de un tren, en lugar de utilizar 5 variables enteras (una para cada vagón), se puede y se debe utilizar un vector. La cantidad de pasajeros de cada vagón ingresa por teclado.

Al final, el programa mostrara el total de pasajeros del tren.

Haremos referencia al número de pasajeros del vagón i-ésimo mediante vagón[i], mientras que el total de viajeros en el tren será:

$$\sum_{i=1}^{15} \texttt{vagon[i]}$$

```
int main(){
  int sum, vagon[5];
  int i;
  sum=0;
  for (i=0;i<5;i++){
                printf("Cantidad pasajeros del vagon %d  : ", i);
                scanf("%d",&vagon[i]);
     }
  for(i=0;i<5;i++)
  {
   sum=sum+vagon[i];
  }
  printf("La cantidad de pasajeros del tren es : %d", sum);
 return 0;
}
```

Arreglos: Matrices

Como ya se dijo, los arrays multidimensionales reciben el nombre genérico de matrices. En este apartado se presentan algunos ejemplos de utilización de matrices.

Evidentemente, la forma de definir los tipos de datos para las matrices es la misma de todos los arrays, así como el modo de declarar y manipular variables de estos tipos.

Ej.: Que rellene una matriz de 3x3 y muestre su traspuesta (la traspuesta se consigue intercambiando filas por columnas y viceversa)

```
int main(void){
    int x,y,num=0, numeros[4][4];
    for (x=0;x<3;x++)
        {
        for (y=0;y<3;y++)
           {
            numeros[x][y]=num;
            num++;
            }
         }
    printf("El array original es: \n\n\n");
    for(x = 0;x < 3;x++)
    {
      for(y = 0;y < 3;y++)
      {
          printf("    %d     ", numeros[x][y]);
      }
      printf("\n\n\n");
    }
    printf("La traspuesta es: \n\n\n");
    for(x = 0;x < 3;x++)
    {
      for(y = 0;y < 3;y++)
      {
          printf("    %d     ", numeros[y][x]);
      }
      printf("\n\n\n");
    }
    system("PAUSE");
    return 0;
}
```

Programación Estructurada. Subprogramas, Procedimientos y Funciones

Modularización

Modularizar significa descomponer un problema en partes funcionalmente independientes.

Cada una de estas partes se denomina módulo.

Los **módulos** representan tareas específicas bien definidas que deben comunicarse entre sí adecuadamente y cooperar para conseguir un objetivo común.

Modularización - División en subtareas

La mejor manera de resolver un problema grande es → **dividirlo en partes más pequeñas**
En programación se conoce como → **división en subtareas o módulos**

La división en subtareas consiste en:

1. Identificar pequeñas tareas fácilmente explicables (avanzar un paso, subir un peldaño, encender una luz) y asignarles un **nombre descriptivo** para que se entienda fácilmente qué representan.
2. Combinar estas pequeñas tareas para definir el programa que resuelve el problema

Modularización

- Cada tarea recibe el nombre de módulo, subprograma o rutina.
- Un módulo se ejecuta cada vez que el *programa principal* lo *invoca*
- Cada módulo debe tener un identificador (nombre) y en algunos casos, una serie de parámetros.
- El identificador del módulo se utiliza para su invocación.

Metodología - Estrategia

TOP DOWN

Ir de lo general a lo particular

Dividir … conectar … y verificar

Es una técnica para resolver problemas que comienza descomponiendo el problema principal en subproblemas, para luego diseñar las soluciones específicas para cada uno de ellos.

Modularización - Ventajas

Mayor productividad

Al dividir un sistema de software en módulos funcionalmente independientes, un equipo de desarrollo puede trabajar simultáneamente en varios módulos, incrementando la productividad (es decir **reduciendo el tiempo de desarrollo global del sistema**).

Modularización - Ventajas

Reusabilidad

Objetivo fundamental de la Ingeniería de Software → *reusabilidad*,

Posibilidad de utilizar repetidamente el producto de software desarrollado

La modularización *favorece el reuso.*

Modularización - Ventajas

Facilidades de mantenimiento correctivo.

Sistema en módulos permite

→ aislar los errores que se producen con mayor facilidad

→ corregir los errores en menor tiempo

→ disminuye los costos de mantenimiento de los sistemas

Modularización - Ventajas

Facilidades de evolución del sistema.

Los sistemas de software reales crecen (es decir aparecen con el tiempo nuevos requerimientos del usuario).

La modularización permite disminuir los riesgos y costos de incorporar nuevas prestaciones a un sistema en funcionamiento.

Modularización - Ventajas

Mayor Legibilidad

Un efecto de la modularización es una mayor claridad para leer y comprender el código fuente.

El ser humano maneja y comprende con mayor facilidad un número limitado de instrucciones directamente relacionadas.

Procedimientos y funciones

Supongamos que queremos escribir un programa que pida al usuario el valor de un cierto ángulo en grados sexagesimales, calcule su tangente y escriba su valor con dos decimales. En una primera aproximación podríamos escribir:

Sean a, t $\in \mathbb{R}$
Leer el valor del ángulo a *(en grados)*
Calcular la tangente, t, *de* a
Escribir el valor de t *con dos decimales*

```
Program CalculoTangente (input, output);
  {Se halla la tangente de un ángulo, dado en grados}
  var
    a, {ángulo}
    t: real; {su tangente}
begin
  Leer el valor del ángulo a (en grados)
  t:=  tangente de a;
  Escribir el valor de t, con 2 decimales
end.  {CalculoTangente}
```

Funciones en C

- Una función es un módulo que realiza una tarea específica.
- Se invoca o llama mediante su identificador.
- No pueden ejecutarse por sí mismas: su ejecución depende de que sea invocada en alguna otra función

Tipos de funciones en C

- Estándares o predefinidas
  - bibliotecas estándar de C
  - uso de la directiva #include
  - Ejemplos: **printf**, **scanf**, **pow**.
- Definidas por el programador

Las diseña el programador según sus necesidades

Sintaxis en C

```
Tipo_de_retorno nombreFuncion (lista de parámetros){
        cuerpo de la función;
        return expresión;
}
```

Tipo_de_retorno	Es el tipo de dato del valor que devuelve la función. Si se omite el tipo de dato, asume entero.
nombreFuncion	Es el nombre asignado a la función.
Lista de parámetros	Lista de variables con sus respectivos tipos de datos: tipo1 parametro1, tipo2 parametro2,..., Cuando existen, éstos son los datos que debe recibir la función cuando se le invoque.
Cuerpo de lafunción	Son las sentencias o instrucciones que ejecutará la función cada vez que sea invocada.
Expresión	Valor que devuelve la función.

Prototipo de una función

- La **declaración** de una función se denomina “prototipo”.
- Describe la lista de argumentos que la función espera recibir y el tipo de datos de su valor de retorno.
- Contiene la cabecera de la función, siendo opcional incluir los nombres de variables, y termina con punto y coma para indicar al compilador que es una instrucción.

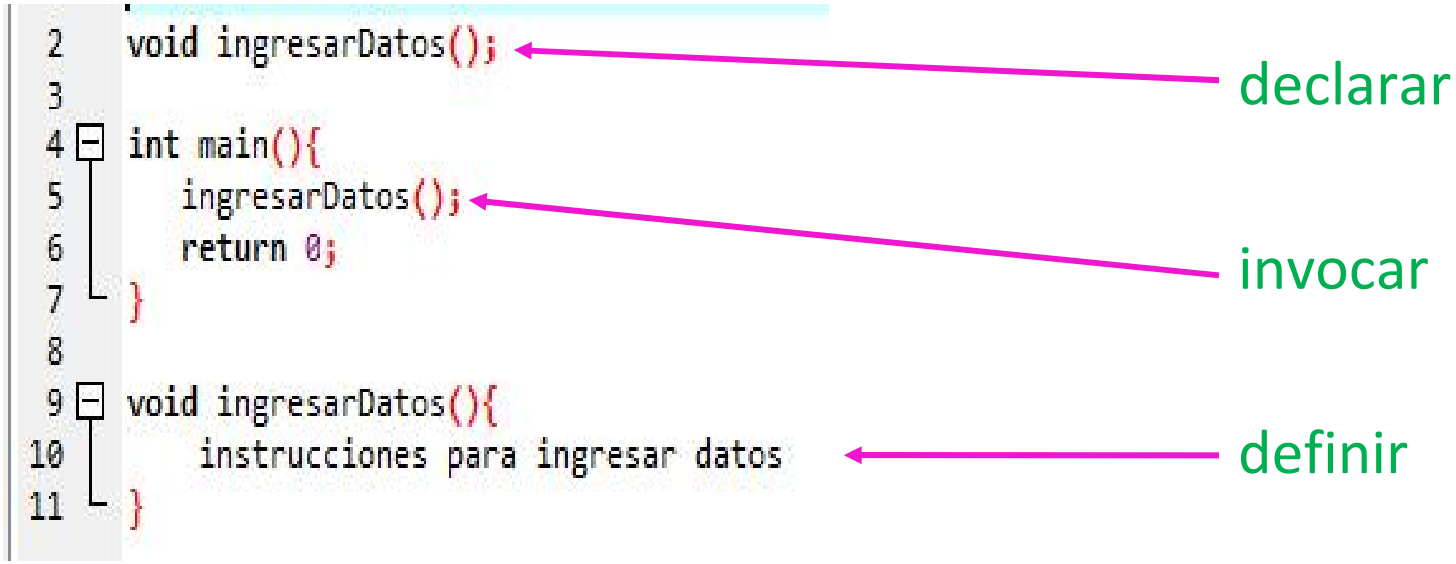
Ejemplo de prototipo:

```
float valorAbsoluto (float);
```

Prototipo de una función

- Se sitúan normalmente al principio del programa, antes de la función main.
- El compilador utiliza los prototipos para validar que el número y los tipos de datos de los parámetros reales son los mismos que el número y los tipos de datos de los parámetros formales de la función llamada. En caso de inconsistencia se visualiza un error.

Implementación



Declaración, invocación y definición (desarrollo)

```c
#include <stdio.h>
float  valorAbsoluto (float); /* prototipo de la función */
int main() {
    float v, a;
    printf("Ingrese un valor numérico: ");
    scanf("%f", &v);
    /* invoco a la función */
    a = valorAbsoluto(v);
    printf("El valor absoluto de %f es %f", v, a);
    return 0;
}
/* desarrollo de la función */
float valorAbsoluto (float d) {
    float valorDevuelto;
    valorDevuelto = d;
        if (d < 0) {
            valorDevuelto = -valorDevuelto;
        }
    return valorDevuelto;
}
```

Los parámetros

Los parámetros permiten que el programa y los procedimientos y funciones puedan comunicarse entre sí intercambiando información. De esta forma las instrucciones y expresiones componentes de los subprogramas se aplican sobre los datos enviados en cada llamada ofreciendo una flexibilidad superior a los subprogramas sin parámetros. Al mismo tiempo, si la ejecución de los subprogramas produce resultados necesarios en el punto de la llamada, los parámetros pueden actuar como el medio de transmisión de esos resultados.

- Son canales de comunicación para pasar datos ente programas y subprogramas en ambos sentidos.

- Están asociados a variables. constantes, expresiones, etc., y por tanto, se indican mediante los correspondientes identificadores o expresiones.

- Se utilizan en la llamada o invocación al subprograma se denominan parámetros actuales, reales o argumentos, y son los que entregan la información al subprograma.

- Los parámetros que la reciben en el subprograma se denominan parámetros formales o ficticios y se declaran en la cabecera del subprograma.

- Los parámetros son opcionales y si no se necesitan no se deben usar. Para utilizarlos es necesario declararlos:
- **void nombre (lista de parámetros);**

- En una llamada a un subprograma tiene que verificarse que:
  - El número de parámetros formales debe ser igual al de actuales.
  - Los parámetros que ocupen el mismo orden en cada una de las Listas deben ser compatibles en tipo.

Parámetros formales y argumentos

- Los **parámetros formales** son las variables que se **declaran** dentro del paréntesis junto con la función.
- Los valores con que se **invoca** la función se denominan **argumentos o parámetros reales o parámetros actuales**.
- Es usual, denominar **parámetros** a los formales y **argumentos** a los reales.
- Los parámetros son variables siempre.
- Los argumentos pueden ser variables, constantes, expresiones aritméticas o valores de otra función.
- Los argumentos deben coincidir en cantidad y tipo de dato de los parámetros formales

```
1   float promedio(float, float);
2
3   int main(){
4       declarar n1 n2 y promeAlum de tipo float
5       ingresar n1 y n2
6
7       promeAlum = promedio(n1, n2);
8
9       mostrar promedio
10
11      return 0;
12  }
13
14  float promedio(float nota1, float nota2){
15      float prome;
16      prome = (nota1 + nota2)/2;
17      return prome;
18  }
```

Parámetros formales o parámetros

```
1   float promedio(float, float);
2
3   int main(){
4       declarar n1 n2 y promeAlum de tipo float
5       ingresar n1 y n2
6
7       promeAlum = promedio(n1, n2);
8
9       mostrar promedio
10
11      return 0;
12  }
13
14  float promedio(float nota1, float nota2){
15      float prome;
16      prome = (nota1 + nota2)/2;
17      return prome;
18  }
```

Parámetros reales o argumentos

Formas de implementación

1. **Funciones sin paso de parámetros.** Son subprogramas que no requieren información adicional de su entorno, pues simplemente ejecutan una acción cada vez que son invocadas.

2. **Funciones con paso de parámetros.** Para la ejecución de estos subprogramas se requiere además de su invocación, que se le pase información adicional de su entorno.

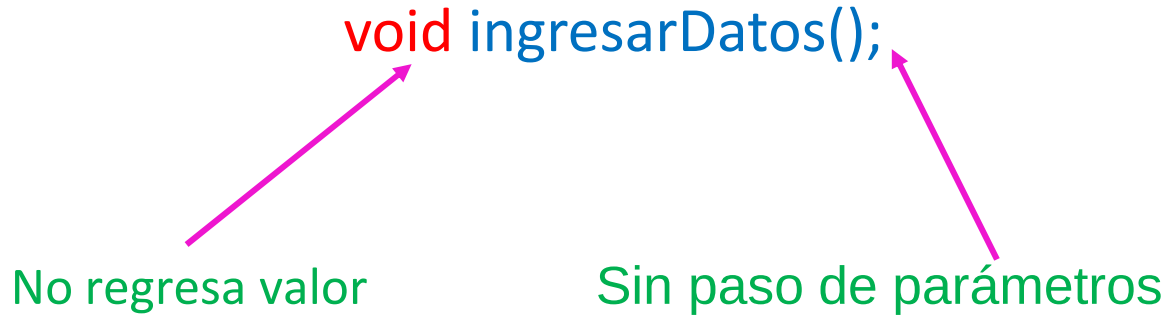
3. **Funciones que no regresan valor.** Subprogramas que luego de su ejecución no devuelven al entorno algún valor como resultado de su ejecución.

4. **Funciones que regresan valor.** Funciones que luego de su ejecución generan un valor como resultado y "entregan" ese valor a su entorno.

Formas de implementación

Funciones que no regresan valor. No devuelven al entorno ningún valor como resultado de su ejecución

Funciones sin paso de parámetros. No requieren información adicional de su entorno.



Formas de implementación

Funciones que regresan valor. Generan un valor como resultado y "entregan" ese valor a su entorno.

Regresa valor

float perimetro();

int suma();

```
1   float promedio(float, float);
2
3   int main(){
4
5       promeAlum = promedio(n1, n2);
6
7       return 0;
8   }
```

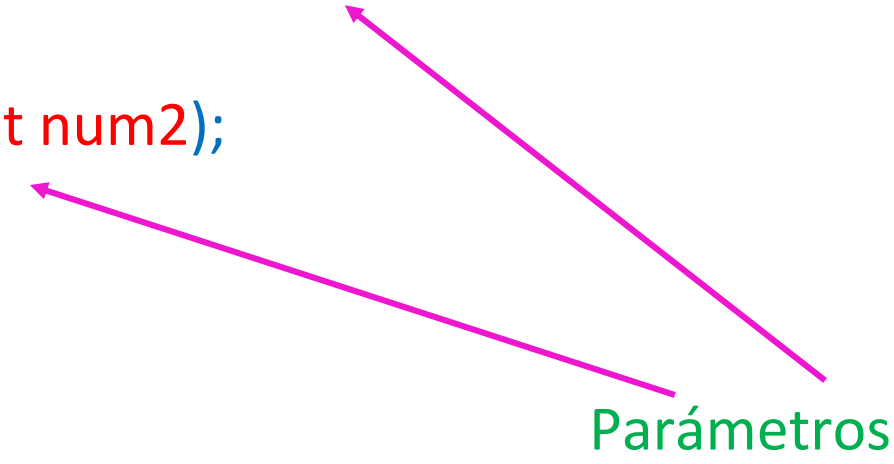
"entrega" valor a su entorno

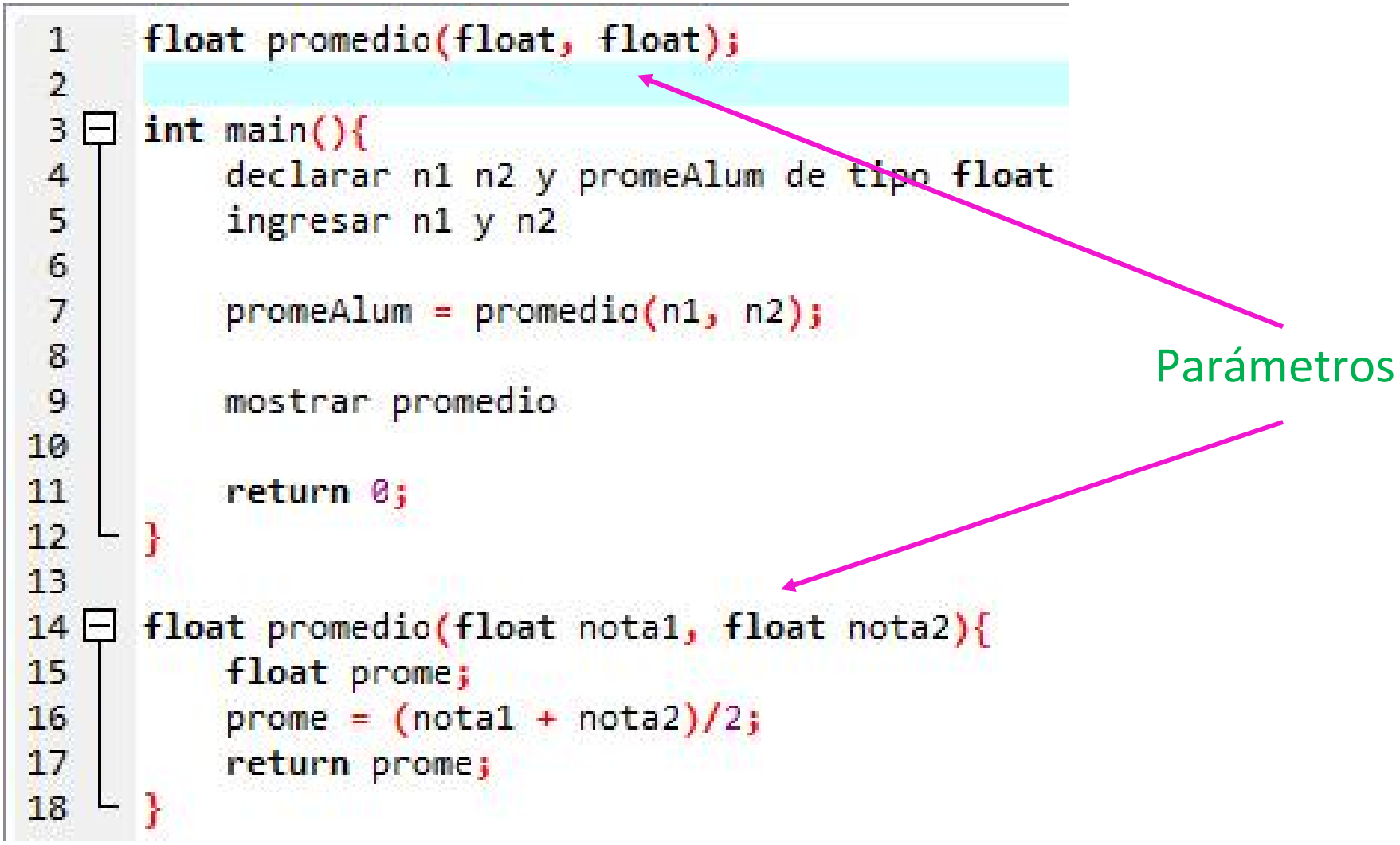
Formas de implementación

Funciones con paso de parámetros. Requiere que se le pase información adicional de su entorno.

```
float promedio(float nota1, float nota2);

void resta(int num1, int num2);
```




1 float promedio(float, float);
2
3 int main(){
4 declarar n1 n2 y promeAlum de tipo float
5 ingresar n1 y n2
6
7 promeAlum = promedio(n1, n2);
8
9 mostrar promedio
10
11 return 0;
12 }
13
14 float promedio(float nota1, float nota2){
15 float prome;
16 prome = (nota1 + nota2)/2;
17 return prome;
18 }
Parámetros

Mecanismos de paso de parámetros

Parámetros por valor: En este caso, se calcula el valor de los parámetros reales y después se copia su valor en los formales, por lo tanto los parámetros reales deben ser expresiones cuyo valor pueda ser calculado. Este mecanismo se llama paso de parámetros por valor y tiene como consecuencia que, si se modifican los parámetros formales en el cuerpo del subprograma, los parámetros reales no se ven afectados. Dicho de otra forma, no hay transferencia de información desde el subprograma al programa en el punto de su llamada. Por lo tanto, los parámetros por valor actúan sólo como datos de entrada al subprograma.

Parámetros por referencia (o por dirección o por variable):: En este otro caso, se hacen coincidir en el mismo espacio de memoria los parámetros reales y los formales, luego los parámetros reales han de ser variables. Este segundo mecanismo se denomina paso de parámetros por referencia (también por dirección o por variable), y tiene como consecuencia que toda modificación de los parámetros formales se efectúa directamente sobre los parámetros reales, y esos cambios permanecen al finalizar la llamada.

Paso de parámetros por valor

Los parámetros por valor reciben una copia de lo que valen los argumentos; su manipulación es independiente, es decir, una vez finalizada la función, los argumentos continúan con el valor que tenían antes.

Este tipo de manejo de parámetros se denomina **función con paso de parámetros por valor**.

```
#include<stdio.h>
int intercambio(int, int);

int main() {
        int n1, n2;
        scanf("%d %d", &n1, &n2);
        intercambio(n1,n2);
        printf("Los valores son n1 = %d y n2 = %d", n1, n2);
return 0;
}

int intercambio(int x, int y) {
        int aux;
        aux=x;
        x=y;
        y=aux;
return 0;
}
```

Parámetros por referencia

El paso de parámetros por referencia implica utilizar los operadores **&** y *****.

El operador & se antepone a una variable, dando con ello acceso a su dirección de memoria asignada. Se utiliza en los argumentos para pasar por referencia dicha variable. El ejemplo clásico de una función de este tipo es **scanf**, donde los valores introducidos por medio del teclado son almacenados por referencia en la variable o variables indicadas; cuando se utiliza la función, se antepone el operador **&** antes del identificador de cada variable.

```
scanf (&a, &b, &c);
```

Luego de recibir las variables argumento, lo que se almacena en ellas permanece incluso después de finalizada la función, dado que pasaron por referencia.

El operador * es un apuntador que "apunta" a la dirección de la variable pasada como argumento. Se utiliza tanto en la declaración de los parámetros formales de la función como en el cuerpo de la misma. Debe aparecer antes del nombre de un parámetro formal en la cabecera para indicar que dicho parámetro será pasado por referencia, y debe aparecer en el cuerpo de la función antepuesto al nombre de un parámetro formal para acceder al valor de la variable externa a la función y referenciada por el parámetro formal.

```
#include<stdio.h>

int intercambio(int *, int *);

int main() {
	int n1, n2;
	scanf("%d %d", &n1, &n2);
	intercambio(&n1,&n2);
	printf("Los valores son n1 = %d y n2 = %d", n1, n2);
}

int intercambio(int *x, int *y) {
			int aux;
			aux=*x;
			*x=*y;
			*y=aux;
return 0;
}
```

Ej.: De uso de Funciones

Se desea calcular el área y el perímetro de una superficie rectangular con las siguientes condiciones:

- Los valores de la base y la altura se leerán en el programa principal.

- Dividir el programa en las siguientes funciones:

    - Una función que retorne el área --> A = base * altura
    - Una función que retorne el perímetro --> P = 2 * (base + altura)
    - Visualización de los resultados.

Programa Principal

```
#include <stdio.h>
float base, altura;

float area();
float perimetro();
int mostrarResultados();

int main (){
        printf("Base: ");
        scanf("%f", &base);

        printf("Altura: ");
        scanf("%f", &altura);
        mostrarResultados();

return 0;
}
```

Función ÁREA

```
float area(){
        return (base * altura);
}
```

Función PERIMETRO

```
float perimetro(){
        return (2 * (base + altura));
}
```

Función MOSTRAR RESULTADOS

```
int mostrarResultados(){
    printf("\nArea = %.2f\n", area());
    printf("\nPerimetro = %.2f\n", perimetro());
return 0;
}
```

Ámbito y visibilidad de los identificadores

Los identificadores declarados o definidos en el programa principal, se denominan globales, y su ámbito es (son visibles en) todo el programa, incluso dentro de los subprogramas (excepto si en éstos se declara una variable con el mismo identificador, la cual ocultará a la variable global homónima.

Los identificadores declarados o definidos dentro de subprogramas se denominan locales, sólo son válidos dentro de los subprogramas a los que pertenecen y, por tanto, no son reconocidos fuera de ellos (es decir, quedan ocultos al resto del programa).

Si dentro de un subprograma se define otro, se dice que los parámetros locales del subprograma superior se denominan no locales con respecto al subprograma subordinado y son visibles dentro de ambos subprogramas.

Los objetos globales se crean al ejecutarse el programa y permanecen definidos hasta que éste termina. En cambio, los objetos locales se crean en el momento de producirse la llamada al subprograma al que pertenecen y se destruyen al terminar éste.

ÁMBITO: VARIABLES LOCALES Y GLOBALES

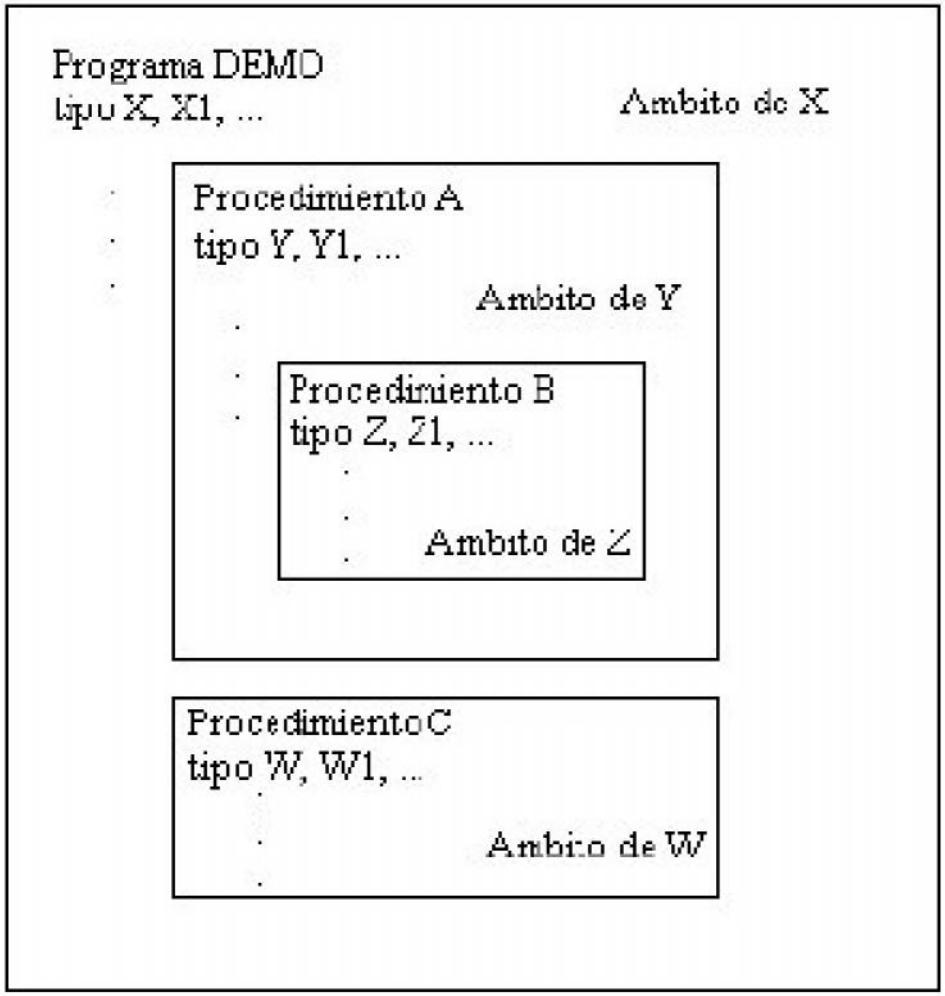
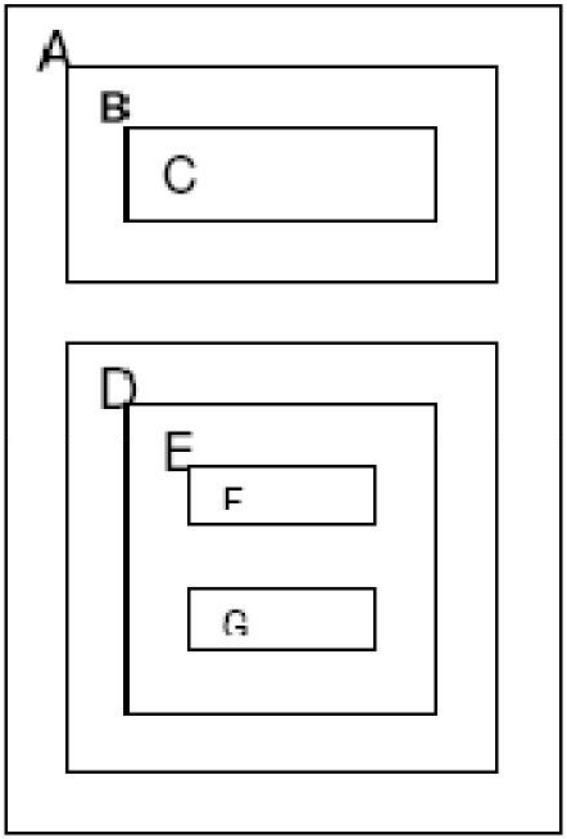


Figura 5.4. Ambito de identificadores.

ÁMBITO: VARIABLES LOCALES Y GLOBALES



Variables definidas en	Accesibles desde
A	A, B, C, D, E, F, G.
B	B, C.
C	C
D	D, E, F, G.
E	E, F, G.
F	F
G	G

Criterios de localidad

- Los diferentes ámbitos de validez de los identificadores, correctamente utilizados, permiten alcanzar una gran independencia entre el programa principal y sus subprogramas, y entre éstos y los subprogramas en ellos contenidos.

- De esta forma se puede modificar un subprograma sin tener que cambiar los demás, facilitando tanto el diseño del programa como posteriormente su depuración y mantenimiento.

- Facilitan la utilización de subprogramas ya creados (bibliotecas de subprogramas) dentro de nuevos programas, eliminando las posibles interferencias entre los objetos del programa y los de los subprogramas.

- Condiciones:

  Principio de máxima localidad: Todos los objetos particulares de un subprograma, necesarios para que desempeñe su cometido, deben ser locales al mismo.

  Principio de autonomía de los subprogramas: La comunicación con el exterior debe realizarse exclusivamente mediante parámetros, evitándose dentro de los subprogramas toda referencia a objetos globales.

Efectos laterales

- Hemos visto distintos mecanismos por los cuales un procedimiento o función pueden devolver o enviar resultados al programa principal (o a otro procedimiento o función).

- Tanto para los procedimientos como para las funciones, dichos valores pueden enviarse mediante parámetros por referencia.

- Una tercera vía consiste en utilizar las variables globales, porque dichas variables son reconocidas en cualquier lugar del bloque. Dicha asignación es correcta, al menos desde el punto de vista sintáctico.

- Sin embargo, esta ultima posibilidad merma la autonomía de los subprogramas, y es perjudicial porque puede introducir cambios en variables globales y errores difíciles de detectar.

Otras recomendaciones sobre el uso de parámetros

Parámetros por valor y por referencia

- Se recomienda emplear parámetros por valor siempre que sea posible (los argumentos no se alteran) y reservar los parámetros por referencia para aquellos casos en que sea necesario por utilizarse como parámetros de salida.

- Cuando se trabaja sobre datos estructurados grandes (ej. vectores o matrices) puede estar justificado pasar dichas estructuras por referencia, aunque solamente se utilicen como parámetros de entrada, porque de esta forma no hay que duplicar el espacio en la memoria para copiar la estructura local, sino que ambas comparten la misma posición de memoria. También se ahorra el tiempo necesario para copiar de la estructura global a la local.

Desarrollo correcto de subprogramas

- Hay dos aspectos de interés:
  - la llamada es la que efectúa un cierto encargo",
  - y la definición la que debe cumplimentarlo.
- El necesario acuerdo entre definición y llamada se garantiza por medio de la especificación, mas o menos formalmente.
- En lo que respecta a la definición, para asegurar la corrección de un subprograma lo consideraremos como lo que es: un pequeño programa. Así, la tarea esencial en cuanto al estudio de su corrección consistirá en considerar sus instrucciones componentes y garantizar que cumple con su cometido, que es el descrito en la especificación.
- Para cada subprograma especificaremos su interfaz de una forma semi-formal. Para ello explicitaremos al principio de cada subprograma una precondición que describa lo que precisa el subprograma para una ejecución correcta y una poscondición que indique los efectos que produciría.

```
function Fac(n: integer): integer;
  {PreC.: 0 ≤ n ≤ 7 y n ≤ MaxInt}
  {Devuelve n!}
  var
    i, prodAcum: integer;
begin
  prodAcum:= 1;
    {Inv.: i ≤ n y prodAcum = i! }
  for i:= 2 to n do
    prodAcum:= prodAcum * i;
    {prodAcum = n!}
  Fac:= prodAcum
    {Fac = n!}
end; {Fac}
```

Programación Estructurada

• Los avances tecnológicos no necesariamente están acompañados por una evolución en las técnicas de **construcción** de programas.

• Se requieren programas mas complejos y de gran tamaño.

Diseño Descendente o Diseño Top-Down. Consiste en una serie de descomposiciones sucesivas del problema inicial, que describen el refinamiento progresivo del conjunto de instrucciones que van a formar parte del programa.

• Las técnicas de desarrollo y diseño de programas, que se utilizan en la programación convencional o lineal, tienen inconvenientes, sobre todo a la hora de verificar y modificar programas.

• Técnicas que facilitan la *comprensión del programa:* **Programación estructurada**

• Pioneros de esta metodología: Dijkstra, Warnier, Jackson, Chapin y Bertini; y de los Lenguajes de Programación Estructurada Niklaus Wirth, Dennis Ritchie y Kenneth Thompson.

Estructura: Es la descomposición ordenada de las partes de un todo. Conjunto de elementos interrelacionados que forman un todo.

Programación Estructurada: Consiste en el diseño, escritura y prueba de un programa, construido con estructura, o sea con una organización ordenada del todo en partes interdependientes.

Herramientas. Estructuras Básicas. Figuras Lógicas. Teorema de la Estructura

Aporte de BOHN y JACOPINI (1966). Matemáticamente todo problema puede resolverse con tres figuras lógicas (secuencia, alternativa y repetitiva o iterativa).

La programación estructurada se consigue mediante la descomposición en partes, segmentos, rutinas o subproblemas interdependientes; que pueden ser considerados como las unidades razonablemente pequeñas, que deben representar por si mismas un contenido lógico, coherente y completo.

Esta segmentación debe realizarse de lo general a lo particular, organizando por niveles con grado de detalle creciente, hasta lograr que todos los tratamientos hayan sido explícitamente considerados.

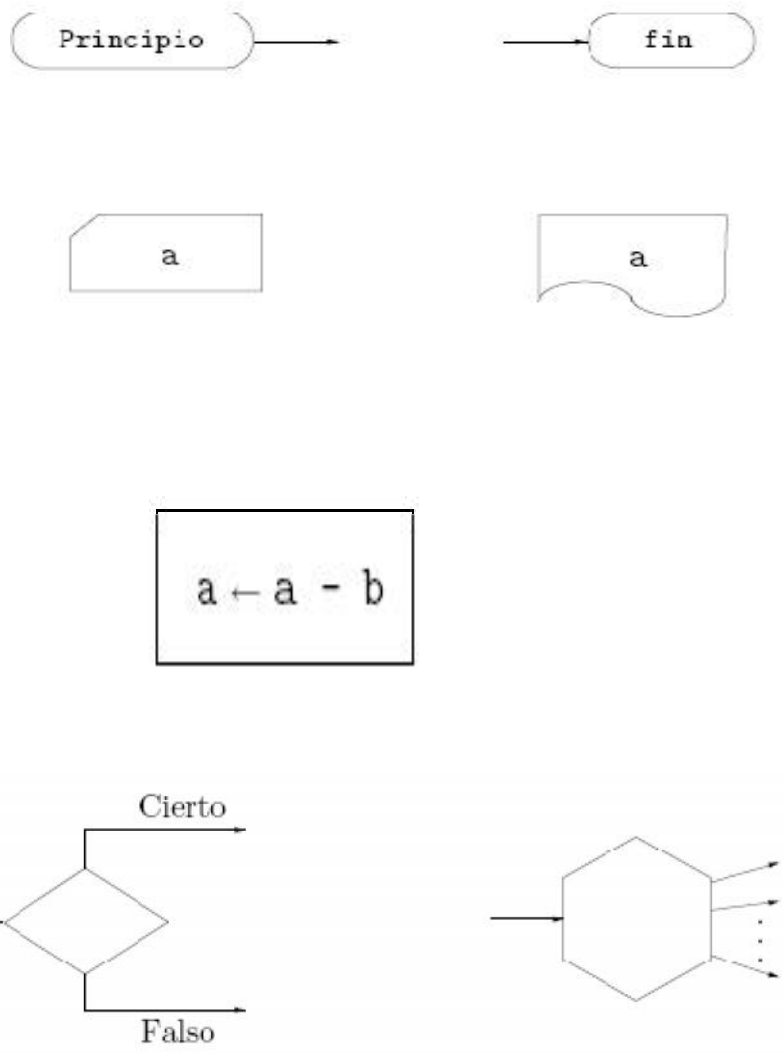
Esto se puede esquematizar de distintas maneras (conjuntos, segmentos, llaves).

Este procedimiento recibe el nombre de **refinamiento paso a paso**, el lema didáctico es: **“divide y será fácil”**.

Programación estructurada

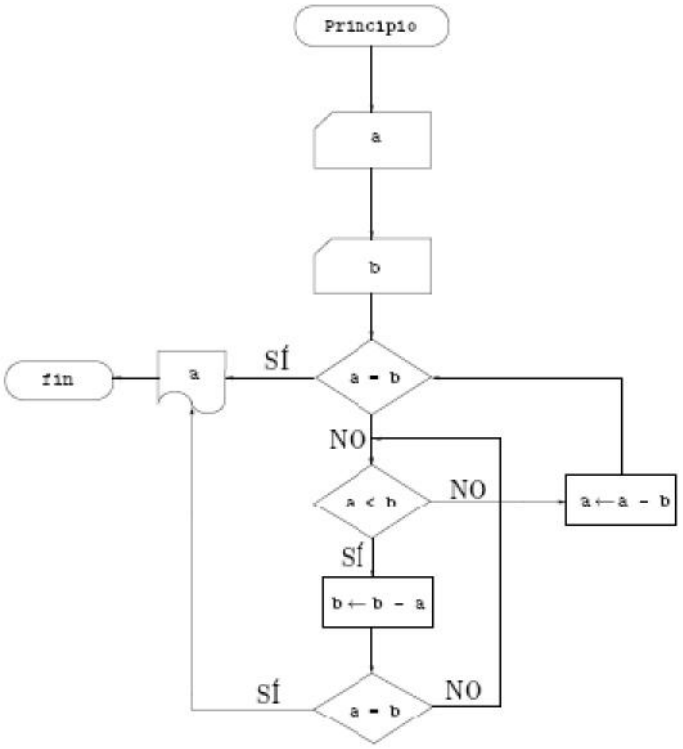
Los bloques de un diagrama de flujo pueden ser de cuatro clases distintas

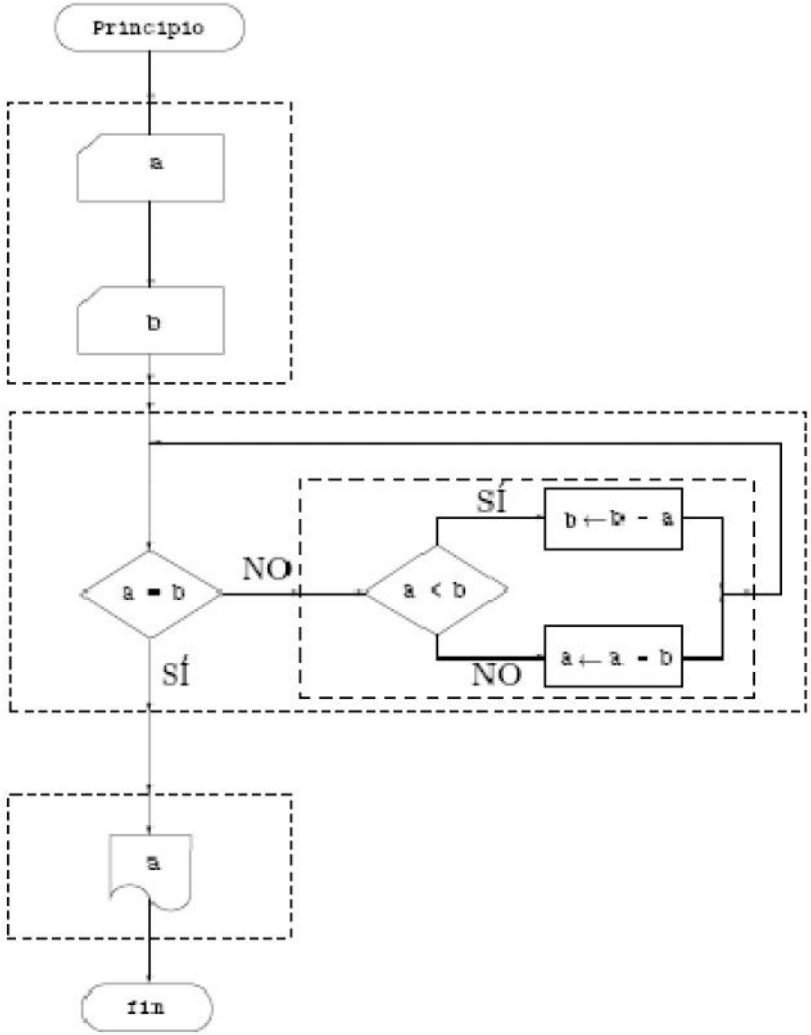
- Símbolos terminales, que indican el principio y el final del algoritmo. Se representan usando óvalos.
- Símbolos de entrada y salida de datos. Respectivamente, significan lectura y escritura.
- Bloques de procesamiento de datos, que realizan operaciones con los datos leídos o con datos privados. Se representan mediante rectángulos que encierran la especificación del proceso.
- Nudos de decisión, en los que se elige entre dos o más alternativas. Según las alternativas sean dos (generalmente dependiendo de una expresión lógica) o más de dos.



Programación estructurada

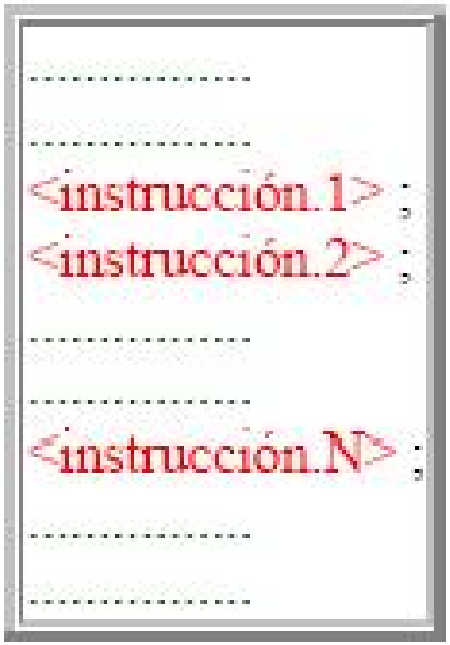
Diagramas y diagramas propios





Figuras Lógicas

Secuencia: En este caso, las instrucciones se ejecutan una después de la otra sin omitir ninguna de ellas.




```
Begin
        N := 0;
        Write (“Nombrey Apellido”, NYA);
        N := N + 1
End.
```

Figuras Lógicas

Selección: La selección de alternativas en Pascal se realiza con alguna de las dos siguientes formas :

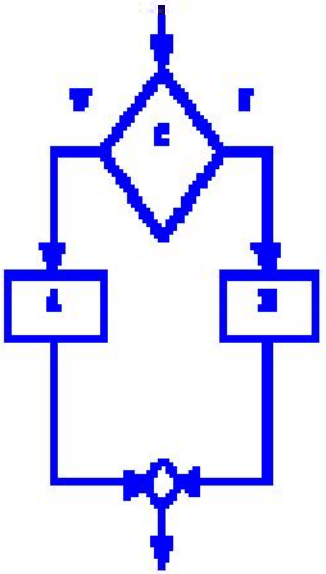


Diagrama de Flujo

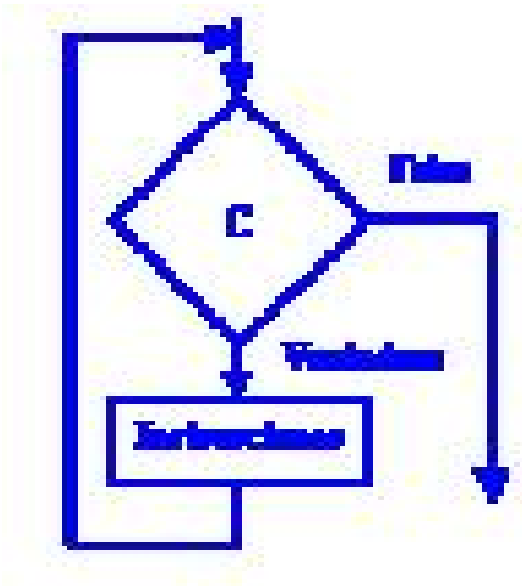
Formato de la sentencia IF :

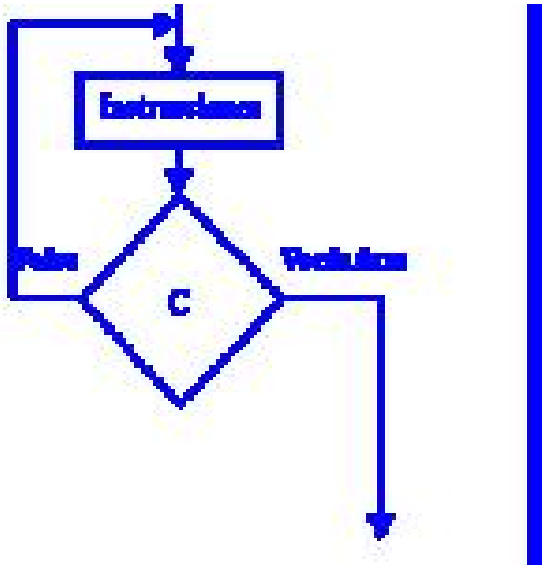
```
then
    <instrucción_1>
else
    <instrucción_2>
```

Figuras Lógicas

Iteración: Las formas de iteración sirven para ejecutar ciclos repetidamente, dependiendo de que se cumplan ciertas condiciones. Una estructura de control que permite la repetición de una serie determinada de sentencias se denomina bucle1 (lazo o ciclo).

El cuerpo del bucle contiene las sentencias que se repiten. Pascal proporciona tres estructuras o sentencias de control para especificar la repetición:





Teorema de la Estructura

Se basa en el concepto de diagrama o programa propio, el mismo consiste en que toda estructura tenga un solo punto de entrada y un solo punto de salida.

Un diagrama es estructurado si tiene únicamente como figuras lógicas: secuencia, capacidad de decisión y capacidad de repetición; y si además está construido con estructura.

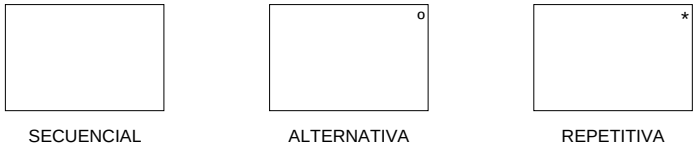
Recursos abstractos: descomponer una tarea compleja en acciones mas simples. Se debe abstraer del como se va a resolver una determinada rutina.

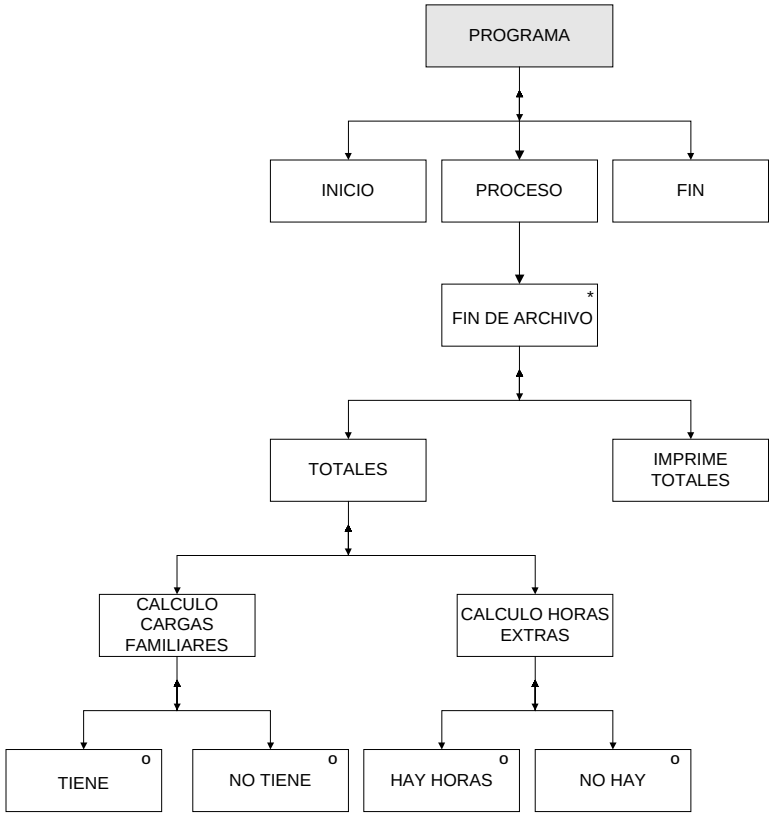
Diseño descendente; descomposición del problema de arriba hacia abajo en forma analítico deductivo, avanzado de lo general a lo particular ,organizado por niveles con grado de detalle creciente, hasta lograr que todos los tratamientos hayan explícitamente considerados.

Métodos

Con la filosofía de la programación estructurada distintos autores han desarrollados trabajos que varían unos de otros en el método, pero todos tienen como bases los conceptos teóricos de Dijkstra.

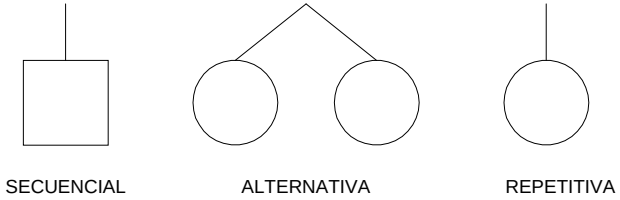
Método Jackson. La filosofía del mismo tiene características comunes con la programación estructurada. Las estructuras básicas se llaman diagramas de bloques y se representan con los símbolo. La representación de la secuencia de acciones se hace en forma de árbol, leyéndose de izquierda a derecha a un mismo nivel.

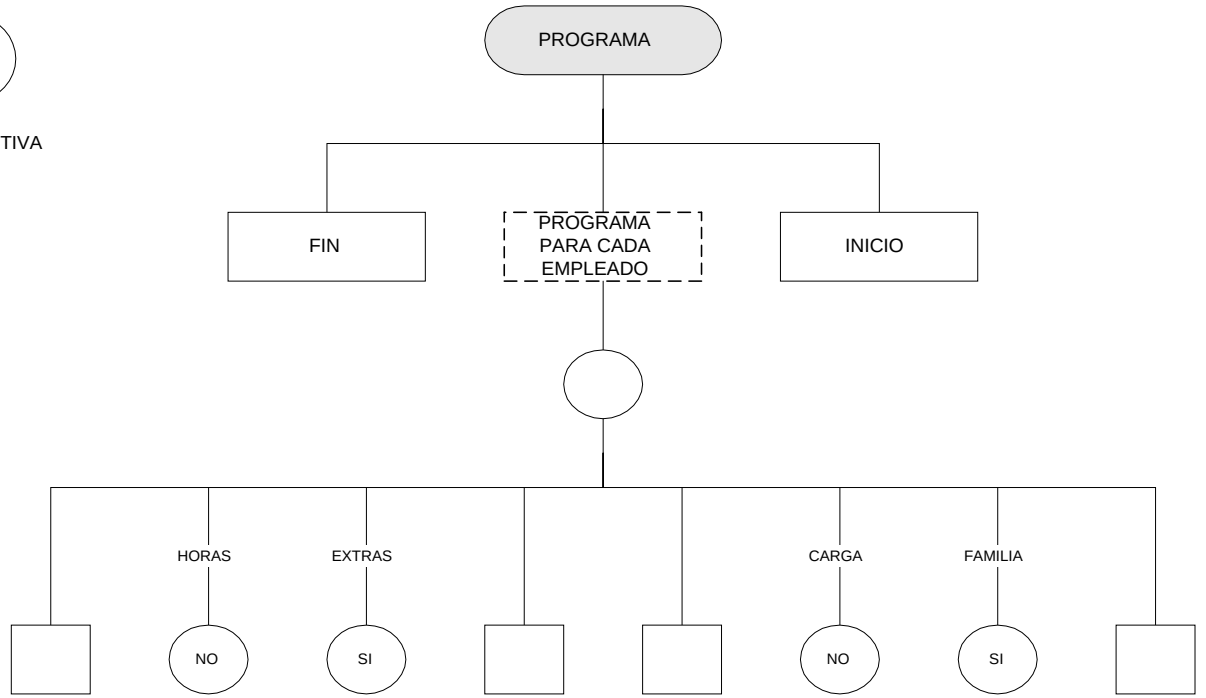




Método Bertini

En esta metodología las estructuras básicas se representan con los siguientes símbolos. Estas estructuras forman los que se denominan árboles programáticos, donde cada nivel es, como en otros modelos, los refinamientos que se van realizando.





Método Warnier

Una de las características de esta manera de hacer programas estructurados consiste en la representación por medio de lo que se denomina cuadro de descomposición en secuencias, forma gráfica que utiliza llaves para representar los niveles de descomposición del problema.

Las estructuras básicas de esta metodología son las siguientes:

- ***Secuencia:*** acciones especificadas que no son ni repeticiones ni alternativas. La secuencia de acciones es de arriba hacia abajo en el cuadro de descomposición de secuencia
- ***Repetitiva:*** En esta estructura, entre paréntesis se escribe el número de veces que se repite

Comparación con programación tradicional

Veamos un ejemplo práctico …

A partir de los datos contenidos en un archivo de Clientes de un Banco se desea obtener un listado del mismo, contarlos y al final informar cuantos clientes tiene el Banco.

```
#include <stdio.h>

void priProg();
void finProg();
void proceso();
void abrirArchivo();
void leerArchivo();
void cerrarArchivo();

    FILE *Fichero;

struct sRegistro {
      int  NumeroCuenta;
      char ApellidoyNombre[35];
      int codOperacion;
      float Importe;
} registro;

                    int contador;
```

```
int main(){
    priProg();
    proceso();
    finProg();
return 0;
}
```

```
 void proceso(){
        while (!feof(Fichero)){
        printf("\n\t%i" "\t\t\t%s" "\t\t%i" "\t\t%.2f" , registro.NumeroCuenta,
                    registro.ApellidoyNombre, registro.codOperacion,
                    registro.Importe);
         contador=contador+1;
         leerArchivo();
      }
 }
```

```
void priProg(){
    contador=0;
    printf("\n\tNumero de Cuenta \tApellido \tCod. Operacion \tImporte");
    abrirArchivo();
    leerArchivo();
}

void abrirArchivo(){
     Fichero=fopen("Clientes.dat", "rb");
}

void leerArchivo(){
                    fread(&registro,sizeof(struct sRegistro), 1,Fichero);
}
```

```
void finProg(){
      printf("\n\nCantidad de Clientes = ");
      printf("%i", contador);
      contador=0;
      cerrarArchivo();
}

     void cerrarArchivo(){
         fclose(Fichero);
         getchar();
         return ;
      }
```

Ventajas

- **Mejor comprensión del problema**: se avanza de lo general a lo particular, segmentando en partes interrelacionadas. Todo programa estructurado puede ser leído desde el principio al fin, sin interrupciones, en la secuencia normal de lectura. Mejor clarificación del problema.

- **Amortigua la lógica individual**: tiende a la construcción lógica normalizada, amortiguando la lógica individual que en principiantes alcanza altos niveles, y después se arraigan en los programadores provocando dificultades en la transferencia del mantenimiento de programas.

- **Facilita el trabajo en equipo**: Las distintas partes o subprogramas pueden encomendarse a distintas personas. Un programa puede hacerse entre varias personas.

- **Propende a la formación de la biblioteca propia:** Puesto que muchas de estas rutinas pueden constituir procesos comunes a otras aplicaciones, con lo que se ahorra tiempo de programación.

- **Facilita la prueba:** Ya que ésta se realiza sobre partes razonablemente pequeñas, lo que hace mas fácil identificar los errores. Se realiza un claro seguimiento del problema.

- **Facilita la legibilidad de la documentación:** Por ende permite una fácil transferencia de la aplicación entre distintos programadores, evitando así los dueños de programas, solo lo entienden los que lo hicieron. El mismo diagrama sirve para documentar.

- **Facilita la optimización, el mantenimiento y las modificaciones:** Como está compuesto por partes razonablemente pequeñas, rápidamente individualizables, es más fácil depurar, mantener o modificar las mismas. Las modificaciones afectan solamente algunas partes.

- **Inconveniente**: Mayor tiempo de ejecución y mayor número de instrucciones

Ventajas

Se cometen menos errores: Por la misma naturaleza del método, que divide en partes al problema, y cada una de esas partes tienen menos instrucciones que el diagrama hecho en forma convencional.

La demostración teórica es la siguiente : **e = k * n2** (1)

Donde :

e: cantidad de errores que se cometen en un programa

k: factor de proporcionalidad que depende de la experiencia del programador

n: número de instrucciones del programa

O sea la cantidad de errores es directamente proporcional al cuadrado del número de instrucciones y a la experiencia del programador.

Supongamos que el programa de n instrucciones lo dividimos en dos subprogramas de n1 y n2 instrucciones respectivamente, donde $n \cong n1 + n2$ y en el que la cantidad de errores de cada subprograma es:

$$e1 = k * n1^{\mathbf{2}}$$

$$e2 = k * n2^{\mathbf{2}}$$

Si sumamos: **et** = $e1 + e2 = k\ (n1^{\mathbf{2}} + n2^{\mathbf{2}})$ (2)

Si en la fórmula (1) reemplazamos n por su equivalente n1 + n2, se obtiene:

$$\mathbf{e} = k\ (n1 + n2)2 = k\ (n12 + n22 + 2\ n1\ n2) = k\ (n12 + n22) + k\ 2\ n1\ n2 \qquad (3)$$

Si restamos al error del programa **e** (3), el error de los subprogramas **et** (2), se tiene:

$$\mathbf{\Delta = e - et = k\ 2\ n1\ n2}$$

Donde Δ es la cantidad mayor de errores que se cometen en la programación tradicional respecto de la estructurada.

Esta deducción nos dice que cuando mayor sea la cantidad de subprogramas en que se divide un programa tradicional, mayor será la diferencia Δ.

Conclusión

- La programación estructurada es una disciplina de programación desarrollada sobre los siguientes principios básicos:

1. La percepción de una estructura lógica en el problema. Esta estructura debe reflejarse en las acciones y datos involucrados en el algoritmo diseñado para la solución.

2. La realización de esa estructura mediante un proceso de refinamiento progresivo, abordando en cada momento únicamente un aspecto del problema.

3. El uso de una notación que asista al refinamiento progresivo de la estructura requerida.

Estructura de Datos: Archivos

Repaso: Archivos. Objetivos. Concepto General. Operaciones. Organización. Acceso.

Objetivo

Las estructuras de datos vistas: son definidas en el algoritmo y ocupan memoria RAM, se utilizan en la ejecución y todos los valores contenidos en ellas se pierden, a lo sumo, cuando el algoritmo finaliza.

Es necesario disponer de otras estructuras de datos que permita guardar su información en soporte volatil (disco, cinta, cd, etc.) y de esa forma preservarla aunque el programa finalice (estructuras de almacenamiento).

Se asocian a un dispostivo de memoria auxiliar permanente.

Memoria principal y secundaria

Estructuras de datos anteriores

=> utilizan memoria interna

=> **memoria principal** (es volátil de acceso aleatorio)

Características:

a) Limitada en la cantidad disponible

b) Es volátil

c) Velocidad de acceso elevada

Estructuras de datos tipo archivo utilizan **memoria secundaria** (almacenamiento secundario magnético u óptico)

=> medios de almacenamiento fuera de la RAM

=> retienen la información (discos, cintas, cd, etc.).

Estructura de Datos: Archivos (ficheros)

Es una colección de registros semejantes, guardados en dispositivos de almacenamientos secundario de la computadora.

Es una estructura de datos que guarda en un dispositivo de almacenamiento secundario de una computadora una colección de elementos del mismo tipo.

En sintesis, un archivo, es una estructura de datos homogenea.

- **ARCHIVOS DE DATOS**
- **ARCHIVOS DE PROGRAMA**

Administración de Archivos

- Cuando se trabaja con un archivo en almacenamiento secundario y estos pueden almacenar un gran numero de archivos.

- Un programa no sabe con exactitud donde residirán los datos (archivo físico).

- El programa envía y recibe información desde y hacia el archivo (archivo lógico).

- Es necesario que el SO realice ciertas tareas de relación entre ambos.

Archivos – Estructura de Datos

- Objetivos (entre otros...)

- Almacenamiento permanente
- Manipulacion de un gran numero de datos
- Independencia de los programas
- Residentes en soportes externos

- Estrutura
- Archivos = Colección de Registros
- Registros = Colección de campos (tamaño, longitud,...)
-

- Clave ➔ Un campo que identifica al registro

Manejo de buffers

- Memoria intermedia entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria o donde los datos residen una vez recuperados de dicha memoria secundaria. Ocupan memoria RAM.

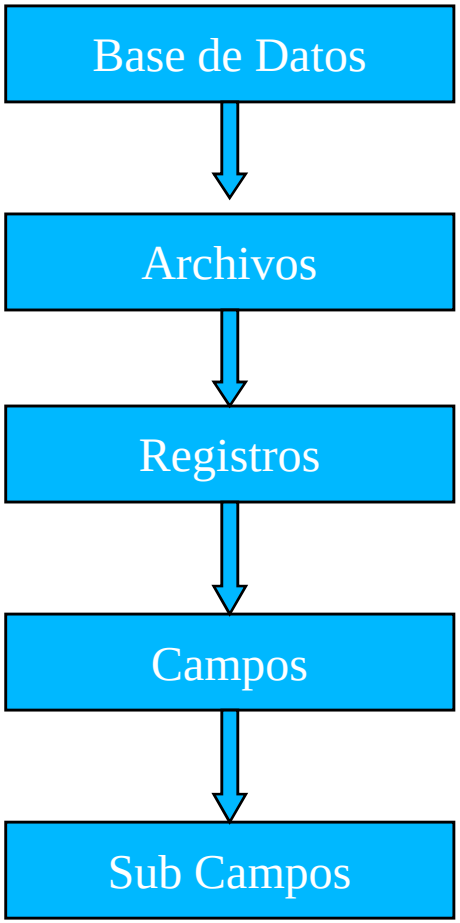
- El SO es el encargado de manipular los buffers.

- Diferentes tiempos de acceso en la memoria principal y memoria secundaria.

Tipos de Registros

- Registro Físico: Cantidad de datos que puede transferirse en una operación de I / O a traves del buffer.
- Registro Lógico: Definido por el programador.
- Factor de Bloqueo: Numero de Registros Lógicos que puede contener un Registro Físico.

Jerarquizacion



Estructura de Datos: Archivos

Campo es un conjunto de caracteres capaz de suministrar una determinada información referida a un concepto. Al igual que en las variables, al definir un campo hay que indicar claramente sus tres características:

***Nombre:* identifica a ese conjunto de caracteres**

***Tipo:* Tipo de caracteres que puede contener (alfabético, entero, etc.-)**

***Tamaño:* Cantidad de caracteres que puede contener**

Por ejemplo, si tenemos que definir al campo número de documento resultaría:

- ***Nombre: documento***
- ***Tipo: numérico***
- ***Tamaño: 8 enteros***

Un campo es la entidad lógica más pequeña, consiste en un conjunto de byte que conforman un dato.

Un campo es la unidad mínima de información de un registro.

Estructura de Datos: Archivos

Registro es un conjunto de campos referentes a una entidad en particular y constituyen una unidad para su proceso. Un ejemplo de un registro puede ser la información de un determinado alumno universitario, que contiene los campos: libreta universitaria, apellido y nombre, número de documento, domicilio, fecha de nacimiento, entre otros campos.

libreta universitaria	Apellido y nombre	número de documento	Domicilio	Fecha de nacimiento

Clasificacion según su funcion

- Maestros: Datos permanentes o históricos.
- De Movimientos: Auxiliares. Contienen registros necesarios para realizar actualizaciones a los archivos permanentes.
- De Maniobras: Efímeros y auxiliares. Contienen información de registros seleccionados o semi elaborados.
- De informes: Contienen datos para ser presentados a los usuarios.

Estructura de Datos: Archivos

Soporte: A) Secuenciales; B) Direccionables;

Un soporte secuencial ➔ Org. secuencial

Un soporte direccionable ➔ Distintos tipos de Org.

Organización de Archivos: La organización de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento, o también se define la organización como la forma en que se estructuran los datos en un archivo. En general, se consideran tres organizaciones fundamentales:

- ***Organización secuencial***
- ***Organización directa o aleatoria (random)***
- ***Organización secuencial indexada***

Obs: En Pascal standar los archivos son de Org. Secuencial

Turbo Pascal permite el acceso aleatorio o directo en todos los archivos (excepto en archivos de textos).

Estructura de Datos: Archivos

Organización secuencial

No es mas que una sucesión de registros almacenados en forma consecutiva sobre un soporte externo.

Los registros están ubicados físicamente en una secuencia usualmente fijada por uno o más campos de control contenidos dentro de cada registro, en forma ascendente o descendente.

Esta organización tiene el último registro en particular, contiene una marca (en general un asterisco) de fin de archivo, la cual se detecta con funciones tipo EOF (end of file) o FDA (Fin de Archivo).

Organización Directa

- Los datos se colocan y se acceden aleatoriamente mediante su posición, es decir, indicando el lugar relativo que ocupan dentro del conjunto de posiciones posibles.

- En esta organización se pueden leer y escribir registros, en cualquier orden y en cualquier lugar.

- Inconvenientes:

a) Establecer la relación entre la posición que ocupa un registro y su contenido;

a) Puede desaprovecharse parte del espacio destinado al archivo.

- Ventaja: Rapidez de acceso a un registro cualquiera.

Organización Indexada

- Un archivo con esta organización consta de tres áreas:

  - Área de índices
  - Área primaria
  - Área de excedentes (*overflow*)

Ventaja:

a) Rápido acceso, y, además, el sistema se encarga de relacionar la posición
b) de cada registro con su contenido por medio del área de índices.
c) Gestiona las áreas de índices y excedentes.

Desventajas:

a) necesidad de espacio adicional para el área de índices.
b) el desaprovechamiento de espacio que resulta al quedar huecos intermedios libres después de sucesivas actualizaciones.

Estructura de Datos: Archivos

Organización secuencial: Archivos de Textos

Son casos particulares de los archivos con organización secuencial. Constan de una serie de lineas, cada una de las cuales se encuentra constituida por una serie de caracteres, separadas por una marca de fin de linea la cual es posible detectar.

Se puede utilizar la instruccion:

Read (id-archivo,var) ….para leer caracter a caracter

Redln (id_archivo, var)… para leer linea a linea

Las instrucciones ***Write y Read*** no solo escriben o leen los datos si no tambien transformacion de los valores.

Ej: valores numericos se convierten en estrng de caracteres antes de escribirse y viceversa.

Estructura de Datos: Archivos

Métodos de acceso

El modo de acceso es la manera de acceder a los registros de un archivo para leer información o para grabar información nueva en el mismo.

Existen fundamentalmente dos formas de acceso:

Acceso secuencial

Se accede a los registros según secuencia física, en el orden es que están escritos. Dicho de otro modo, para acceder al registro N hay que pasar previamente por los N-1 registros anteriores.

Acceso Directo

Permite el acceso a un registro determinado sin tener que pasar previamente por los registros anteriores.

Estructura de Datos: Archivos

Operaciones sobre archivos

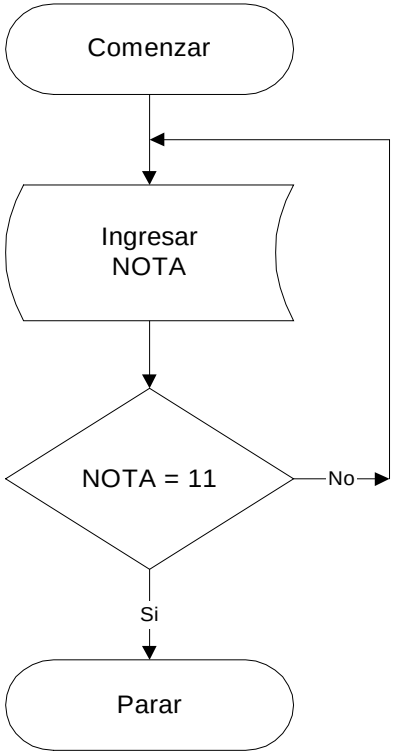
Es necesario considerar las operaciones que se pueden realizar con los mismos. Estas son:

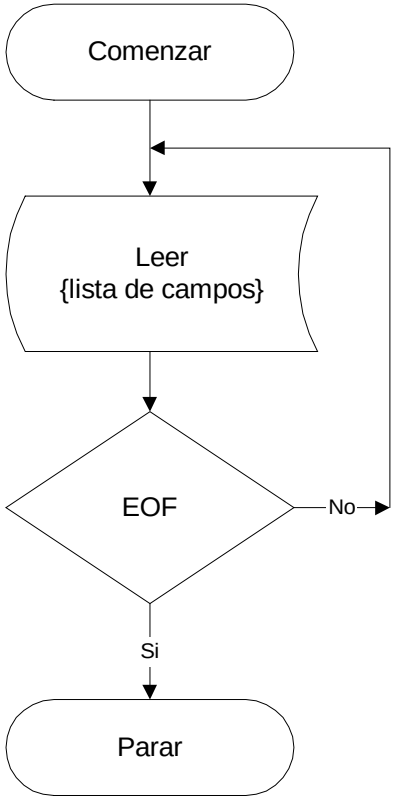
Creación: Definicion del archivo
Apertura: Comunicacion del archivo logico con el fisico.
Cierre: Cerrar la conexion.
Lectura / Consulta: Acceder al archivo para ver su contenido.
Fin de Archivo: Detecta el final del archivo.
Destrucción: Borra el Archivo.
Reorganizacion: Optimiza la estructura.
Fusion: Reune varios archivos en uno solo.
Actualizacion – Alta: Adicion de registros.
Act. - Baja: Eliminacion o borrado logico de registros.
Act. – Modificacion: Altera la informacion del contenido.

Estructura de Datos: Archivos

Fin de Archivo

Toda instrucción de ingreso de datos, ya sea desde teclado o desde un dispositivo magnético, requiere una condición de fin de ingreso de los datos, con el objeto de determinar cuando se han terminado los registros de un archivo o cuando ya no se desean ingresar mas datos desde el teclado. Esta condición la denominamos **fin de archivo** conocida como EOF *(end of file)***.**





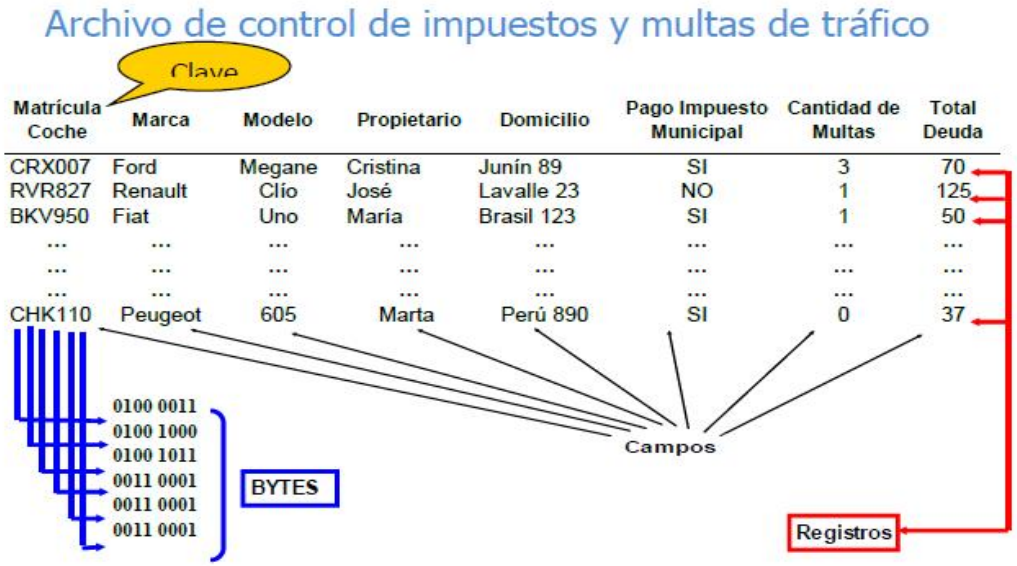


Figura 7.3. Estructura de datos de un archivo

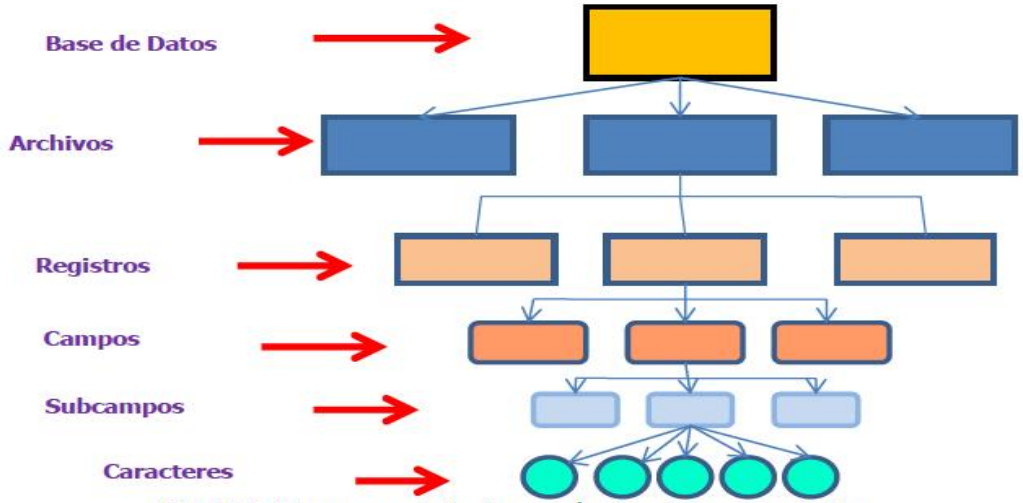


Fig. 7.9. Esquema de la jerarquía de almacenamiento

4.1. Definición de Archivo

Un archivo, fichero o *file*, es una estructura de datos de tipo lógica, compuesta por una secuencia finita de bytes y almacenado en una memoria secundaria.

El Sistema Operativo (SO) es el encargado de interactuar con los archivos, agregando o borrando información, creando nuevos o eliminando los ya existentes. Se les debe asignar:

a) Un nombre, para su identificación.

b) Una ruta: los archivos se agrupan en estructuras y directorios (carpetas) dentro de un sistema de archivos. Los directorios poseen una estructura de árbol, con un nodo raíz y ramificaciones que representan las diferentes carpetas.

Los archivos en C

En C hay dos tipos de archivos, **archivos de texto** y **archivos binarios**.

Un archivo de texto es una secuencia de caracteres organizados en líneas que terminan con un carácter de nueva línea (/n). Los archivos de texto se caracterizan por ser "planos", es decir, contienen solo texto (caracteres) sin información sobre el tipo de letra, ni formato (negrita, subrayados) ni tamaño. Técnicamente cualquier archivo puede abrirse como texto plano desde un editor de texto, por ejemplo, el Bloc de notas de Windows. Por costumbre, los archivos de texto llevan la extensión **.txt.** Se utilizan también para almacenar código fuente en programación, archivos de configuración, etc.

Un archivo binario almacena la información tal cual la representa en forma interna, es una secuencia de bytes que tiene una correspondencia uno a uno con un dispositivo externo. La información contenida en el archivo no es visible directamente como los archivos .txt.

Un fichero en C es sencillamente una colección de bytes que lleva asociado un nombre. El sistema operativo de la computadora administra los archivos por su nombre. Las

- Abrir un archivo
- Cerrar un archivo
- Leer datos de un archivo
- Escribir datos en un archivo
- Añadir datos al final de un archivo

C no tiene palabras reservadas para las operaciones de E/S, estas se realizan a través del uso de la biblioteca de funciones. La siguiente tabla muestra las principales funciones que se utilizan para el manejo de archivos, las que están incluidas en la librería **stdio.h.**

Nombre	Función
fopen()	Abre un archivo
fclose()	Cierra un archivo
fgets()	Lee una cadena de un archivo
fputs()	Escribe una cadena en un archivo
fseek()	Busca un byte especifico de un archivo
fprintf()	Escribe una salida con formato en el archivo
fscanf()	Lee una entrada con formato desde el archivo
fread()	Lee un bloque de datos en archivos binarios
fwrite()	Escribe un bloque de datos en archivos binarios
feof()	Devuelve cierto si es final del archivo
ferror()	Devuelve cierto si se produce un error
fflush()	Vacía un archivo

El puntero a un archivo.

El puntero a un archivo es el hilo común que unifica el sistema de E/S con el buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia identifica un archivo específico y utiliza la secuencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Un puntero a un archivo es una variable de tipo puntero al tipo FILE que se define en stdio.h. Un programa necesita utilizar punteros a archivos para leer o escribir en los mismos. Para obtener una variable de este tipo se utiliza una secuencia como esta:

```
FILE *f;
```

Apertura de un archivo

La función **fopen()** abre una secuencia para que pueda ser utilizada y la asocia a un archivo. Su prototipo es:

```
FILE *fopen(constcharnombre_archivo, constcharmodo);
```

Donde *nombre_archivo* es un puntero a una cadena de caracteres que representa un nombre valido del archivo y puede incluir una especificación del directorio. La cadena a la que apunta *modo* determina como se abre el archivo. La siguiente tabla muestra los valores permitidos para *modo*.

Modo	Significado
r	Abre un archivo de texto para lectura
w	Crea un archivo de texto para escritura
a	Abre un archivo de texto para añadir
rb	Abre un archivo binario para lectura
wb	Crea un archivo binario para escritura
ab	Abre un archivo binario para añadir

La función **fopen()** devuelve un puntero a archivo. Si se produce un error cuando se está intentando abrir un archivo, **fopen()** devuelve un puntero nulo. La macro NULL está definida en **stdio.h**. Este método detecta cualquier error al abrir un archivo: como por ejemplo disco lleno o protegido contra escritura antes de comenzar a escribir en él. Ejemplo:

```
if (architext==NULL {
Printf("No existe el fichero";
Exit(EXIT-FAILURE);
}
```

Cierre de un archivo.

La función **fclose()** cierra una secuencia que fue abierta mediante una llamada a **fopen()**. Escribe toda la información que todavía se encuentra en el buffer del disco y realiza un cierre formal del archivo a nivel del sistema operativo.
Un error en el cierre de una secuencia puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y posibles errores intermitentes en el programa.
El prototipo de esta función es:

```
intfclose(FILE *F);
```

Donde F es el puntero al archivo devuelto por la llamada a fopen(). Si se devuelve un valor cero significa que la operación de cierre ha tenido éxito. Generalmente, esta función solo falla cuando un disco se ha retirado antes de tiempo o cuando no queda espacio libre en el mismo.

Leer y grabar datos en un archivo

Para leer o grabar datos de un archivo tenemos las siguientes cuatro funciones:

fgets() yfputs()

Estas funciones pueden leer y escribir cadenas a o desde los archivos.
Los prototipos de estas funciones son:

```
char *fputs(char *str, FILE *F);
char *fgets(char *str, intlong, FILE *F);
```

La función **fputs()** escribe la cadena a un archivo especifico. La función **fgets()** lee una cadena desde el archivo especificado hasta que lee un carácter de nueva línea o longitud-1 caracteres. Si se produce un EOF (End of File) la función gets retorna un NULL.

fscanf() y fprintf()

Estas funciones realizan la lectura y escritura de variables escalares (es decir tipos de datos que no son compuestos) y se comportan exactamente como **prinft()** y **scanf()**usadas anteriormente, excepto que operan sobre archivo.
Sus prototipos son:

int fprintf (FILE *F, constchar *cadena_de_control, ...);

int fscanf (FILE *F, constchar *cadena_de_control, **.....);**

Donde F es un puntero al archivo devuelto por una llamada a **fopen().**
fprintf() y **fscanf()** dirigen sus operaciones de E/S al archivo al que apunta F.

Función feof()

La función **feof()** se utiliza para detectar cuando no quedan más elementos en un archivo, devolviendo un valor distinto de cero en este caso.
El prototipo de la función es:

int feof (FILE *F);

Devuelve verdadero si detecta fin de archivo, en cualquier otro caso, devuelve 0. Se aplica de la misma forma a archivos de texto y binarios.
Tener en cuenta que esta función no indica fin de fichero hasta que no se intenta volver a leer después de haber leído el último elemento del archivo. Por lo tanto, en el programa debe hacerse la lectura siguiendo estos pasos:

```
Leer un dato del fichero
mientras (!feof(archivo)
        Procesar el dato leído
        Leer un dato del archivo
fin-mientras
```

ferror()

La función **ferror()** determina si se ha producido en error en una operación sobre un archivo. Su prototipo es:

int ferror(FILE *F);

donde F es un puntero a un archivo válido. Devuelve cierto si se ha producido un error durante la última operación sobre el archivo. En caso contrario, devuelve falso. Debido a que cada operación sobre el archivo actualiza la condición de error, se debe llamar a **ferror()** inmediatamente después de la operación de este tipo; si no se hace así, el error puede perderse.

fflush()

La función escribe todos los datos almacenados en el buffer sobre el archivo asociado con un apuntador. Su prototipo es:

int fflush(FILE *F);

Si se llama esta función con un puntero nulo se vacían los buffers de todos los archivos abiertos. Esta función devuelve cero si tiene éxito, en otro caso, devuelve EOF.

Tipo de dato registro: struct

Tal como se mencionó en el tema 4, los datos se clasifican en simples y compuestos. Dentro de estos últimos se encuentran los tipos de datos cadenas, registros y arreglos. Los registros están vinculados a los archivos binarios, dado que permiten el procesamiento de datos estructurados.

Una estructura que permite almacenar diferentes tipos de datos bajo una misma variable se denomina **registro**. Un registro es un "contenedor" de datos de diferentes tipos. Un registro se declara con la palabra reservada **struct**, cuyo formato es el siguiente:

```
struct  datos {
      tipo1 dato1:
      tipo2 dato2;
      /* …otros campos del registro */
} variable1, variable2;
```

```
struct medida {
      char hora:
      char minuto;
      float temperatura;
} registro
```

```
registro.hora=10;
registro.minuto=55;
registro.temperatura=37.5
```

La declaración typedef

En C la declaración **typedef** permite crear nuevos nombres de tipos de datos. Por ejemplo:

```
typedef struct reg {
        char letra;
        int numero;
} REGISTRO;
REGISTRO nuevo, actual [10];
```

En este caso, REGISTRO no es un nombre de variable de tipo reg sino, es un nuevo nombre de tipo de dato que puede utilizarse para definir nuevas variables. En el ejemplo una variable *nuevo* y un vector *actual* de 10 elementos.

Bibliografìa

- **Plantillas de clases de Teoria de la Asignatura. 2015**
- **Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642.**
- **ALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2**
- **ESTRUCTURA DE DATOS; Luis Joyanes Aguilar, 2001,Editorial: MCGRAW-HILL. ISBN:.ISBN: 8448120426.**
- **FUNDAMENTOS DE PROGRAMACIÓN. Libro de Problemas en Pascal y Turbo Pascal; Luis Joyanes Aguilar Luis Rodríguez Baena y Matilde Fernandez Azuela; 1999; Editorial: MCGRAW-HILL. ISBN: 844110900.**
- **ESTRUCTURA DE DATOS; Cairó y Guardati; 2002; Editorial: MCGRAW-HILL. ISBN: 9701035348.**