Tecnicatura Superior en Análisis, Desarrollo y Programación de Aplicaciones
Programación I

"Traduciendo Diagramas de Flujo a Programas C"

Versión 5.0 Julio 2021
Mgter. Guillermo Cherencio

INDICE

Introducción

El programa PseInt (http://pseint.sourceforge.net/) nos permite expresar un algoritmo en su propio lenguaje o bien representarlo a través de un diagrama de flujo. También permite probarlo, ejecutarlo, ejecutarlo paso a paso, etc. es decir, es un entorno de aprendizaje de programación algorítmica muy completo, no obstante, una vez que hemos aprendido los elementos básicos que constituyen un algoritmo (secuencia, selección, iteración) debemos expresarlos en un lenguaje de programación real. En este caso, vamos a pasar del lenguaje de PseInt al Lenguaje ANSI C, para luego poder compilar y linkeditar el programa C y obtener un archivo binario ejecutable que podamos distribuir profesionalmente.

Previo a este apunte se recomienda la lectura de ProgramacionI-2008-sintaxis.pdf (http://www.grch.com.ar/docs/p1/Apuntes/ProgramacionI-2008-sintaxis.pdf) en donde se indican los lineamientos generales de la sintaxis del lenguaje de programación ANSI C.

Secuencia

Proceso

El comienzo de un algoritmo en PseInt se escribe como:

```
Proceso <nombre de proceso>
...
FinProceso
```

esto puede traducirse como el cuerpo principal del programa C y las librerías de uso básico:

```
//comentarios del programa: autor, fecha, objetivo,
//forma de uso, referencias web, etc.

//cabeceras de las bibliotecas a usar en este programa
#include <stdio.h>
#include <stdlib.h>

//prototipo de funciones propias:

int main(int argc, char **argv) {
     // instrucciones del programa principal

     //fin del programa principal, devuelve 0
     //al sistema operativo
     return 0;
}

// implementación de funciones propias
```

Escribir mensaje simple

Un simple mensaje en PSeInt se escribe como:

```
Escribir 'Ingrese un Dato:'
```

puede traducirse como:

```
printf("Ingrese un Dato:\n");
```

\n es una "secuencia de escape" que representa una nueva línea en la consola, es decir, el cursor se posiciona al comienzo de la línea siguiente.

Escribir variables y mensajes

```
Escribir a
Escribir a,b,c
Escribir 'La variable a vale: ',a,' y b vale: ',b
```

puede traducirse como:

```
printf("%d\n",a);
printf("%d %d %d\n",a,b,c);
printf("La variable a vale: %d y b vale: %d\n",a,b);
```

asumiendo -en todos los casos- que a y b son variables de tipo **int** (enteras), pero existen otros tipos de datos, en tal caso:

si a y b fuesen de tipo **char** reemplazar %d por %c

si a y b fuesen de tipo **char *** reemplazar %d por %s

si a y b fuesen de tipo **long** reemplazar %d por %lu

si a y b fuesen de tipo **float** reemplazar %d por %f

si a y b fuesen de tipo **double** reemplazar %d por %lf

Leer variable

```
Leer a
```

puede traducirse como:

```
int a;
scanf("%d",&a);
```

asumiendo que a es de tipo int (entera), sino cambiar la declaración de la variable y %d por lo que corresponda.

Asignar variable

```
a<-4
b<-2
c<-a+b
```

puede traducirse como:

```
int a,b,c;
a=4;
b=2;
c=a+b;
```

asumiendo a,b,c como variables de tipo int (enteras), sino cambiar declaración de variable.

Otros cálculos

```
x<-123.35
a<-4
b<-2
c<-a^b
raizcuadrada<-RAIZ(x)
f<-TRUNC(x)
resto<-x MOD 2
```

puede traducirse como:

```
double x,raizcuadrada;
int a,b,c,f,resto;
x=123.35;
a=4;
b=2;
c=pow(a,b);
raizcuadrada=sqrt(x);
f=(int) x;
resto=x % 2;
```

puesto que C no implementa la potencia como un operador matemático sino como una función y esta función se encuentra en la biblioteca **`math.h`** por lo tanto, se deberá agregar la cabecera **`#include <math.h>`** . Lo mismo sucede con la raíz cuadrada, en este caso, la función se llama **`sqrt()`** y también se encuentra dentro de **`math.h`**. Sin embargo, sucede lo contrario con el módulo o resto de la división entera, en este caso, se implementa no como función sino como operador, en este caso el operador `%`, si escribo **`resto = x % 2`**, significa *"guarde en la variable* **`resto`** *el resto de la división entera de* **`x`** *dividido* **`2`***"*.

Para truncar un número a su parte entera o bien para quedarse con la parte entera de un número, lo más simple en C es forzar la conversión del número al tipo de dato int.

Ejemplos de programas completos

Ejemplo 1

El programa PseInt:

```
Proceso sin_titulo
     Escribir 'Ingrese un numero:'
     Leer nro1
     Escribir 'Ingrese otro numero:'
     Leer nro2
     suma<-nro1+nro2
     Escribir suma
FinProceso
```

puede traducirse en C como:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
     int nro1,nro2,suma;
     printf("Ingrese un numero:\n");
     scanf("%d",&nro1);
     printf("Ingrese otro numero:\n");
     scanf("%d",&nro2);
     suma=nro1+nro2;
     printf("%d\n",suma);
     return 0;
}
```

Ejemplo 2

El programa PseInt :

```
Proceso sin_titulo
     Escribir 'Ingrese cateto b: '
     Leer b
     Escribir 'Ingrese cateto c: '
     Leer c
     a<-RAIZ((c^2)+(b^2))
     Escribir 'la hipotenusa (a) es: ',a
FinProceso
```

puede traducirse en C como:

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

int main(int argc, char **argv) {
     double a,b,c;
     printf("Ingrese cateto b:\n");
     scanf("%lf",&b);
     printf("Ingrese cateto c:\n");
     scanf("%lf",&c);
     a=sqrt(pow(c,2)+pow(b,2));
     printf("la hipotenusa (a) es: %lf\n",a);
     return 0;
}
```

Decisión o Selección

Condición

Las expresiones condicionales en PSeInt pueden expresarse de forma coloquial y ello es interpretado por PSeInt; pero esto difiere de cómo se escribe en lenguaje C, veamos esta tabla de expresiones a modo de ejemplo:

PSeInt		**Lenguaje C**
Expresión Coloquial	**Significado**	
x ES y	x = y	x == y
x ES IGUAL A y	x = y	x == y
x ES DISTINTO DE y	x <> y	x != y
x ES MAYOR QUE y	x > y	x > y
x ES MENOR QUE y	x < y	x < y
x ES MAYOR O IGUAL QUE y	x >= y	x >= y
x ES MENOR O IGUAL QUE y	x <= y	x <= y
x ES CERO	x = 0	x == 0
x ES POSITIVO	x > 0	x > 0
x ES NEGATIVO	x < 0	x < 0
x ES PAR	x MOD 2 = 0	(x % 2) == 0
x ES IMPAR	x MOD 2 = 1	(x % 2) == 1
x ES MULTIPLO DE y	x MOD y = 0	(x % y) == 0
x ES DIVISIBLE POR y	x MOD y = 0	(x % y) == 0
x ES MAYOR O IGUAL A 1 y x ES MENOR O IGUAL A 10	x >= 1 y x <= 10	x >= 1 && x <= 10
x ES MAYOR O IGUAL A 1 o x ES MENOR O IGUAL A 10	x >= 1 o x <= 10	x >= 1 || x <= 10

Si Condición Entonces … FinSi

```
Leer nro
Si nro = 0 Entonces
     Escribir 'Error!, ',nro,' no puede ser cero!'
FinSi
```

puede traducirse como:

```
int nro;
scanf("%d",&nro);
if ( nro == 0 ) {
     printf("Error!, %d no puede ser cero!\n",nro);
}
```

es muy importante en C mantener *"encolumnadas"* las llaves ({ }) para que la estructura condicional (if) quede bien clara en el código del programa.

Si Condición Entonces … Sino … FinSi

```
Leer nro
Si nro = 0 Entonces
     Escribir 'Error!, ',nro,' no puede ser cero!'
Sino
     doble<-nro * 2
     Escribir 'El doble de ',nro,' da ',doble
FinSi
```

puede traducirse como:

```
int nro,doble;
scanf("%d",&nro);
if ( nro == 0 ) {
     printf("Error!, %d no puede ser cero!\n",nro);
} else {
     double=nro*2;
     printf("El doble de %d da %d\n",nro,doble);
}
```

es muy importante en C mantener *"encolumnadas"* las llaves ({ }) para que la estructura condicional (if) quede bien clara en el código del programa.

Si Condición Entonces … Sino … FinSi Anidadas

```
Escribir 'Ingrese Rango Desde'
Leer _desde
Escribir 'Ingrese Rango Hasta'
Leer _hasta
Escribir 'Ingrese Numero entre ',_desde,' y ',_hasta
Leer nro
Si (nro >= _desde) Entonces
     Si ( nro <= _hasta ) Entonces
          Escribir nro,' esta dentro del rango ',
_desde,'-',_hasta
     Sino
          Escribir nro,' NO esta dentro del rango ',
_desde,'-',_hasta
     FinSi
Sino
     Escribir nro,' NO esta dentro del rango ',_desde,'-
',_hasta
FinSi
```

puede traducirse como:

```
int _desde,_hasta,nro;
printf("Ingrese Rango Desde ");
scanf("%d",&_desde);
printf("Ingrese Rango Hasta ");
scanf("%d",&_hasta);
printf("Ingrese Numero entre %d y %d\n",_desde,_hasta);
scanf("%d",&nro);
if (nro >= _desde) {
     if ( nro <= _hasta ) {
          printf("%d esta dentro del rango %d - %d\n",
nro,_desde,'-',_hasta);
     } else {
          printf("%d NO esta dentro del rango %d - %d\n",
nro,_desde,_hasta);
     }
} else {
     printf("%d NO esta dentro del rango %d - %d\n",
nro,_desde,_hasta);
}
```

es muy importante en C mantener *"encolumnadas"* las llaves ({ }) para que la estructura condicional (if) quede bien clara en el código del programa.

Segun variable Hacer ...

Veamos un simple ejemplo de un menú de opciones, que, acorde con la opción elegida por el usuario, el algoritmo tomará por diferentes "caminos":

```
Escribir 'Menu de Opciones'
Escribir '1. Clientes'
Escribir '2. Productos'
Escribir '3. Facturas'
Escribir 'Ingrese su Opcion:'
Leer opcion
Segun opcion Hacer
     1:
          Escribir 'Ingrese Codigo Cliente:' Leer cliente
     2:
          Escribir 'Ingrese Codigo Producto:' Leer producto
     3:
          Escribir 'Ingrese Nro Factura'
          Leer factura
     De Otro Modo:
          Escribir 'Opcion Incorrecta!'
FinSegun
```

puede traducirse como:

```
int opcion,cliente,producto,factura;
printf("Menu de Opciones\n");
printf("1. Clientes\n");
printf("2. Productos\n");
printf("3. Facturas\n");
printf("Ingrese su Opcion:\n");
scanf("%d",&opcion);
switch(opcion) {
     case 1:
          printf("Ingrese Codigo Cliente:\n");
          scanf("%d",&cliente);
          break;
case 2:
          printf("Ingrese Codigo Producto:\n");
          scanf("%d",&producto);
          break;
     case 3:
          printf("Ingrese Nro Factura:\n");
          scanf("%d",&factura);
          break;
     default:
          printf("Opcion Incorrecta!\n");
}
```

La sentencia "break" sirve para terminar abruptamente con cualquier estructura repetitiva, en este caso, dá por terminada la sentencia swtich() . Si no utilizáramos la sentencia break, luego de ingresar por la opción 1 (supongamos) se ejecutaría la opción 2, la opción 3 y default (y esto no es lo deseado en este algoritmo). Es decir, sin break, un switch entra en una opción y a partir de ahí "cae" hasta terminar con la sentencia switch. Debido a esto, generalmente encontrará ejemplos de código en donde por cada valor posible de la variable se incluirá un break.

Repetición o Iteración

Repetir … Hasta Que condición

```
suma<-0
contador<-0
Repetir
     Escribir 'Ingrese un numero (0=salir):'
     Leer nro
     Si nro <> 0 Entonces
          contador<-contador+1
          suma<-suma+nro
     FinSi
Hasta Que nro = 0
Escribir 'Ud ingreso ',contador,' numeros!'
Escribir 'El total de los numeros es ',suma
```

No tiene una traducción directa en lenguaje C, pues C, no cuenta con una estructura repetitiva equivalente, existe un repetir … mientras condición ; en vez de un repetir … hasta; por lo tanto, debemos invertir la lógica de la condición:

```
int suma=0,contador=0,nro;
do {
     printf("Ingrese un numero (0=salir):\n");
     scanf("%d",&nro);
     if ( nro != 0 ) {
          contador=contador+1;
          suma=suma+nro;
     }
} while(nro != 0);
printf("Ud ingreso %d numeros!\n",contador);
printf("El total de los numeros es %d\n",suma);
```

otra forma de "invertir" la lógica de la condición, es utilizando la negación (!): hasta que nro = 0 puede traducirse en do { … } while(!(nro == 0));

Mientras condición Hacer … FinMientras

```
suma<-0
contador<-0
Escribir 'Ingrese un numero (0=salir):'
Leer nro
Mientras nro>0 Hacer
     contador<-contador+1
     suma<-suma+nro
     Escribir 'Ingrese un numero (0=salir):'
     Leer nro
FinMientras
Escribir 'Ud ingreso ',contador,' numeros!'
Escribir 'El total de los numeros es ',suma
```

puede traducirse como:

```
int suma=0,contador=0,nro;
printf("Ingrese un numero (0=salir):\n");
scanf("%d",&nro);
while(nro > 0) {
     contador=contador+1;
     suma=suma+nro;
     printf("Ingrese un numero (0=salir):\n");
scanf("%d",&nro);
}
printf("Ud ingreso %d numeros!\n",contador);
printf("El total de los numeros es %d\n",suma);
```

Para variable ← valor inicial Hasta valor final Hacer … FinPara

Por ejemplo, supongamos que deseamos imprimir los números del 0 al 9 inclusive:

```
Para n<-0 Hasta 9 Hacer
     Escribir n
FinPara
```

en este caso, PSeInt asume un "paso" igual a 1, es decir, que la variable n se incrementa de 1 en 1. Esto puede traducirse en lenguaje C como:

```
int n;
for(n=0;n <= 9;n++) {
     printf("%d\n",n);
}
```

la expresión n++; es equivalente a: n=n+1;
la expresión n--; es equivalente a: n=n-1;
la expresión suma+=nro; es equivalente a: suma=suma+nro;
la expresión suma-=nro; es equivalente a: suma=suma-nro;
la expresión n+=2; es equivalente a: n=n+2;
la expresión n-=2; es equivalente a: n=n-2;
etc (también puede combinar con *, /)

la instrucción **for()** en C se divide en tres partes (todas opcionales): **for(<valor inicial>;<condición>;<incremento o paso>)** . La estructura **for**, primero ejecuta **<valor inicial>** luego, mientras **<condición>** sea verdadera ejecuta su bloque de código y por cada "vuelta" de esta estructura repetitiva se ejecuta **<incremento o paso>**.

En el caso de **for(n=0;n <= 9;n++) { printf("%d\n",n); }** primero pone a la variable **n** (que tiene que estar previamente declarada) con valor igual a 0 (cero), luego pregunta si n es menor o igual que 9, como esto es verdadero, entonces hace el **printf()** para mostrar el valor actual de n, luego de ello, incrementa en uno más el valor de **n** (**n++;**) y vuelve a preguntar y así sucesivamente hasta que la pregunta es falsa y n termina valiendo 10 ! (pues lo último que se ejecuta es **n++**).

Para variable ← valor inicial Hasta valor final Con Paso paso Hacer … FinPara

Por ejemplo, supongamos que deseamos imprimir los números pares del 2 al 10 inclusive:

```
Para n<-2 Hasta 10 Con Paso 2 Hacer
     Escribir n
FinPara
```

en este caso, la idea es incrementar a la variable n “de 2 en 2” para ir obteniendo los distintos números pares entre 2 y 10 inclusive:

```
int n;
for(n=2;n <= 10;n+=2) {
     printf("%d\n",n);
}
```

Modularidad

Función o SubProceso

PSeInt permite la programación modular, la idea es simple: dividir al problema en módulos; estos módulos son ejecutados desde el programa principal[1]; los módulos de software tienen un **objetivo** (se focalizan en resolver una parte "mínima" del problema[2]), tienen un **nombre**, pueden **devolver un único valor** (que puede ser guardado en una variable del programa principal), pueden tener parámetros (los cuales pueden ser pasados por valor o por referencia). La posibilidad de devolver un valor y de contar con parámetros le permiten a los módulos poder comunicarse entre sí. Se puede asumir que los **parámetros** son la **entrada**, el **objetivo** es el **proceso** y el **valor de retorno** es la **salida**. Un **módulo** puede pensarse **como** una **caja negra**, independiente del resto de los módulos[3] y que puede comunicarse con el resto de los módulos a través de entradas y salidas bien definidas[4].

Un ejemplo simple

Supongamos que tenemos que registrar las ventas totales de un negocio que tiene 10 sucursales, ingresar por teclado los totales de ventas, sumarizarlos, mostrar los valores ingresados y el total general.

[1] Que, a su vez, también es un módulo de software!.
[2] El entrecomillado se refiere a que también podemos hacer módulos que dividan al problema en partes no tan pequeñas, pues, un módulo que resuelva una parte importante del problema podría ejecutar a otros módulos que resuelvan partes más pequeñas del problema
[3] Un módulo no debería hacer referencia a una variable global por ejemplo, pues ello impediría el reuso de este módulo en otro contexto (si en el otro contexto no existe dicha variable global, el modulo no funcionaría).
[4] Es lo que llamamos "interfase" o API (application program interface), tal vez, este punto sea el de mayor dificultad: "pensar en cuál es la mejor interfase para que este módulo cumpla con su objetivo y permita una comunicación fácil y efectiva con el resto de los módulos, facilitando luego su reutilización en otros contextos"

Implementación tradicional

```
Proceso no_modular
     Dimension ventas(10)
     Para i<-1 Hasta 10 Hacer
          Escribir 'Ingrese total ventas ',i,': '
          Leer ventas(i)
     FinPara
     suma<-0
     Para i<-1 Hasta 10 Hacer
          suma<-suma+ventas(i)
     FinPara
     Para i<-1 Hasta 10 Hacer
          Escribir 'ventas(',i,') = ',ventas(i)
     FinPara
     Escribir 'Total General ',suma
FinProceso
```

Implementación modular

Este problema lo podemos dividir en los siguientes subprocesos genéricos[5]:

-ingresar ventas (ingresar un vector de N elementos por teclado)

-sumarizar ventas (totalizar un vector de N elementos)

-mostrar ventas (mostrar por pantalla un vector de N elementos)

Ahora bien, analicemos cada subproceso, ¿qué entradas necesita? ¿qué salida se requiere de él? ¿qué nombre vamos a darle?:

[5] Modularizar, en parte también implica, generalizar!, yo sé que se trata de totales de ventas, pero, en realidad, desde el punto de vista del módulo, se trata de operaciones sobre un arreglo o vector de N elementos, nada más.

Subproceso	Nombre	Entrada	Salida
ingresar ventas	cargarVector	-vector a cargar por referencia[6] -largo del vector[7] -mensaje[8]	
sumarizar ventas	sumarVector	-vector a sumar por referencia -largo del vector	Sumatoria de todos los elementos del vector
mostrar ventas	mostrarVector	-vector a mostrar por referencia -largo del vector -mensaje	

Ahora bien, analicemos el modulo principal, ¿cómo utilizaríamos cada uno de los subProcesos?

```
dimension ventas(10)
cargarVector(ventas,10,'Ingrese total ventas ')
total<-sumarVector(ventas,10)
mostrarVector(ventas,10,'ventas')
Escribir 'Total General ',total
```

Esta forma de análisis del problema la podemos llamar "**down-top**" porque pensamos desde los subprocesos hacia el proceso principal. Pero también podría haberse pensado en forma inversa: desde cómo se usarían estos subprocesos en el proceso principal, para luego implementar cada subproceso; este enfoque se llama "**top-down**". Cada uno puede optar o combinar el enfoque que le sea más apropiado para resolver el problema modularmente. Veamos el pseudocódigo PSeInt:

```
Proceso si_modular
     dimension ventas(10)
     cargarVector(ventas,10,'Ingrese total ventas ')
     total<-sumarVector(ventas,10)
     mostrarVector(ventas,10,'ventas')
     Escribir 'Total General ',total
FinProceso

SubProceso cargarVector(vector por referencia,largo,mensaje)
     Para i<-1 Hasta largo Hacer
               Escribir mensaje,i,': '
               Leer vector(i)
     FinPara
FinSubProceso

SubProceso mostrarVector(vector por
referencia,largo,mensaje)
     Para i<-1 Hasta largo Hacer
               Escribir mensaje,'(',i,') = ',vector(i)
     FinPara
FinSubProceso

SubProceso suma<-sumarVector(vector por referencia,largo)
     suma<-0
     Para i<-1 Hasta largo Hacer
               suma<-suma+vector(i)
     FinPara
FinSubProceso
```

Obsérvese que los módulos son independientes y por lo tanto, reusables en otros programas similares en donde se deban realizar estas mismas operaciones sobre arreglos o vectores. También queda mucho más claro en dónde buscar un problema en caso de error.

Función o SubProceso sin valor de retorno

Veamos ahora un ejemplo de cómo traducir en lenguaje C una función o subproceso que no tiene valor de retorno:

```
SubProceso hagoAlgo
      …
FinSubProceso
```

puede traducirse en lenguaje C como:

```
void hagoAlgo();

int main(int argc, char **argv) {

      …
}

void hagoAlgo() {
      …
}
```

en lenguaje ANSI C estamos obligados a **declarar** el prototipo de la función a implementar (`void hagoAlgo();` en este caso) y luego **implementar** la función (por lo general, luego de la función `main()` se implementan todas las funciones, implementar implica usar { } , mientras que declarar solo finaliza con ;)

Función o SubProceso con valor de retorno

Veamos ahora un ejemplo de cómo traducir en lenguaje C una función o subproceso que tiene valor de retorno:

```
SubProceso suma<-sumaAlgo
     ...
FinSubProceso
```

puede traducirse en lenguaje C como (asumiendo que **suma** es un valor real decimal grande):

```
double sumaAlgo();

int main(int argc, char **argv) {
     …
}

double sumaAlgo() {
     …
}
```

Función o SubProceso con parametros por referencia

Veamos ahora un ejemplo de cómo traducir en lenguaje C una función o subproceso que tiene algún parámetro por referencia:

```
SubProceso hagoAlgo(vector por referencia)
     …
FinSubProceso
```

puede traducirse en lenguaje C como (asumiendo que vector se refiere a un arreglo con valores real decimales grandes):

```
void hagoAlgo(double *);

int main(int argc, char **argv) {
     …
}

void hagoAlgo(double *vector) {
     …
}
```

los parámetros por referencia en C se implementan como punteros. En el lenguaje C todos los parámetros son por valor, es decir, todos los argumentos que son pasados a una función se copian al espacio de memoria de la función. Entonces, para implementar el pasaje por referencia, se utilizan variables que en su interior contienen la dirección de memoria de otra variable (puntero).

Un ejemplo simple, implementación en lenguaje C

Volviendo al ejemplo simple propuesto anteriormente, la versión modular del mismo, en lenguaje C podría traducirse como:

```
#include <stdio.h>
#include <stdlib.h>
// prototipos de funciones/subprocesos
void cargarVector(double *,int,char *);
void mostrarVector(double *,int,char *);
double sumarVector(double *,int);
int main(int argc, char **argv) {
     double ventas[10],total;
     cargarVector(&ventas[0],10,"Ingrese total ventas");
     total=sumarVector(&ventas[0],10);
     mostrarVector(&ventas[0],10,"ventas");
     printf("Total General %lf\n",total);
     return 0;
}
void cargarVector(double *vector,int largo,char *mensaje) {
     int i;
     for(i=0;i < largo;vector++,i++) {
          printf("%s %d : ",mensaje,i);
          scanf("%lf",vector);
     }
}
void mostrarVector(double *vector,int largo,char *mensaje) {
     int i;
     for(i=0; i < largo;vector++,i++) {
          printf("%s (%d) = %lf\n",mensaje,i,*vector)
     }
}
double sumarVector(double *vector,int largo) {
     double suma=0.0;
     int i;
     for(i=0;i < largo; i++,vector++) {
          suma+=*vector;
     }
     return suma;
}
```

Obsérvese la lógica de punteros utilizada, el parámetro por referencia se indica como double * o char * (en este caso, las cadenas de caracteres también se pasan por referencia); desde el programa principal, se llaman a las funciones indicando la dirección de memoria del primer elemento del arreglo (&ventas[0]), los arreglos están ubicados en forma contigua en la memoria, con lo cual, dentro de cada módulo, solo es cuestión de ir incrementando el puntero para pasar de un elemento a otro del arreglo apuntado.

Atte. Guillermo Cherencio
Programación I
ISFT N° 189